CMPE 209 – Network Security and Applications

Homework3

First Name: Harish

Last Name: Marepalli

SJSU ID: 016707314

Professor: Dr. Younghee Park

TABLE OF CONTENTS

1. Differences between secret key cryptography and public key cryptography:

Secret key cryptography and public key cryptography are two different methods of encryption. Secret key cryptography, also known as symmetric-key cryptography, uses the same secret key for encryption and decryption. In contrast, public key cryptography, also known as asymmetric cryptography, uses two different keys – a public key and a private key.

When using secret key cryptography, the sender and receiver must use the same secret key to encrypt and decrypt messages. However, the key must be kept secure, as anyone who has access to it can read the message. Examples of secret key cryptography include AES and DES.

In contrast, public key cryptography uses a public key to encrypt messages and a private key to decrypt them. The public key can be distributed widely, while the private key must be kept secure. This means that only the intended recipient, who holds the private key, can read the message. Public key cryptography allows secure communication without requiring a pre-shared secret key, which is necessary in secret key cryptography.

One of the significant differences between secret key and public key cryptography is the method of key exchange. Secret key cryptography requires a secure channel for key exchange, while public key cryptography does not. With public key cryptography, anyone can send an encrypted message to the receiver using their public key, without needing to exchange any secret information beforehand. Public key cryptography also enables digital signatures and key exchange, which are not possible with secret key cryptography.

Answer in the form of a table:

	Secret key cryptography	**Public key cryptography**
Also known as	Symmetric-key cryptography	Asymmetric cryptography
Key type	One shared secret key	Two keys: public and private
Key exchange	Requires a secure channel for key exchange	Key exchange can be done publicly
Encryption/Decryption speed	Fast	Slower than secret key cryptography
Key management	Requires fewer keys to manage	Requires more keys to manage
Security	Less secure since the same key is used for encryption and decryption	More secure since the private key is kept secret
Usage	Generally used for encrypting small amounts of data	Widely used for secure communication, digital signatures, and key exchange
Examples	AES, DES	RSA, ECC

2. Message authentication code and 3 approaches to message authentication:

A message authentication code (MAC) is a cryptographic technique used to verify the integrity and authenticity of a message. A MAC is a small piece of data that is generated using a secret key and appended to the message to protect against tampering and forgery. The receiver can use the same secret key to verify the MAC and ensure that the message has not been altered.

There are three main approaches to message authentication:

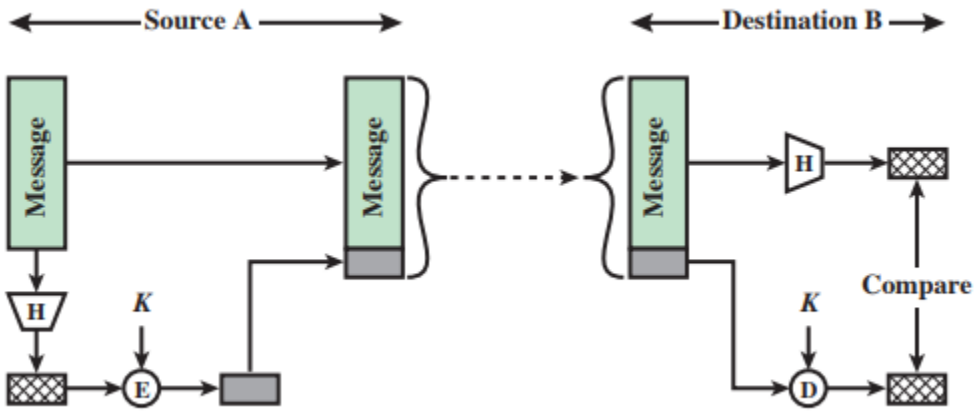
Symmetric-key MAC: In this approach, the sender and receiver share a secret key that is used to generate and verify the MAC. The steps for creating a symmetric-key MAC are:

a. The sender calculates the MAC for the message using a cryptographic hash function and the shared secret key.

b. The sender sends the message along with the MAC to the receiver.

c. The receiver recalculates the MAC for the received message using the same cryptographic hash function and the shared secret key.

d. The receiver compares the calculated MAC with the received MAC to verify the authenticity and integrity of the message.



(a) Using symmetric encryption

Public-key MAC: In this approach, the sender uses the receiver's public key to encrypt the MAC and append it to the message. The receiver uses their private key to decrypt the MAC and verify the authenticity and integrity of the message. The steps for creating a public-key MAC are:

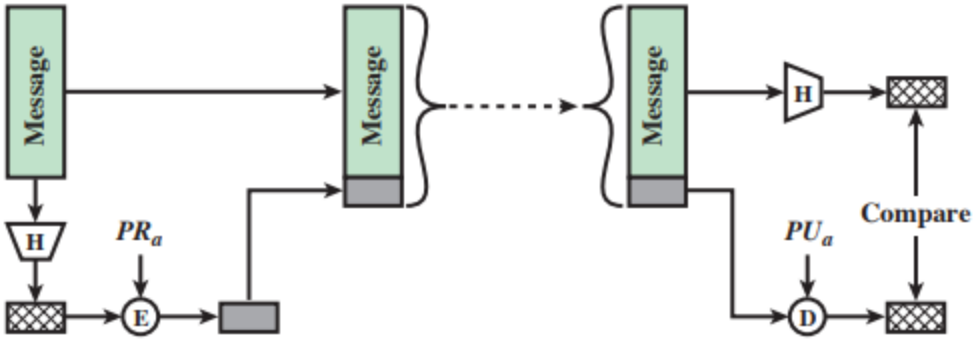
a. The sender calculates the MAC for the message using a cryptographic hash function and their secret key.

b. The sender encrypts the MAC using the receiver's public key and appends it to the message.

c. The receiver decrypts the encrypted MAC using their private key.

d. The receiver calculates the MAC for the received message using the same cryptographic hash function and the sender's public key.

e. The receiver compares the calculated MAC with the decrypted MAC to verify the authenticity and integrity of the message.



(b) Using public-key encryption

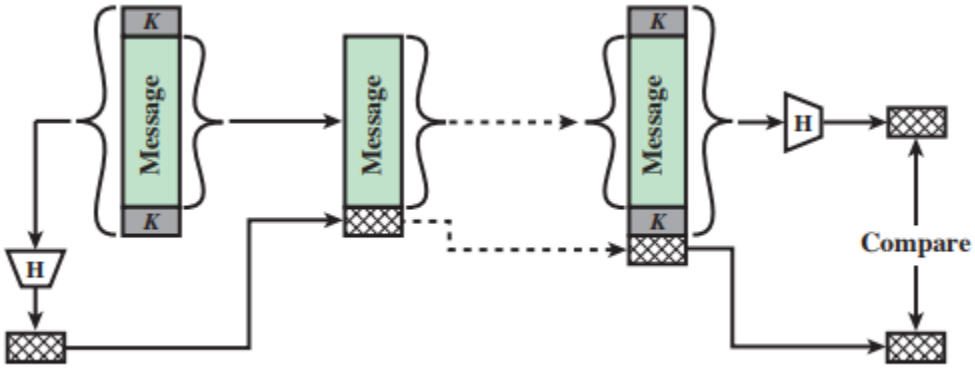
Hash-based/Secret value MAC: In this approach, a secret key and a hash function are used to generate the MAC. The steps for creating a hash-based MAC are:

a. The sender calculates the MAC for the message using a cryptographic hash function and the shared secret key.

b. The sender sends the message along with the MAC to the receiver.

c. The receiver recalculates the MAC for the received message using the same cryptographic hash function and the shared secret key.

d. The receiver compares the calculated MAC with the received MAC to verify the authenticity and integrity of the message.



(c) Using secret value

3. Hash function useful properties for message authentication:

A hash function is a mathematical function that takes an input (a message or data) and produces a fixed-length output, called a hash or message digest. Hash functions are commonly used in message authentication to ensure the integrity of the message, which means that the message has not been altered or tampered with during transmission. In order for a hash function to be useful for message authentication, it must have the following desirable properties:

Collision resistance: This property means that it is computationally infeasible to find two different inputs that produce the same output hash value. In other words, it is extremely difficult to find two messages with the same hash value. This property is important for message authentication because if an attacker can find two

messages with the same hash value, they can modify one message and replace it with the other without detection.

Preimage resistance: This property means that it is computationally infeasible to find an input that produces a given output hash value. In other words, given a hash value, it should be extremely difficult to find the message that produced it. This property is important for message authentication because if an attacker can find a different input that produces the same hash value as the original message, they can replace the original message with the new one without detection.

Message digest size: The size of the hash value produced by the hash function should be fixed and relatively small. A larger message digest size provides greater resistance to collision attacks, but also requires more storage space and longer computation time. A smaller message digest size may be faster to compute and require less storage space, but it may also increase the likelihood of collision attacks.

Deterministic: The hash function should produce the same hash value for the same input message every time it is computed. In other words, the hash function must be deterministic. This property is important for message authentication because the receiver must be able to compute the same hash value as the sender in order to verify the integrity of the message.

Non-reversible: Given the hash value produced by a hash function, it should be computationally infeasible to determine the original input message. This property is important for message authentication because an attacker should not be able to recover the original message even if they know the hash value.

4. Finding a pair of a private key and a public key in RSA:

To generate a pair of private and public keys in RSA, you can follow these steps:

1. Choose two large prime numbers, p and q.
2. Calculate N = p * q, which is the modulus for the public and private keys.
3. Calculate φ(N) = (p-1) * (q-1), where φ is Euler's totient function.
4. Choose a value for e that is $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$. e is the public key exponent.
5. Calculate d, the modular multiplicative inverse of e modulo φ(n) such that $d*e \equiv 1 \pmod{\phi(N)}$. d is the private key exponent.
6. The public key is (N, e) and the private key is (N, d).

Here is a step-by-step guide on how to find the private key (d) in RSA given p = 11, q = 3, and e = 3:

1. Calculate N = p * q = 11 * 3 = 33. This is the modulus for the public and private keys.
2. Calculate φ(N) = (p-1) * (q-1) = 10 * 2 = 20.
3. Check that e = 3 is relatively prime to φ(n) = 20. Since 3 and 20 have no common factors other than 1, e is relatively prime to φ(n).
4. Choose e = 3 as the public key exponent.
5. Calculate d, the modular multiplicative inverse of e modulo φ(n). To calculate d, we can use the extended Euclidean algorithm:
    i. Equation: $e * d \equiv 1 \pmod{\varphi(n)}$
    ii. Plug in the values of e, φ(n), and n: $3 * d \equiv 1 \pmod{20}$
    iii. The algorithm starts by dividing the larger number (20 in this case) by the smaller number (3 in this case) and then finding the remainder:
    $20 = 6 * 3 + 2$

iv. Now, the algorithm divides the previous divisor (3 in this case) by the remainder (2 in this case) to find a new quotient and remainder:
$3 = 2 * 1 + 1$

v. Apply back substitution (second part of extended Euclidean)
Substitute the previous remainder (2 in this case) into the equation and solve for it in terms of the previous quotient and remainder:
$2 = 3 - 1 * 2$

vi. Repeat the process by substituting the previous remainder (2 in this case) into the equation until we get a remainder of 1:
$1 = 3 - 1 * (20 - 6 * 3) = 7 * 3 – 20$
Here, the solution is found for the equation $d * e \equiv 1 \pmod{\varphi(N)}$, which is $d = 7$.

6. For the above example, the public key is (N, e) = (33, 3) and the private key is (N, d) = (33, 7).

5. Use of Public-key encryption to distribute a secret key:

In the Diffie-Hellman key exchange protocol, both parties use a common public parameter and their private keys to generate a shared secret key. This shared secret key can then be used as a symmetric key for encryption and decryption of messages.

Here's how the Diffie-Hellman key exchange protocol works:

1. Alice and Bob agree on a common public parameter, such as a large prime number p and a primitive root g modulo p.
2. Alice chooses a random secret value a, and Bob chooses a random secret value b.
3. Alice calculates A = g^a mod p and sends A to Bob.
4. Bob calculates B = g^b mod p and sends B to Alice.
5. Alice calculates the shared secret key K = B^a mod p.
6. Bob calculates the shared secret key K = A^b mod p.

Now, both Alice and Bob have the same shared secret key K, which they can use as a symmetric key for encryption and decryption of messages.

There is one more method as mentioned in the hint, is the public-key distribution of a secret key. Below is a simple example of public-key distribution of a secret key.

Assuming Alice and Bob want to share a secret key for communications, they can use public-key encryption to distribute the secret key using the following steps:

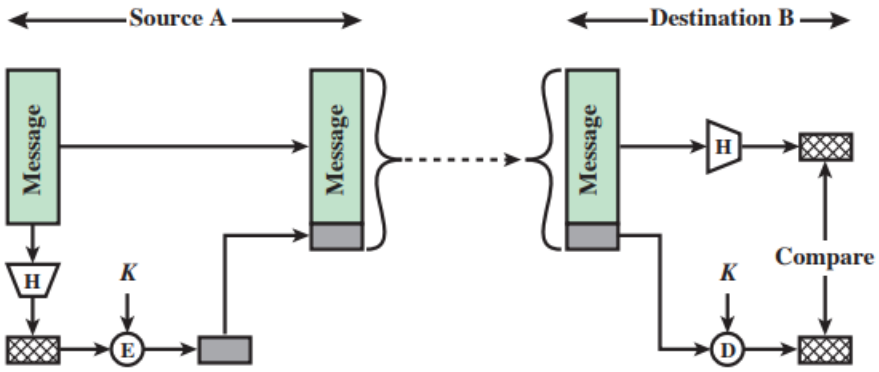
1. Alice and Bob both generate a public-private key pair for themselves.
2. Alice and Bob agree on a public key encryption algorithm and exchange their public keys with each other.
3. Alice generates a random secret key to be used for their communication and encrypts it using Bob's public key.
4. Alice sends the encrypted secret key to Bob.
5. Bob decrypts the encrypted secret key using his private key, which only he knows. Now both Alice and Bob have the same secret key that they can use to communicate with each other securely.

Both of these methods have advantages and disadvantages. In general, the Diffie-Hellman key exchange protocol is considered to be more secure than the public-key encryption method for distributing secret keys, because it does not rely on the security of the public key encryption algorithm, and instead relies on the discrete logarithm problem. However, the public-key encryption method is more versatile and can be used in a wider range of scenarios.

6. Procedures of sending and receiving messages:

1. A and B can achieve their goal of ensuring the integrity and authenticity of the messages with symmetric cryptography by using a message authentication code (MAC).

Below is the diagram that show the procedures of sending and receiving messages.



(a) Using symmetric encryption

The procedure for sending and receiving messages with a MAC is as follows:

i. Sender (A) generates a MAC by applying a cryptographic hash function to the message along with a secret key (K). This produces a fixed-size MAC.

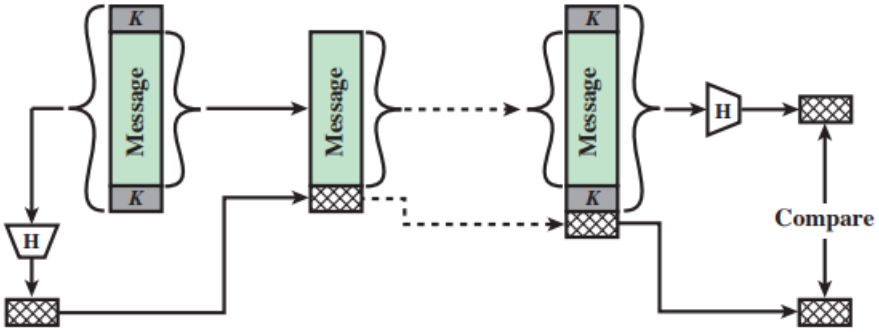
ii. Sender appends the MAC to the message and sends it to the receiver (B).

iii. Receiver (B) also generates a MAC using the same hash function and the same secret key (K).

iv. Receiver appends this MAC to the received message and then compares it to the MAC sent by the sender. If the two MACs match, the receiver knows that the message has not been tampered with during transmission.

2. A and B can achieve their goal of ensuring the integrity and authenticity of the messages with a hash function (e.g., MD5) using a secret value by using a Message Authentication Code (MAC).

Below is the diagram:



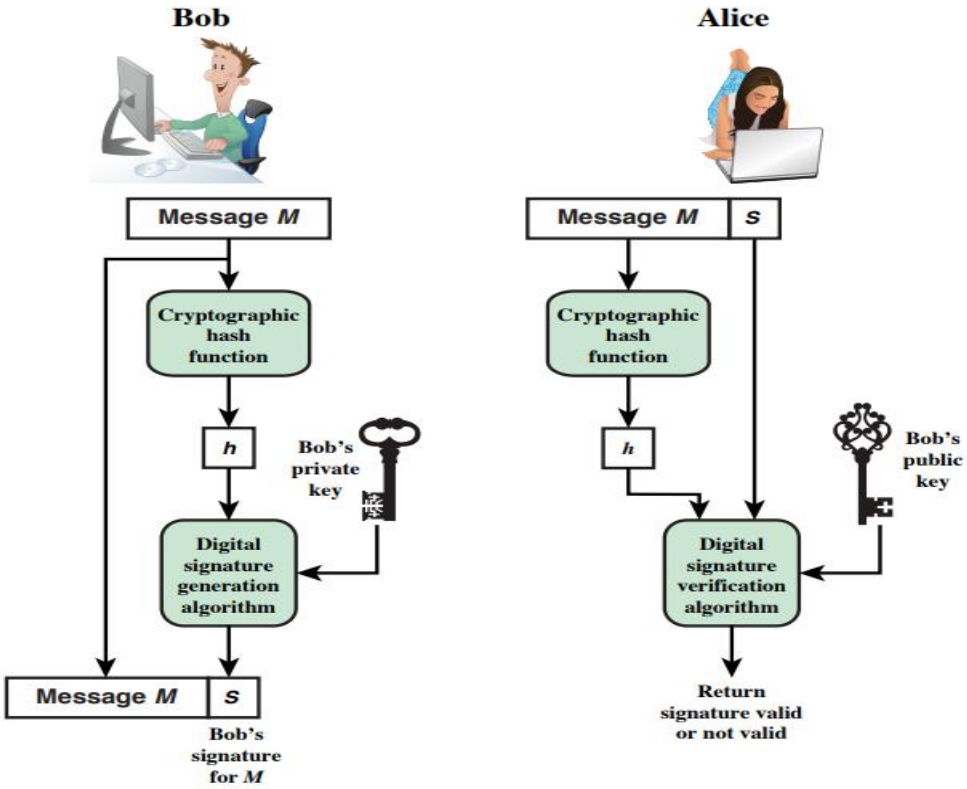
(c) Using secret value

In this method, a secret value (K) is assumed between the sender and receiver. The sender prepends and appends this secret value to the message and the entire data is hashed using a hash function (H) to get a hash value. This hash value is tagged to the message and sent to the destination. At the destination, the receiver again prepends and appends the secret value to the message and calculates the hash function to get certain value. This value is then compared to the hash that is tagged to the message. If same, then the message is authentic else, it is not.

3. No, they cannot get a non-repudiation service by using symmetric cryptography alone. Non-repudiation is the assurance that the sender of a message cannot deny sending the message. This requires the use of digital signatures, which require the use of asymmetric cryptography.

4. To provide non-repudiation service, A and B can use digital signatures with asymmetric cryptography.

Below is the diagram of the procedure for sending and receiving messages with digital signatures is as follows:



(a) Bob signs a message

(b) Alice verifies the signature

This is the Bob and Alice example:

1. Sender (A) generates a hash of the message using a cryptographic hash function.

2. Sender encrypts the hash using their private key to create a digital signature, which is appended to the message.

3. Receiver (B) uses the sender's public key to decrypt the digital signature and obtain the hash of the message.

4. Receiver computes the hash of the received message and compares it to the decrypted hash. If the two hashes match, the receiver knows that the message has not been tampered with during transmission.

5. To prove non-repudiation, the receiver (B) can provide the message and the digital signature to a third party who has access to the sender's public key. This third party can verify the digital signature using the sender's public key and confirm that the sender sent the message.

7. SEED Labs:

1. Secret-Key Encryption Lab:

1. Task1 – Frequency Analysis:

a. Given freq.py in the Labsetup file. Figure 1 shows running the freq.py file and getting the frequency statistics for n-grams such as monograms, bigrams, and trigrams.

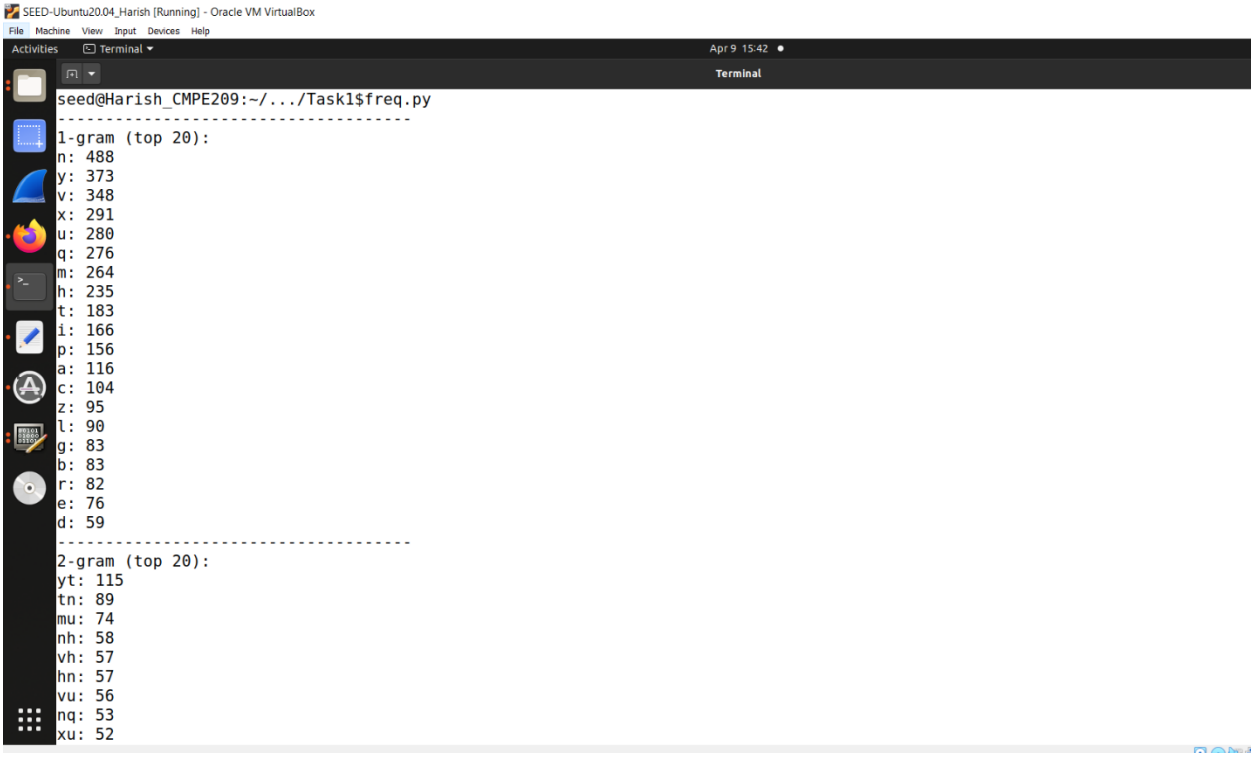


Fig. 1: Run freq.py

b. Figure 2 shows the continuation of the output.

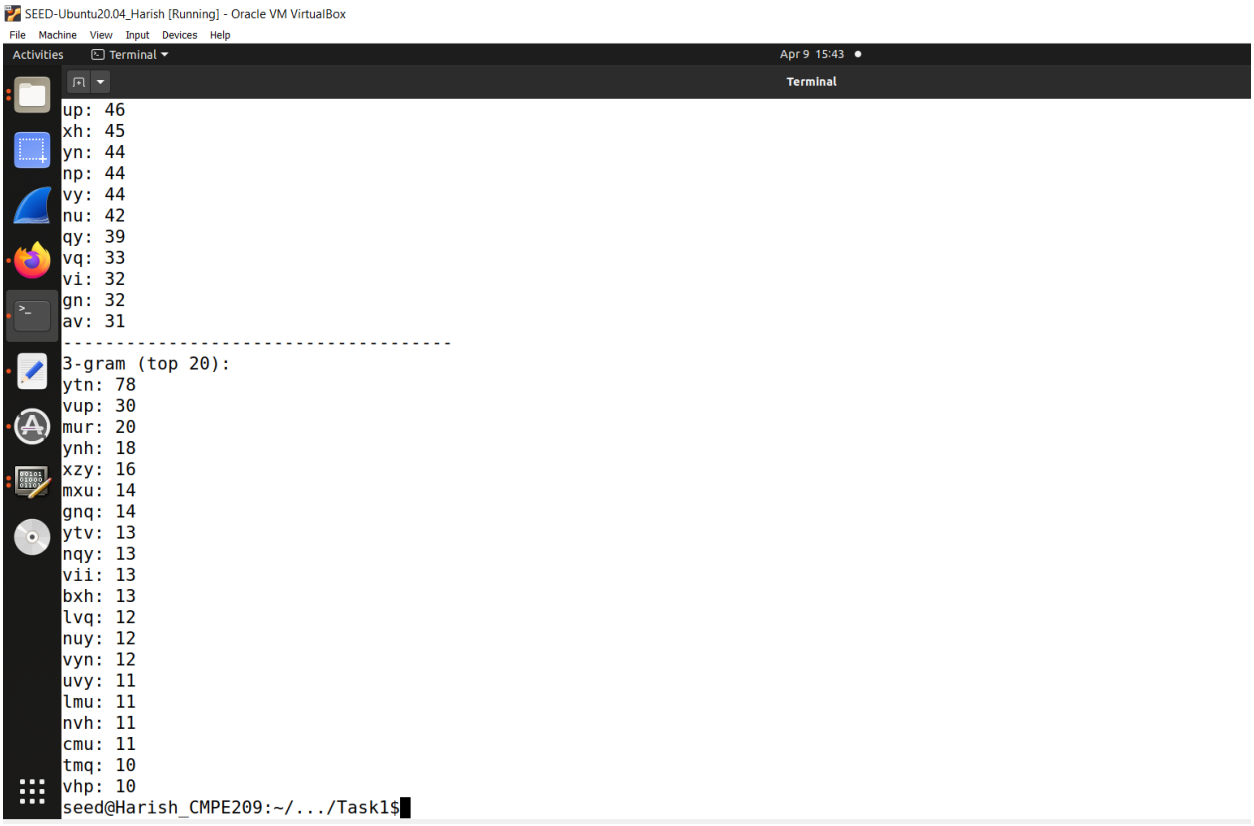


Fig. 2: Freq.py output continuation

c. From the output, it can be seen that the top 3 monograms are n, y, and v and the top 3 English language monograms are E, T, and A. The top 3 bigrams are TH, HE, IN and the top 3 trigrams are THE, AND, ING. By using these we can break it down into several phases to understand it better.

d. For the first phase, the cipher text to plain text conversion is:
n->E, y->T, t->H, v->A, u->N, p->D, m->I, r->G.

e. Second phase:
s->K, q->S, i->L, h->R.

f. Third phase:
x->O, g->B, b->F, e->P.

g. Fourth phase:
a->C, z->U, c->M, f->V, d->Y, l->W.

h. Fifth phase:
j->Q, o->J, k->X, w->Z.

i. Using the above process, converted the cipher text to plain text. Figure 3 shows the command for converting cipher to plain text.

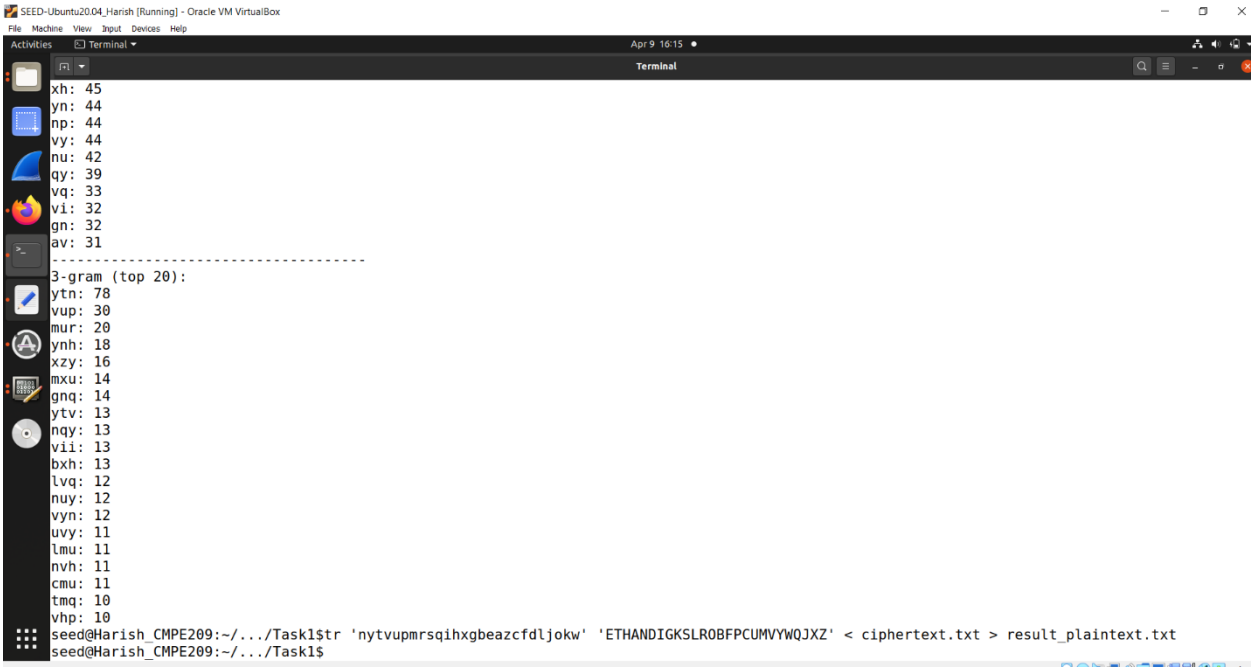


Fig. 3: Command for converting cipher to Plain text

j. Figure 4 shows the generated plain text.

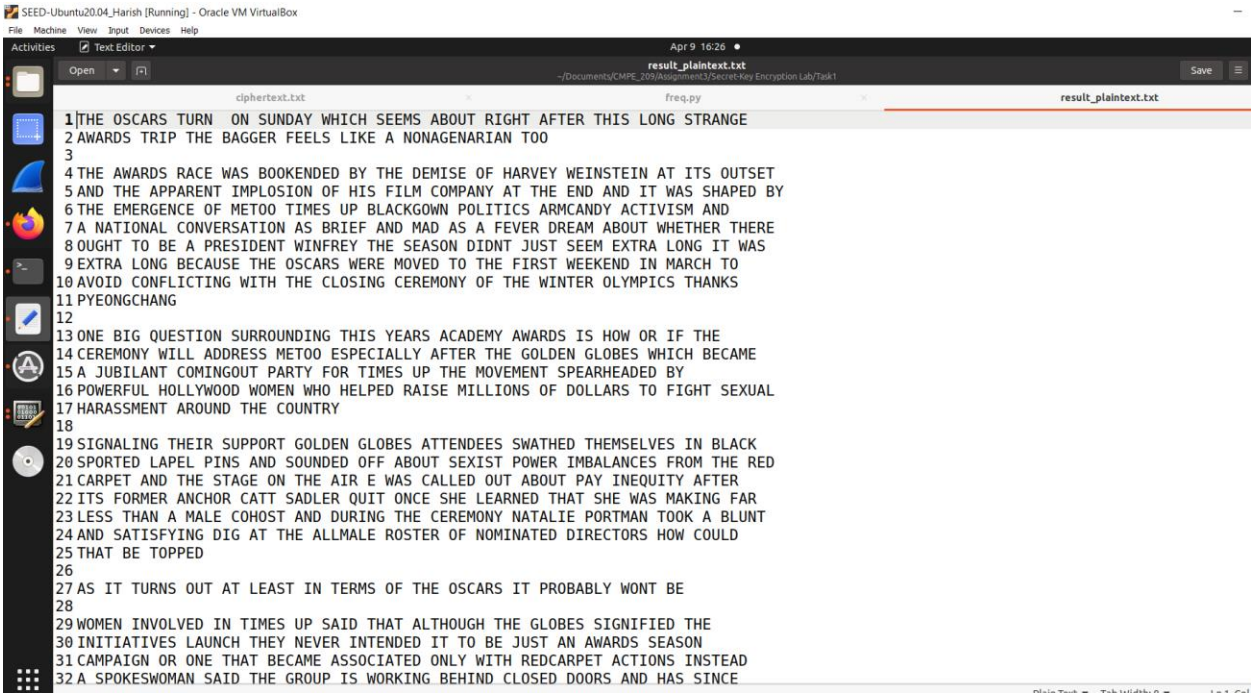


Fig. 4: Plain Text

2. Task2: Encryption using Different Ciphers and Modes:

a. Figure 5 shows the created plain text file.

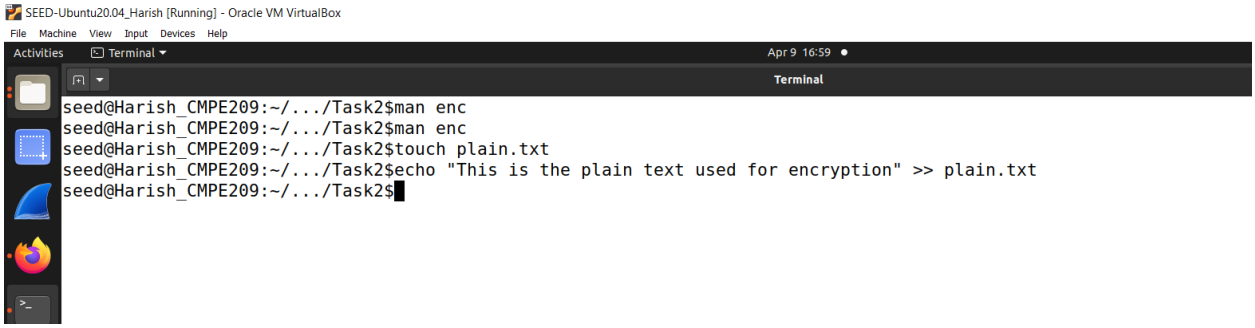


Fig. 5: Plain text file creation

b. Figure 6 shows the plain text.

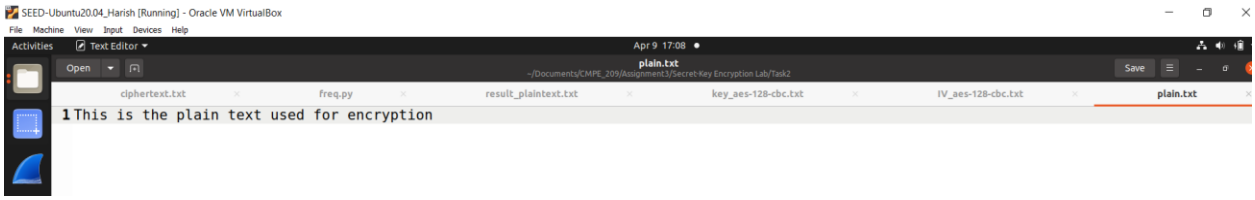


Fig. 6: Plain Text

c. Used openssl enc command to encrypt/decrypt a file. The below images are shown for AES-128-CBC mode. The cipher text length is 32 bytes. Figure 7 shows the Key/IV hex generation.

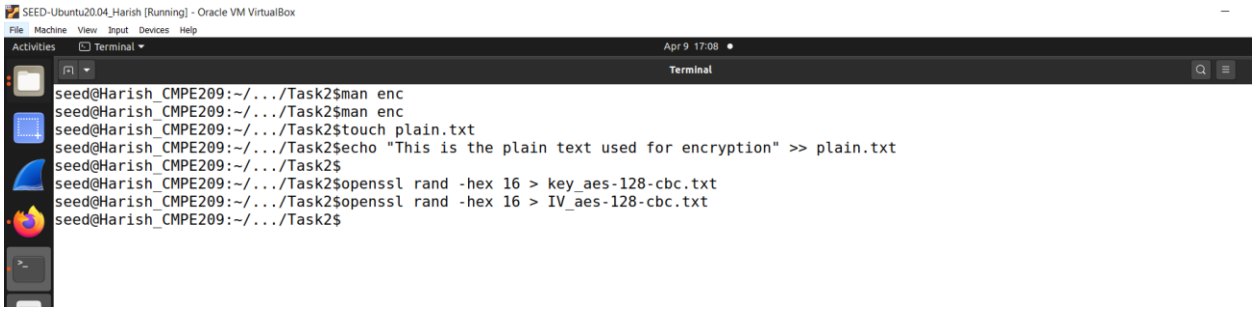


Fig. 7: Key/IV hex generation

d. Figure 8 shows the Key hex file.

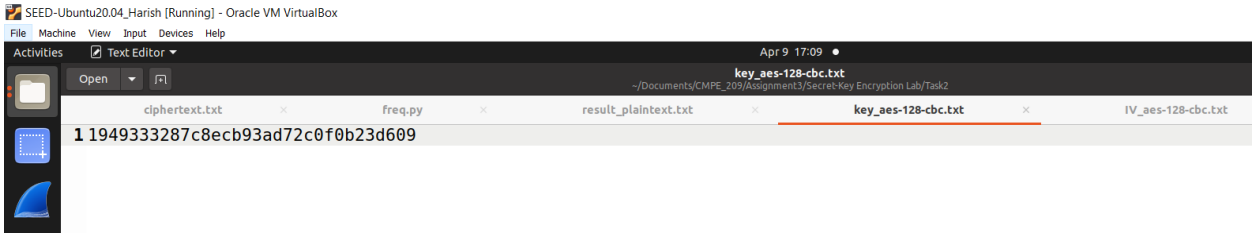


Fig. 8: Key hex file content

e. Figure 9 shows the IV hex file.

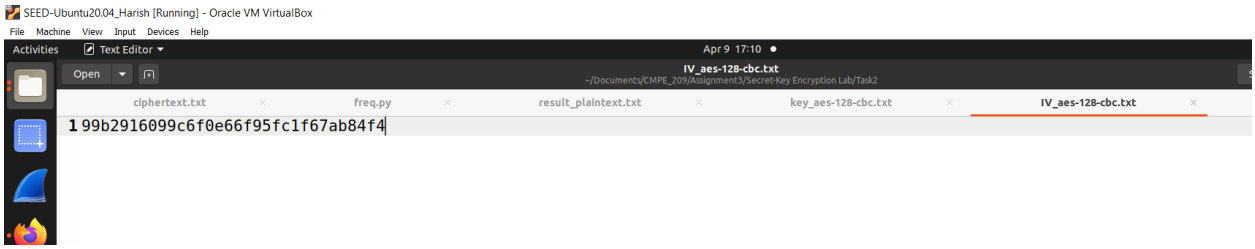


Fig. 9: IV hex file content

f. Figure 10 shows the encryption command.

Fig. 10: Encryption command

g. Figure 11 shows the cipher bin file that is created.

Fig. 11: Cipher bin file

h. Figure 12 shows the command for generation of decrypted text file.

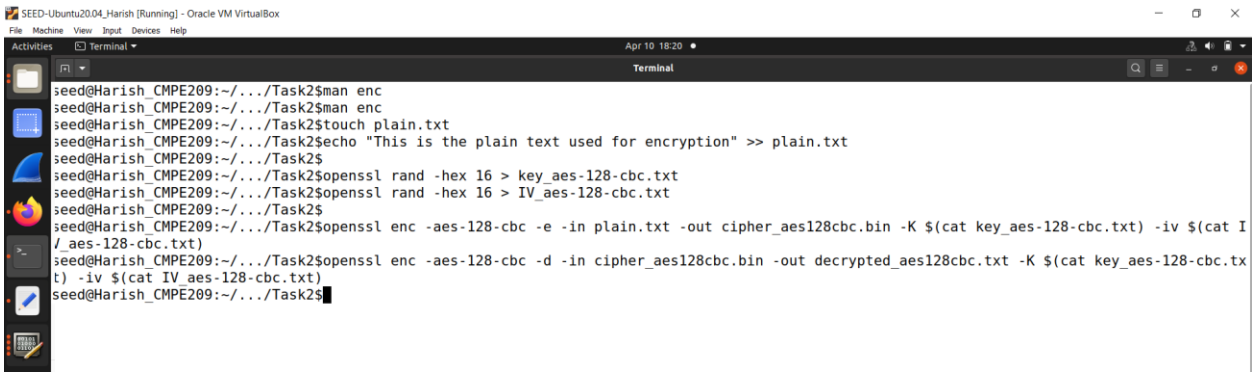


Fig. 12: Command for Decryption

i. Figure 13 shows the decrypted text.

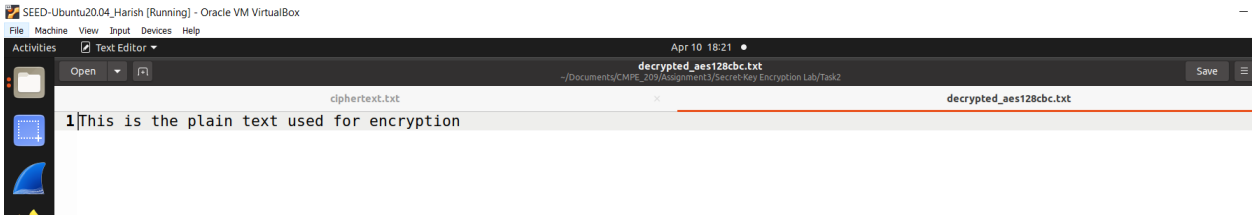


Fig. 13: Decrypted Text

j. The below images are shown for AES-128-CFB mode. The cipher text length is 23 bytes. Figure 14 shows the Key/IV hex generation and encryption command.

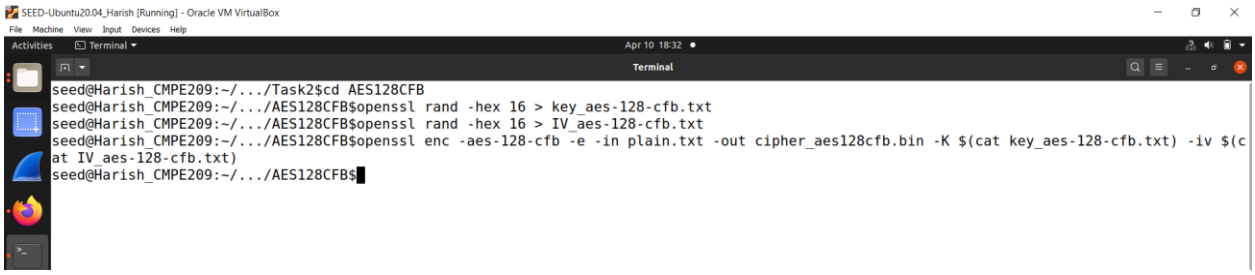


Fig. 14: Encryption command

k. Figure 15 shows the cipher bin file that is created.

Fig. 15: Cipher bin file

l. Figure 16 shows the command for generation of decrypted text file.

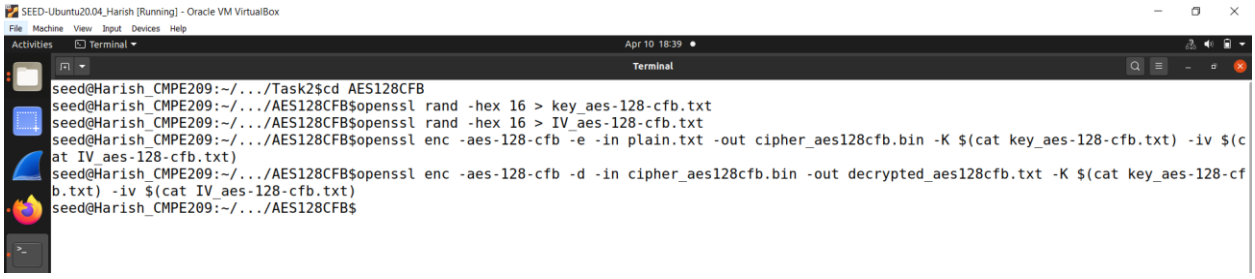


Fig. 16: Command for Decryption

m. Figure 17 shows the decrypted text.

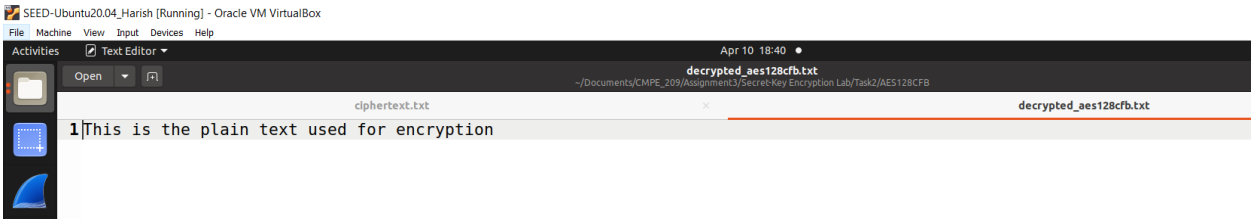


Fig. 17: Decrypted Text

n. The below images are shown for AES-128-CFB mode. The cipher text length is 24 bytes. Figure 14 shows the Key/IV hex generation and encryption command.

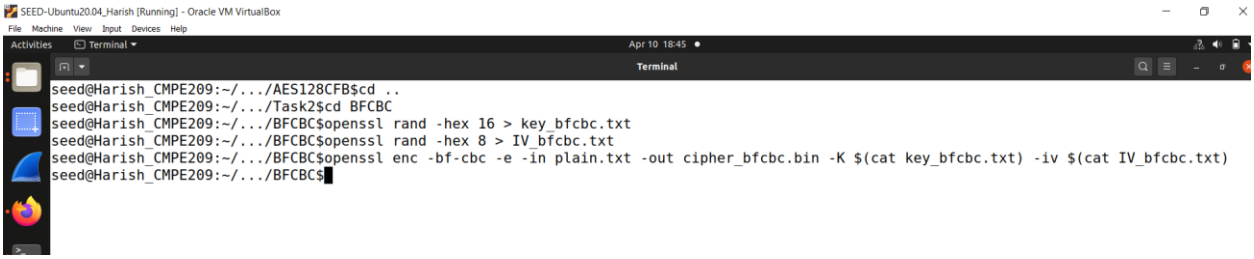


Fig. 18: Encryption command

o. Figure 19 shows the cipher bin file that is created.

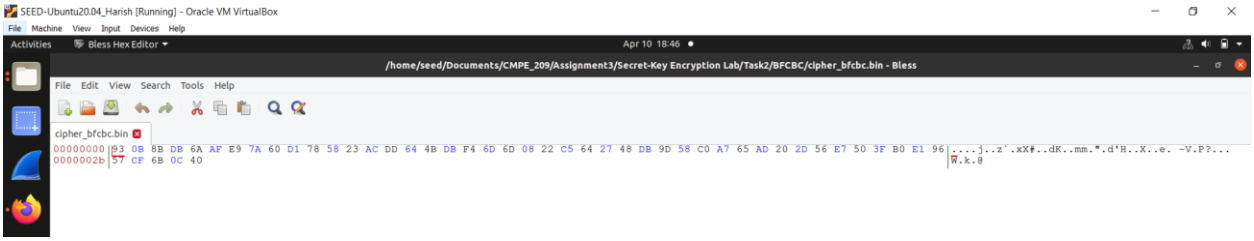


Fig. 19: Cipher bin file

p. Figure 20 shows the command for generation of decrypted text file.

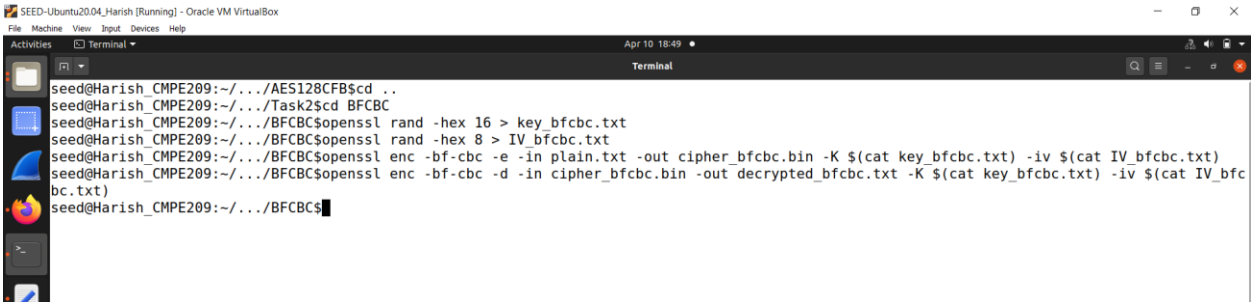


Fig. 20: Command for Decryption

q. Figure 21 shows the decrypted text.

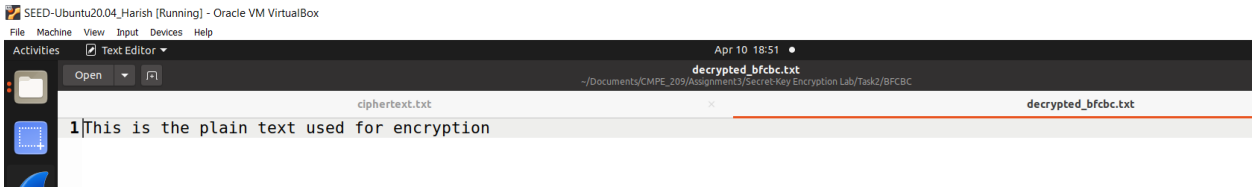


Fig. 21: Decrypted Text

3. Task3 – Encryption Mode – ECB vs. CBC:

a. Original pic bmp file is provided in the Labsetup folder. Figure 22 shows the original pic bmp file.

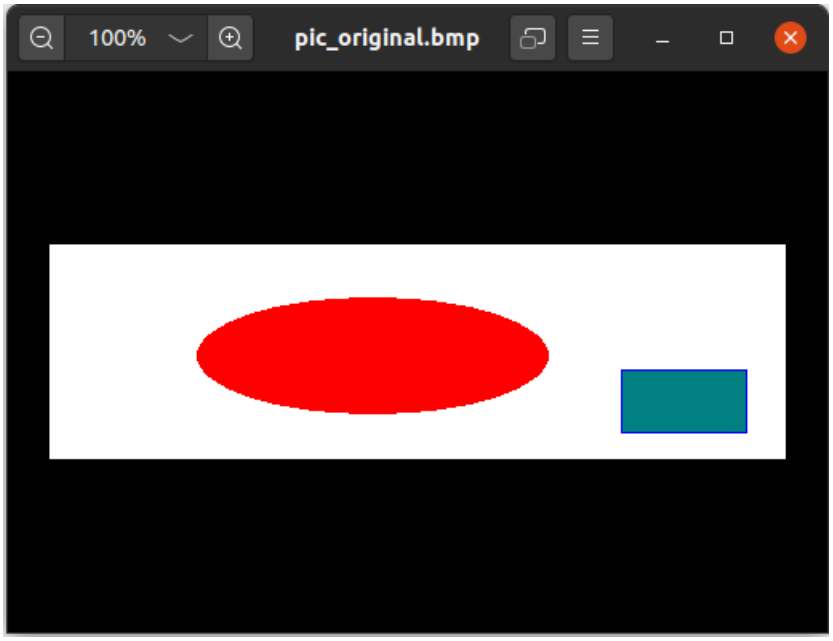


Fig. 22: Original Pic bmp file

b. The following snippets shows commands to encrypt the picture in AES-256-ECB mode. Figure 23 shows the encryption command that is run.

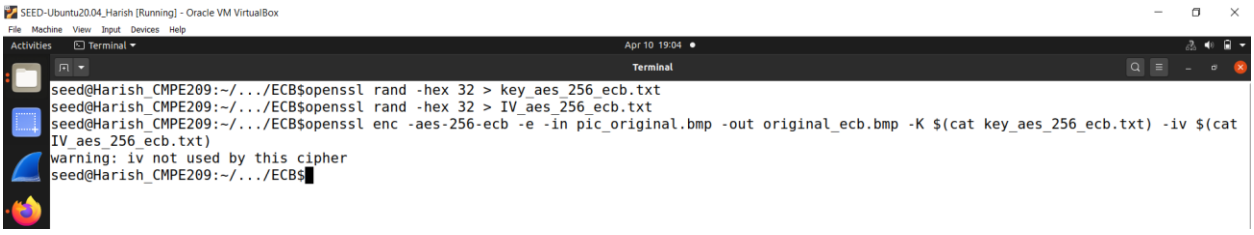


Fig. 23: Encryption command

c. There is a problem that bmp files carry a specific header, and it needs to be present for the bmp file to render the image properly. So, using head and tail commands, the first 55 bytes are extracted from original picture and the rest of the body from encrypted picture and combine them to form a new bmp file, which would be good to open. Figure 24 shows the generation of header ECB.

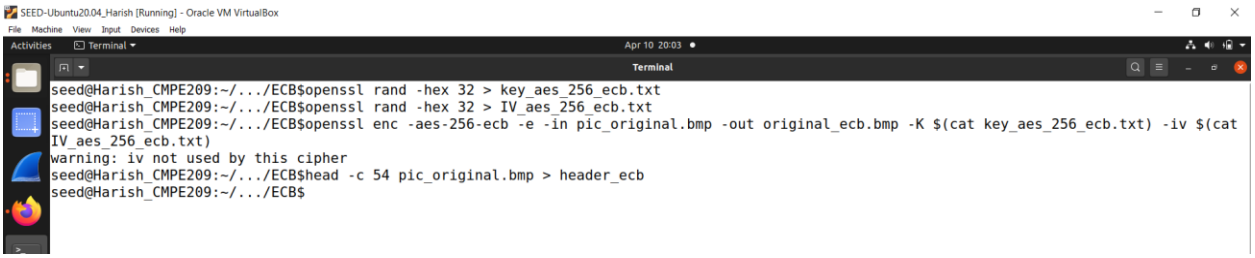


Fig. 24: Generation of Header ECB

d. Figure 25 shows the header ECB file.

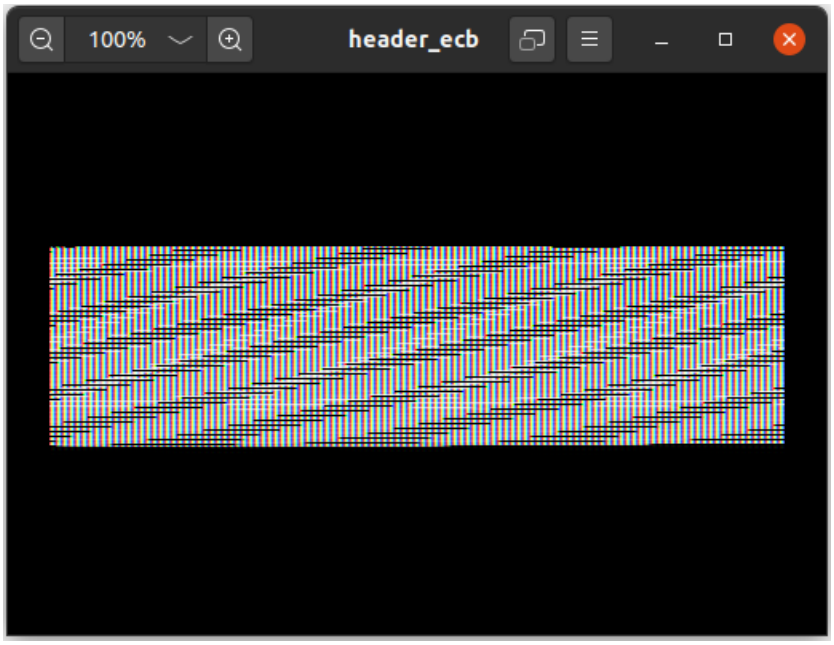


Fig. 25: Header ECB file

e. Figure 26 shows the generation of body ECB.

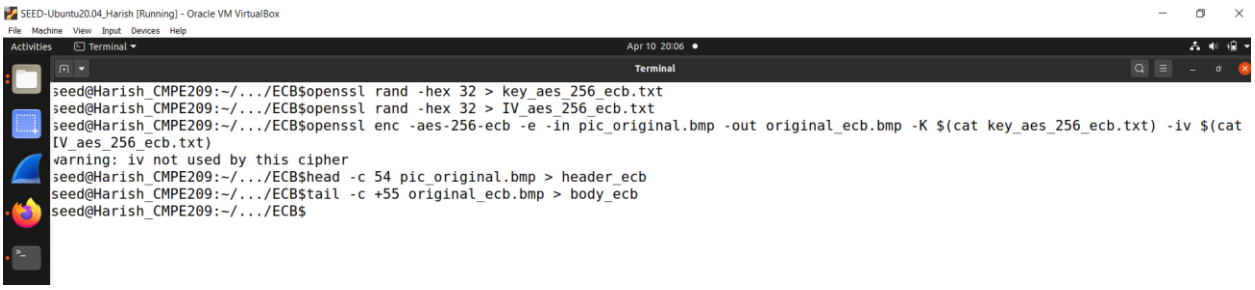


Fig. 26: Generation of body ECB

f. Figure 27 shows the body ECB file.

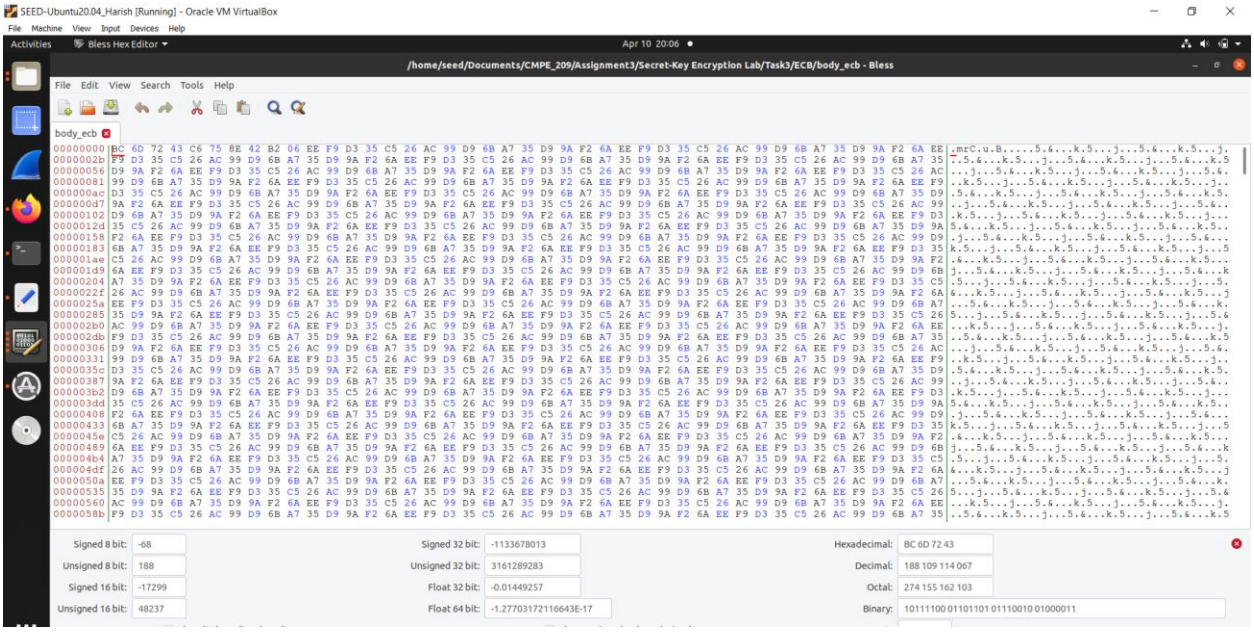


Fig. 27: Body ECB file

g. Figure 28 shows the generation of encrypted ECB.

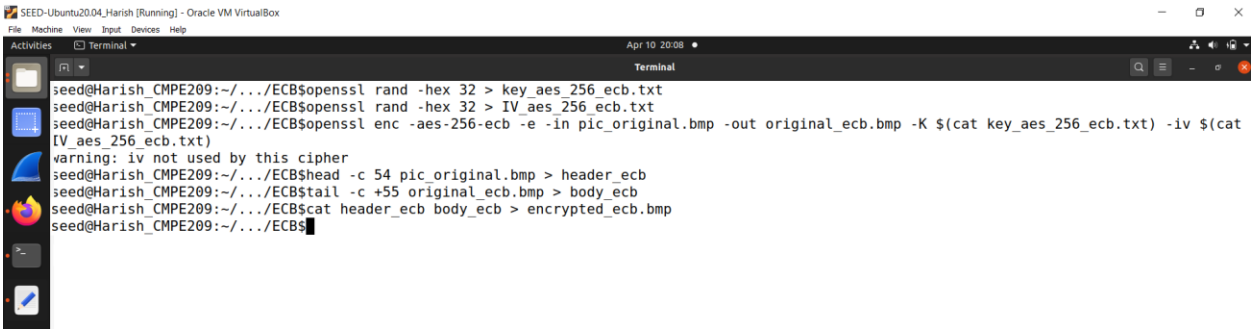


Fig. 28: Generation of encrypted ECB

h. Figure 29 shows the encrypted ECB.

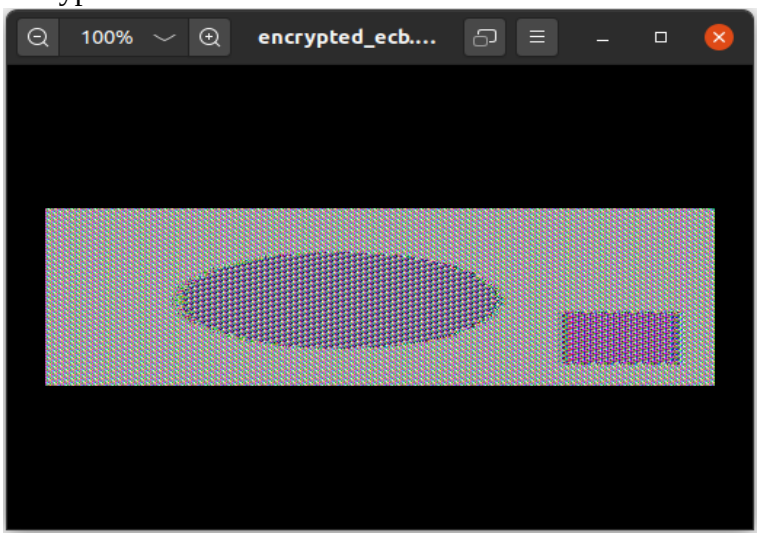


Fig. 29: Encrypted ECB

i. The following snippets shows commands to encrypt the picture in AES-256-CBC mode. Figure 30 shows the encryption command that is run.

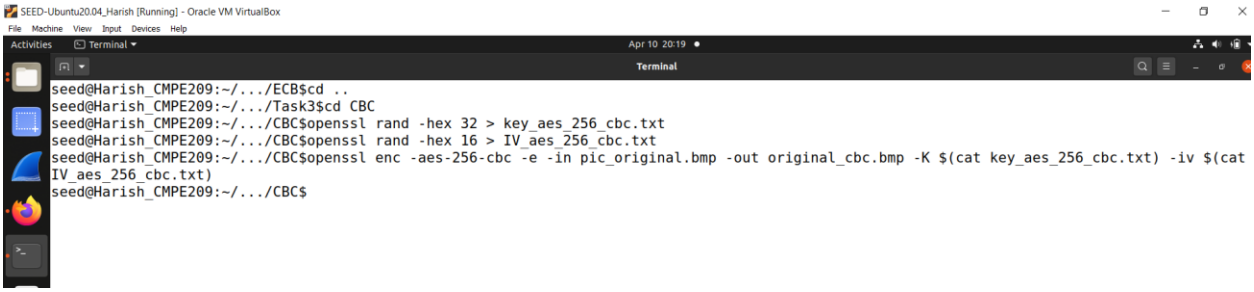


Fig. 30: Encryption command

j. Figure 31 shows the generation of header CBC.

Fig. 31: Generation of Header CBC

k. Figure 32 shows the header CBC file.

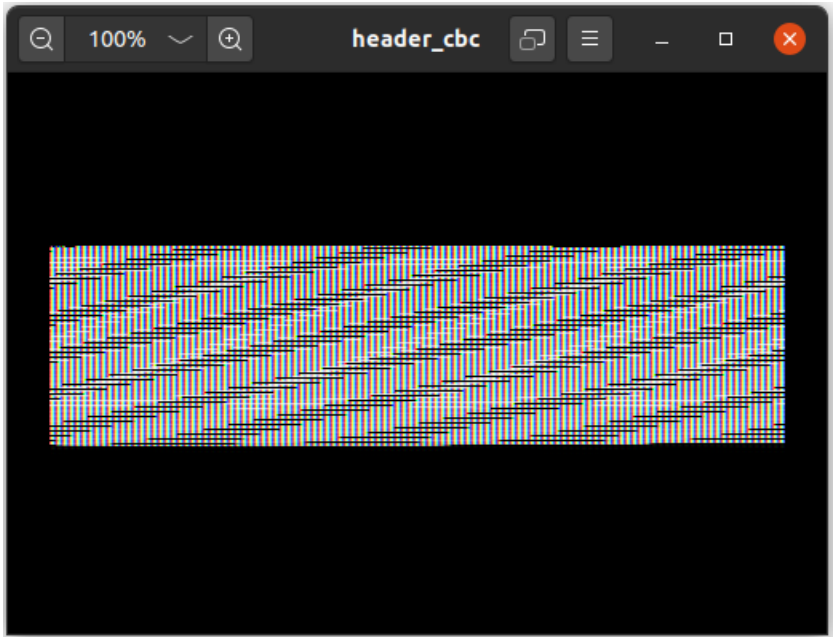


Fig. 32: Header CBC file

l. Figure 33 shows the generation of body CBC.

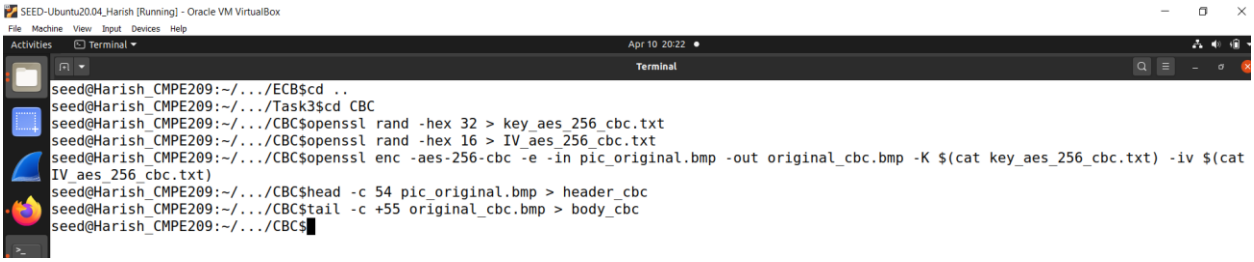


Fig. 33. Generation of body CBC

m. Figure 34 shows the body CBC file.

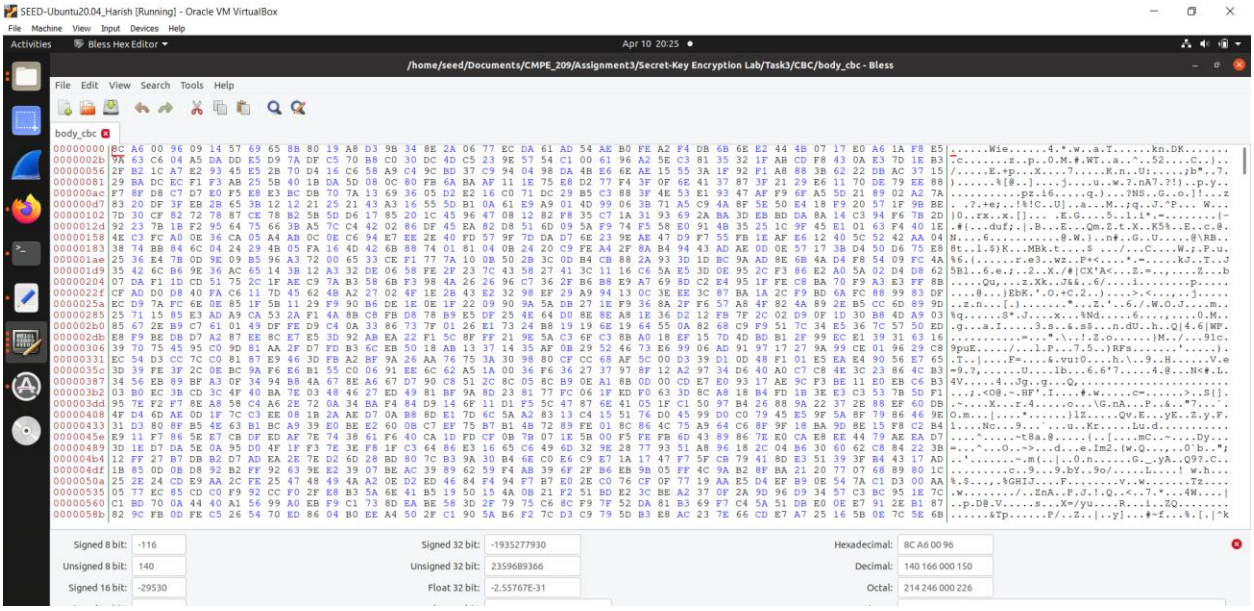


Fig. 34: Body CBC file

n. Figure 35 shows the Generation of encrypted CBC file.

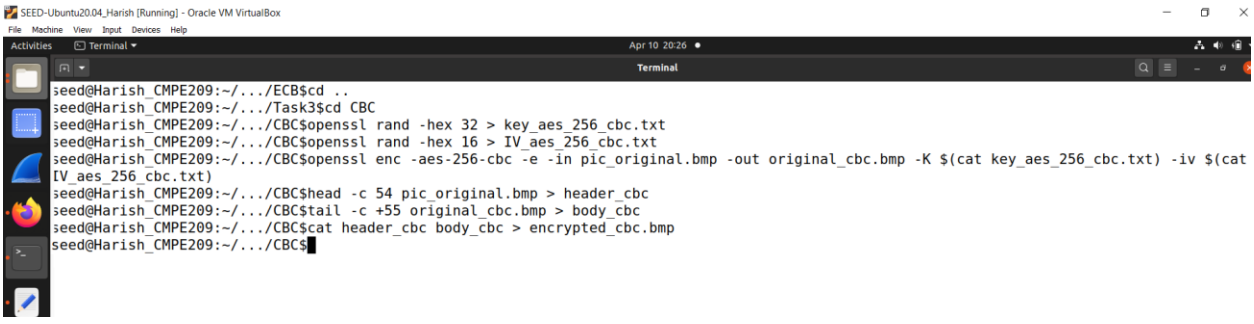


Fig. 35: Generation of Encrypted CBC file

o. Figure 36 shows the encrypted CBC file.

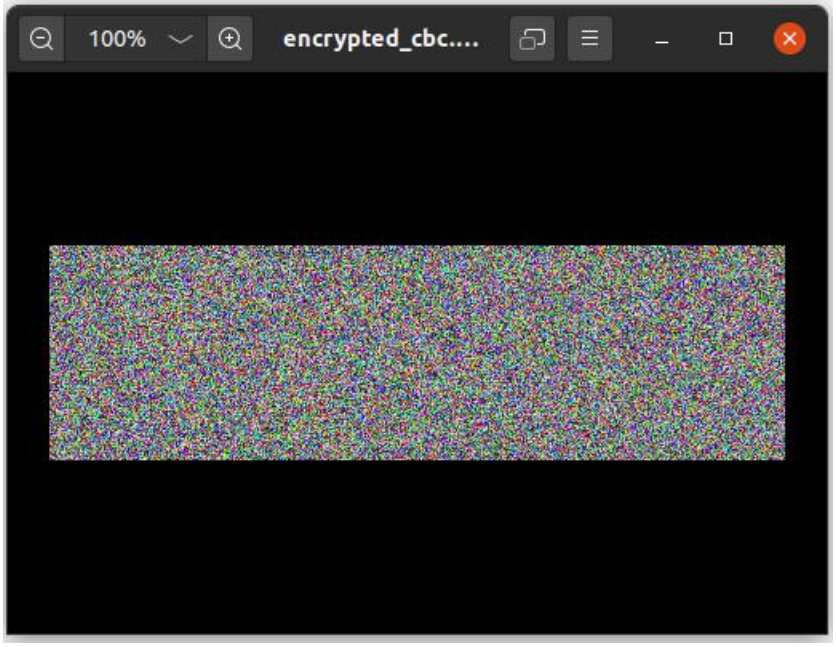


Fig. 36: Encrypted CBC file

Observations:
It is not possible to derive any useful information about the original picture from the encrypted picture, as the encryption process should ensure that the data is completely random and unreadable without the correct key. However, it is possible to observe the difference in the pattern of the encrypted picture between ECB and CBC modes. ECB mode tends to create a pattern of identical blocks, while CBC mode creates a more random-looking pattern. This can be seen when comparing the two encrypted pictures side by side.

2. RSA Encryption and Signature Lab:

1. Task 1: Deriving the Private Key:

   a. Here, we derive the private key, d. Figure 37 shows the CCode in which three BIGNUM variables a,b, and n are intitialized and computed a*b and (a^b mod n).

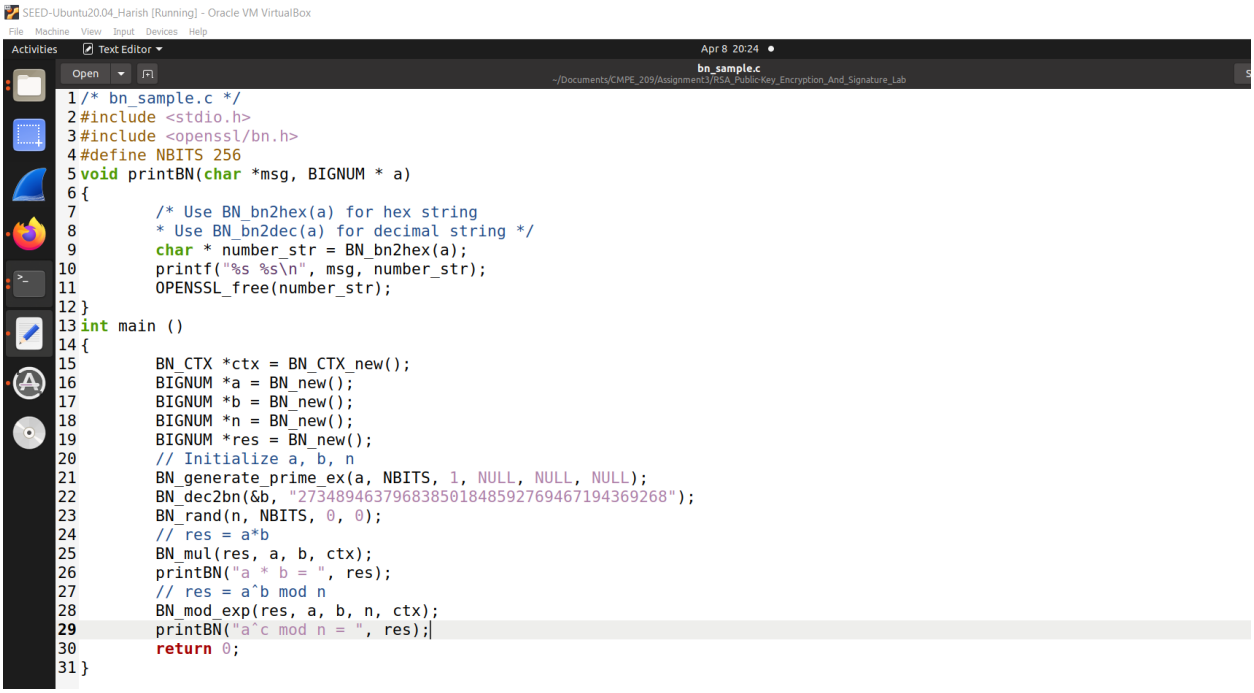


Fig. 37: CCode

b. Figure 38 shows the CCode compilation. We can compile bn sample.c with the following command (the character after - is the letter l, not the number 1; it tells the compiler to use the encryption library)

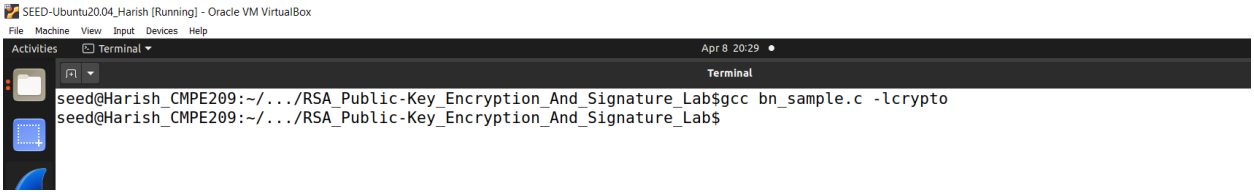


Fig. 38: CCode Compilation

c. Figure 39 shows the aoutfile created.

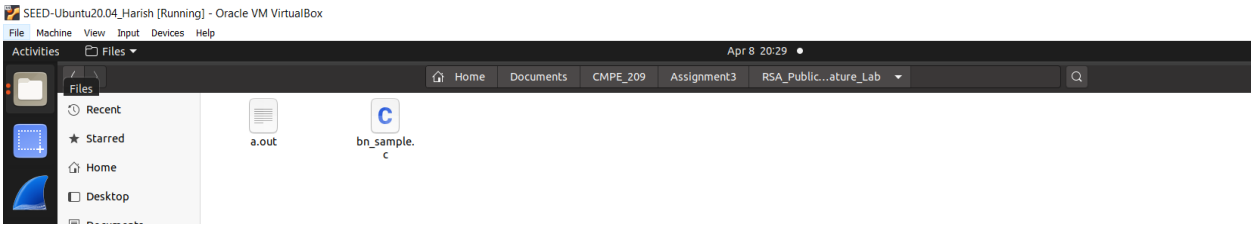


Fig. 39: aoutfile created

d. Task 1 starts here. Figure 40 shows the CCode for deriving a private key.

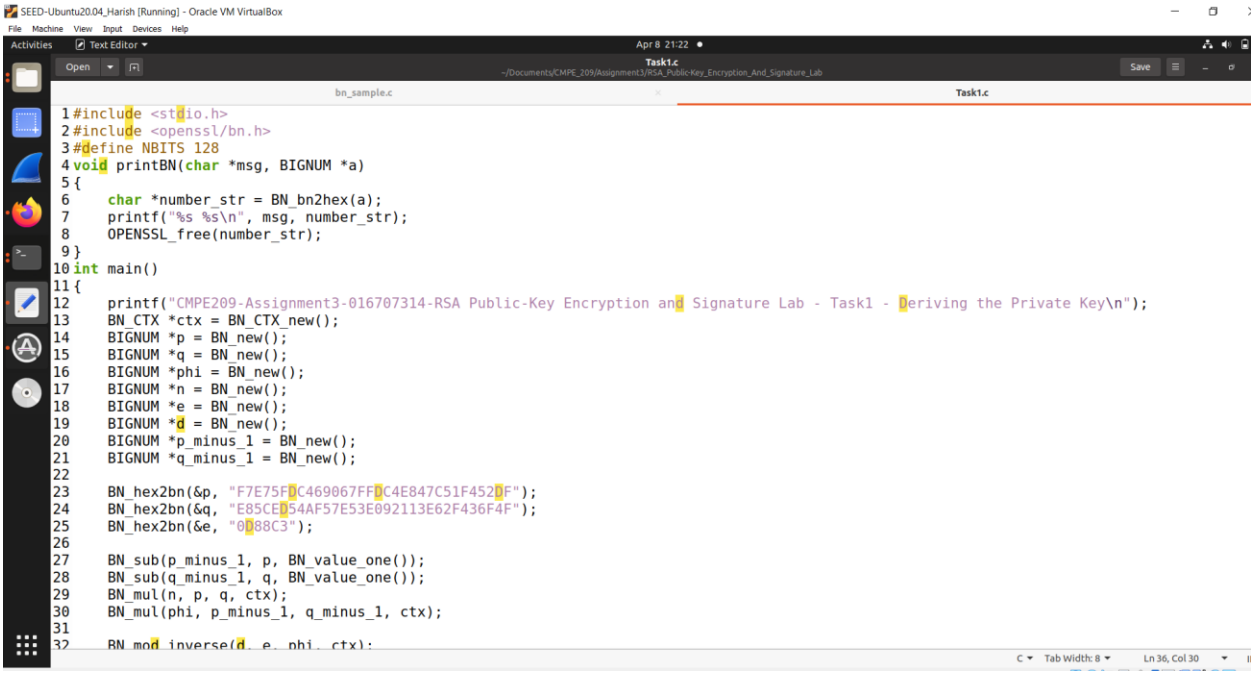


Fig. 40: CCode

e. Figure 41 shows the CCode continuation.

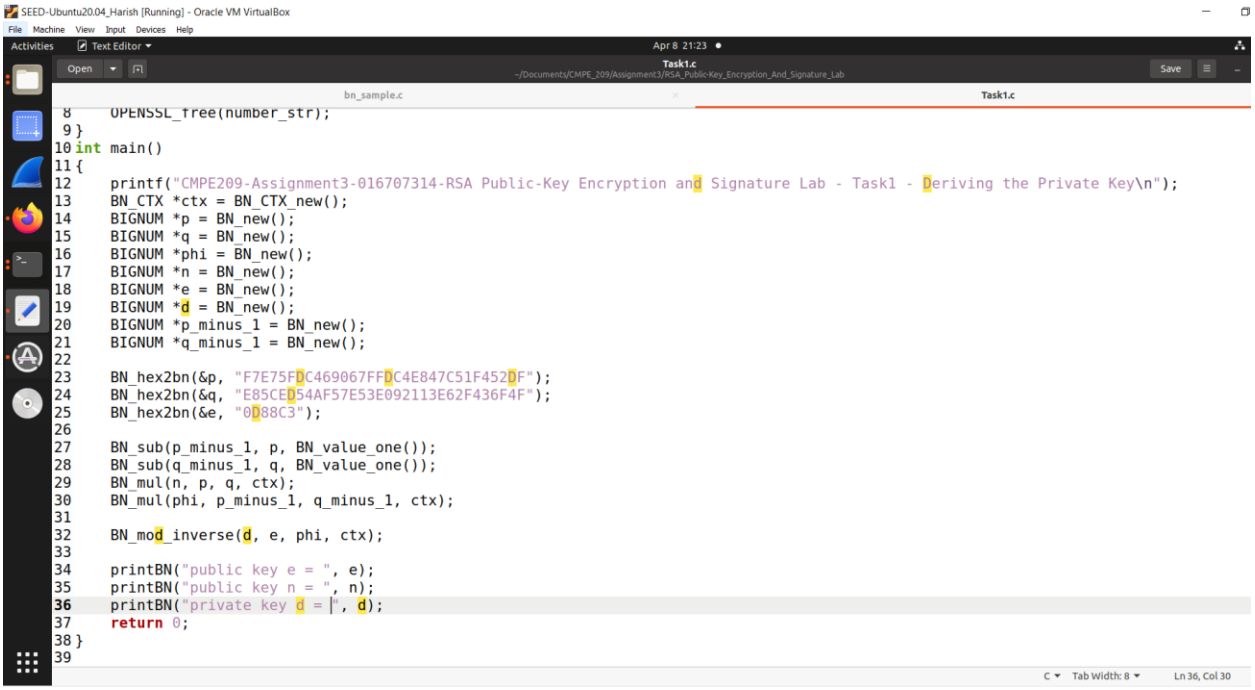


Fig. 41: CCode continuation

f. Figure 42 shows the CCode compilation.

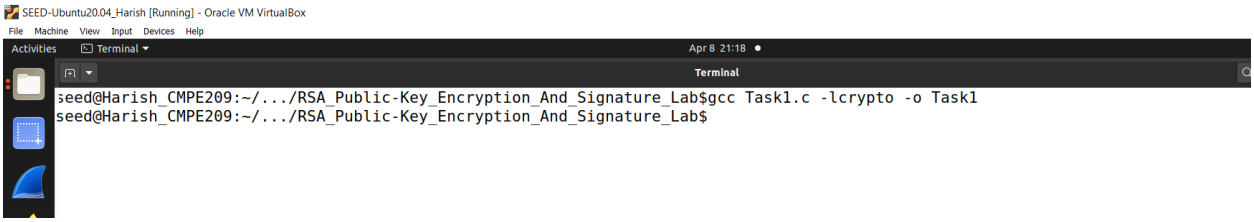


Fig. 42: CCode compilation

g. Figure 43 shows the Task1 file created.

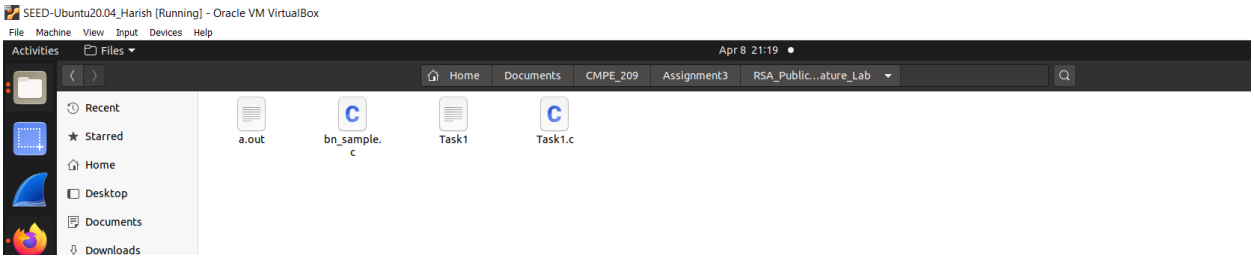


Fig. 43: Task1 file

h. Figure 44 shows the derived private key.

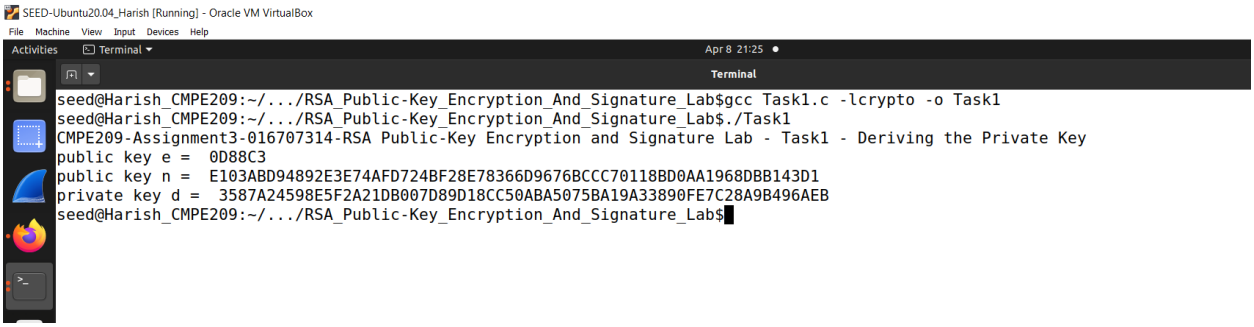


Fig. 44. Derived Private Key

2. Task2: Encrypting a Message:

a. Figure 45 shows the CCode for encrypting a message.

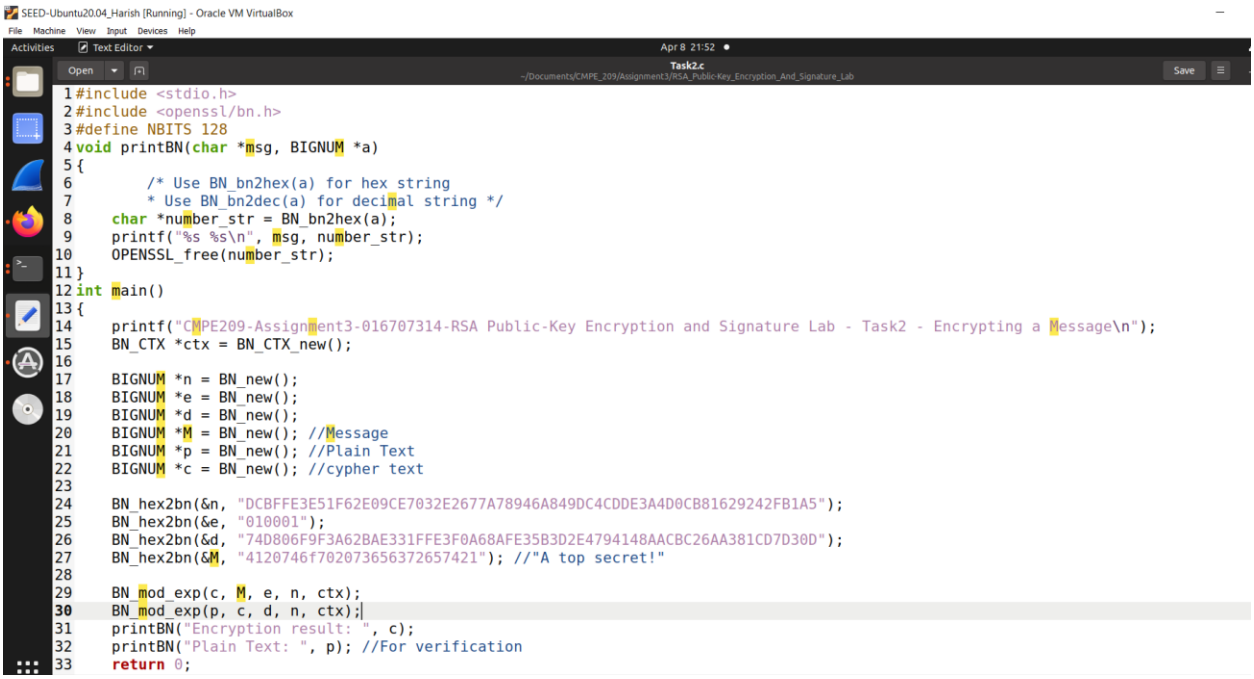


Fig. 45: Encrypting a message

b. Figure 46 shows the encrypted message.

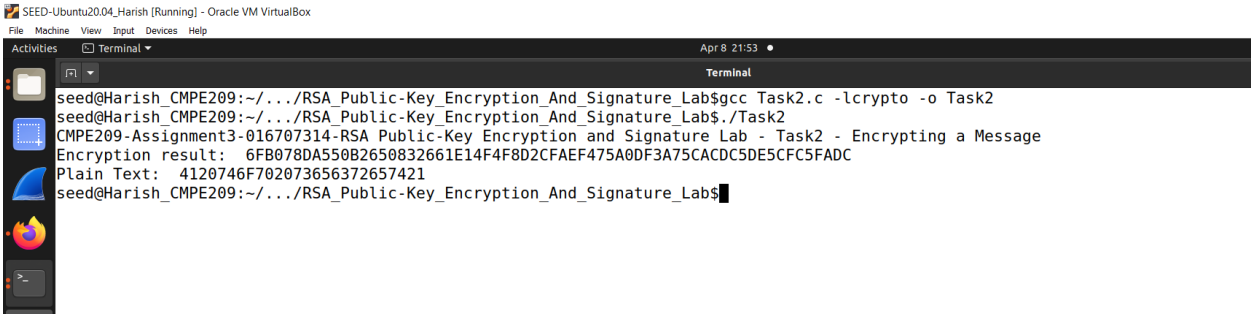


Fig. 46: Encrypted Message

3. Task 3 - Decrypted Message:

a. Figure 47 shows the CCode for decrypting a message.

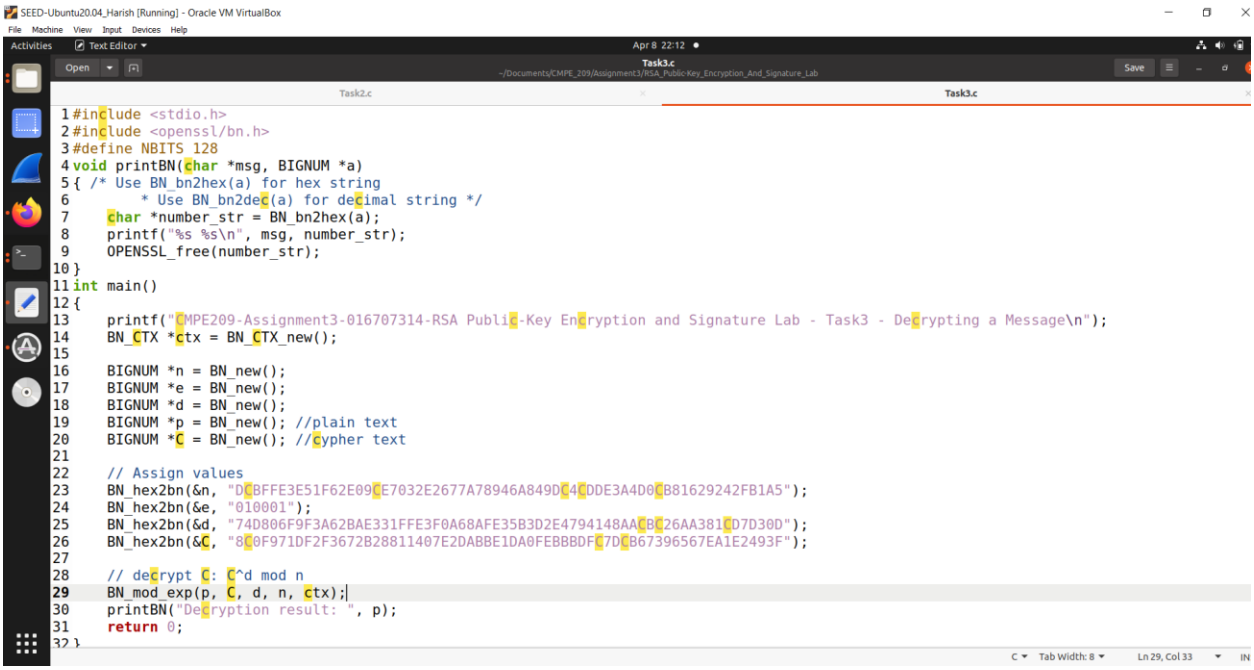


Fig 47: CCode

b. Figure 48 shows the decrypted message.

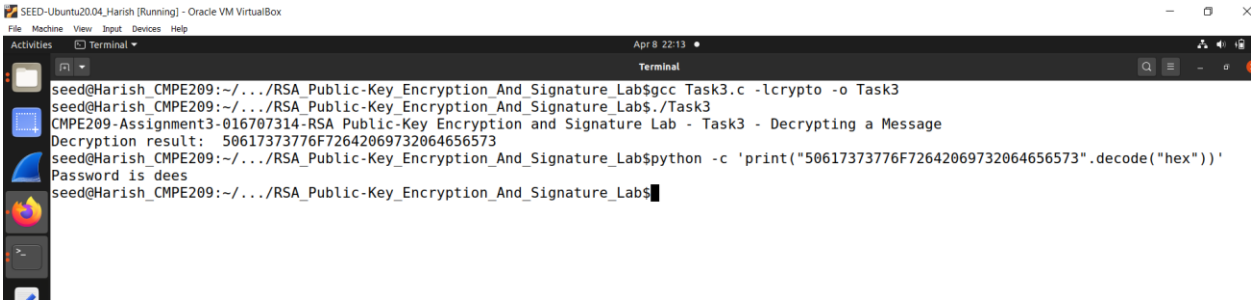


Fig. 48: Decrypted message

4. Task4 – Signing a Message:

a. Figure 49 shows the commands to get the hex strings.

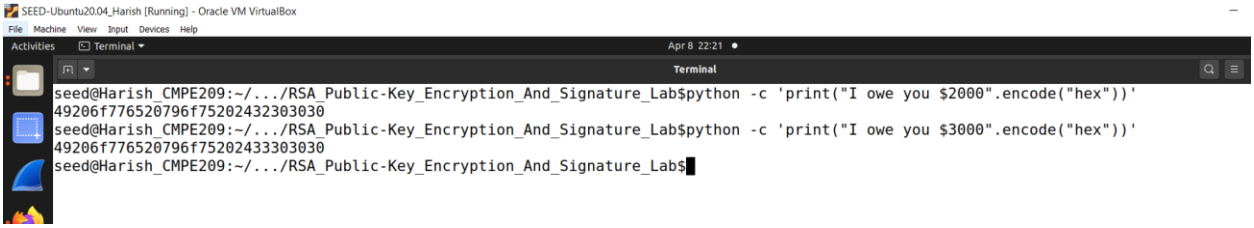


Fig. 49: Hex Strings

b. Figure 50 shows the CCode for signing a message.

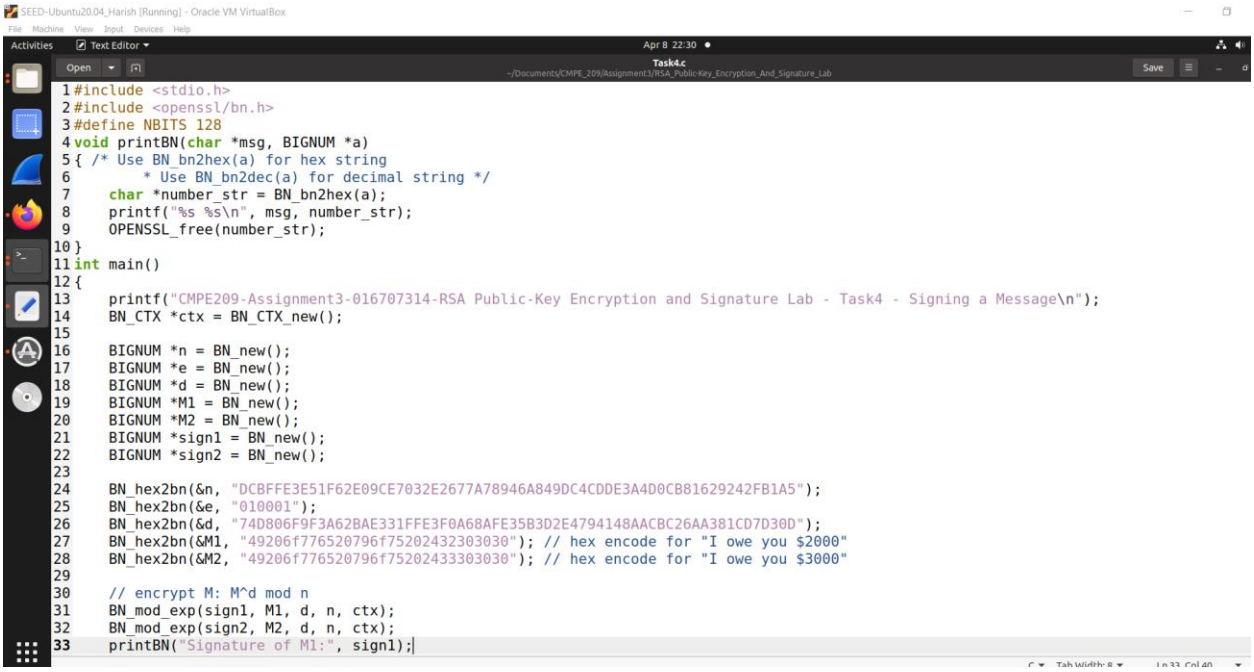


Fig. 50: CCode

c. Figure 51 shows the CCode continuation.

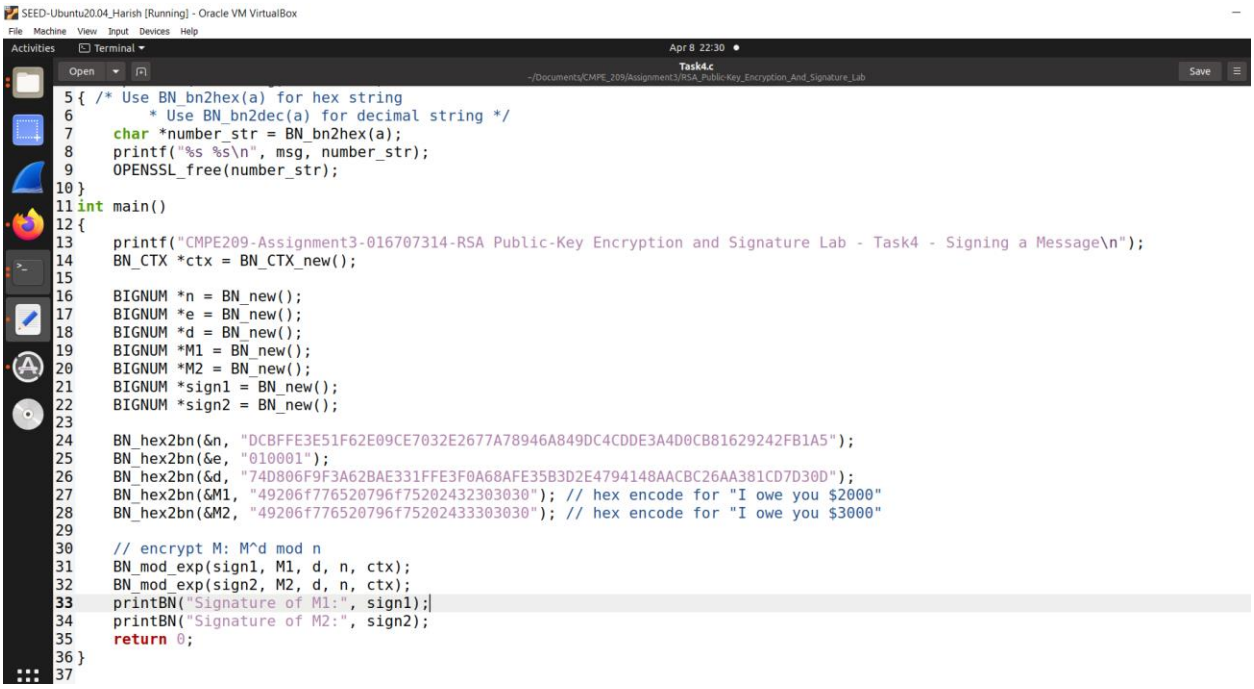


Fig. 51: CCode continuation

d. Figure 52 shows the signed message.

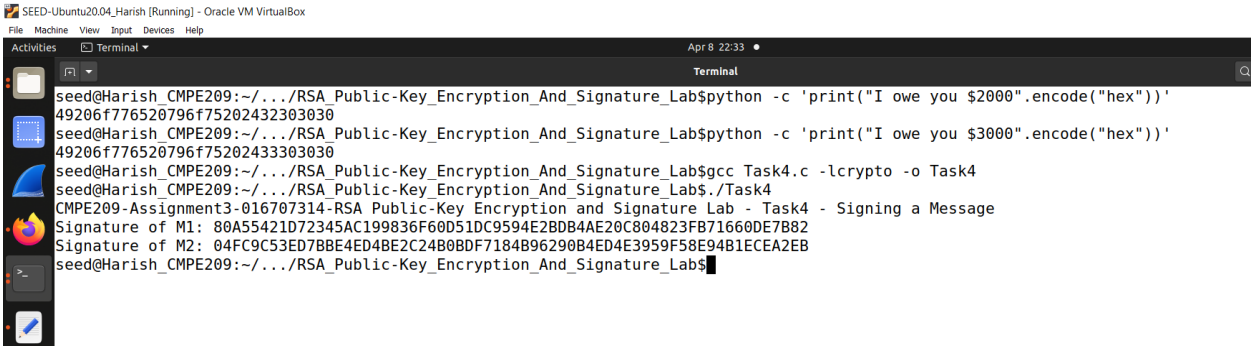


Fig. 52: Signed Message

Observation:

The task also said to describe the observations on if a slight change in the message would or would not alter the signature value. To achieve this, we had run the $2000 message and also the $3000 message and generated the hex string again to use it in the program. The result snippet is as follows where the observation is that the signatures are far different even for a slight change in the input message which is a good thing. It can be seen that although the information is only slightly changed, the signature result will also change greatly.

3. Pseudo Random Number Generation:

1. Task1 – Generate Encryption Key in a Wrong Way:
    a. To generate good pseudo-random numbers, we need to start with something random; otherwise, the results will be very predictable. Figure 53 is the CCode which uses the current time as the seed for a pseudo random generator.

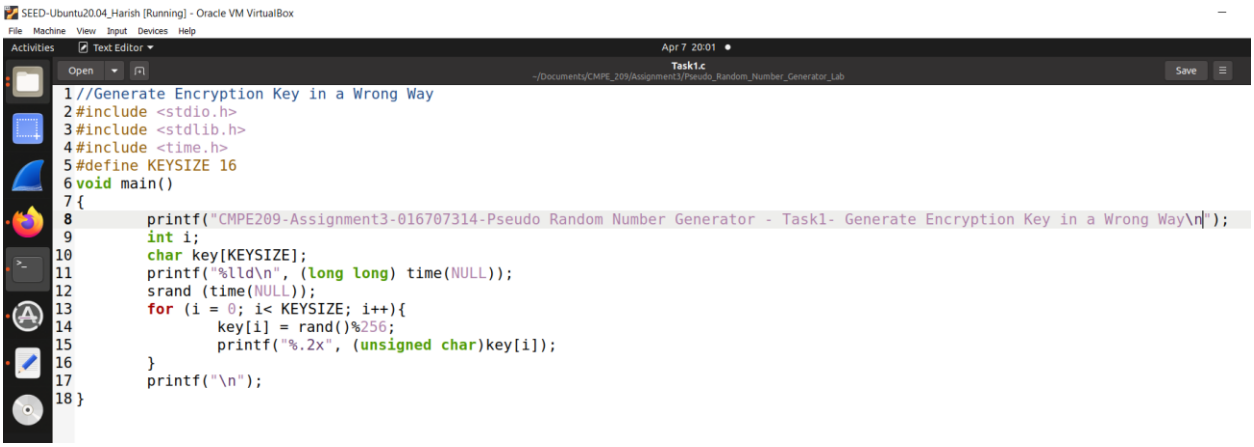


Fig. 53: CCode

b. Figure 54 shows the C file compilation.

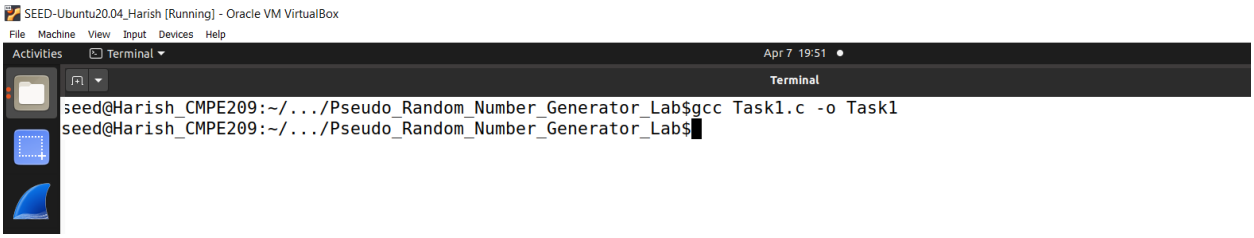


Fig. 54: Cfile compilation

c. Figure 55 shows the file created after compilation.

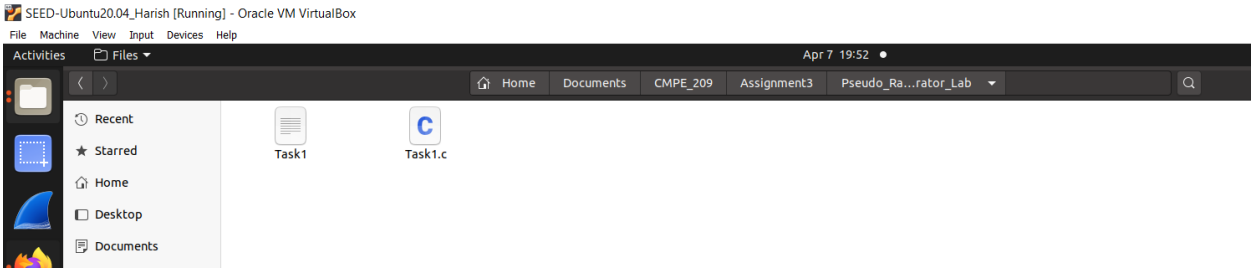


Fig. 55: Task1 file created

d. Figure 56 shows the random numbers generated.

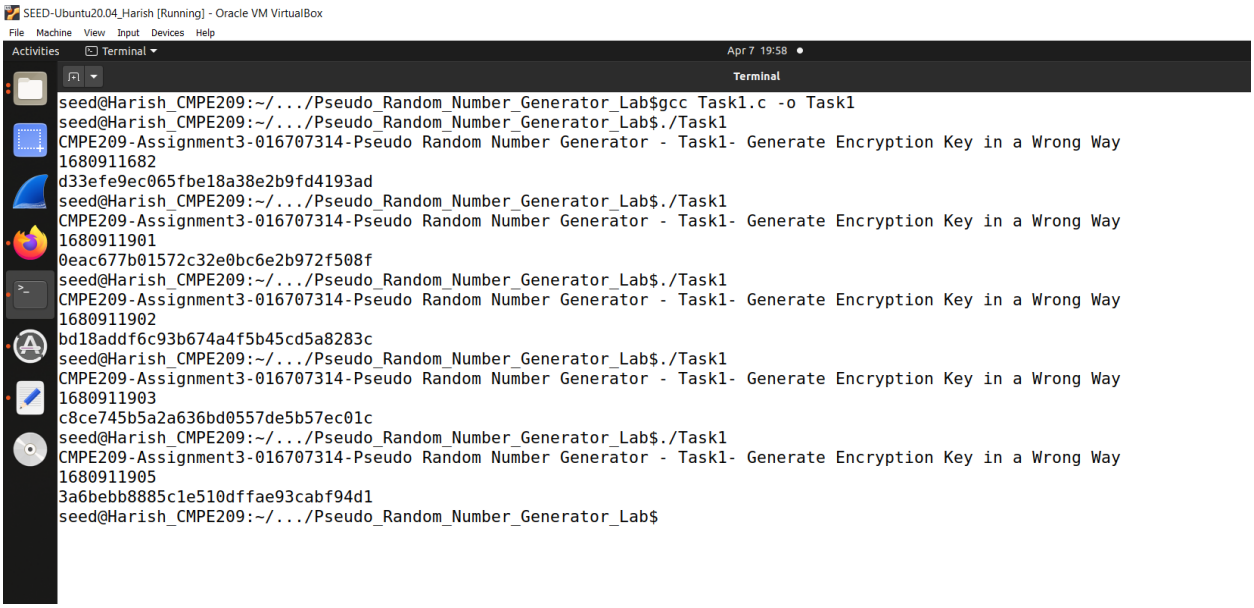


Fig. 56: Random numbers generation

e. Figure 57 shows the commented CCode.

Fig. 56: Commented srand line

f. Figure 57 shows the same random numbers because of commenting srand.

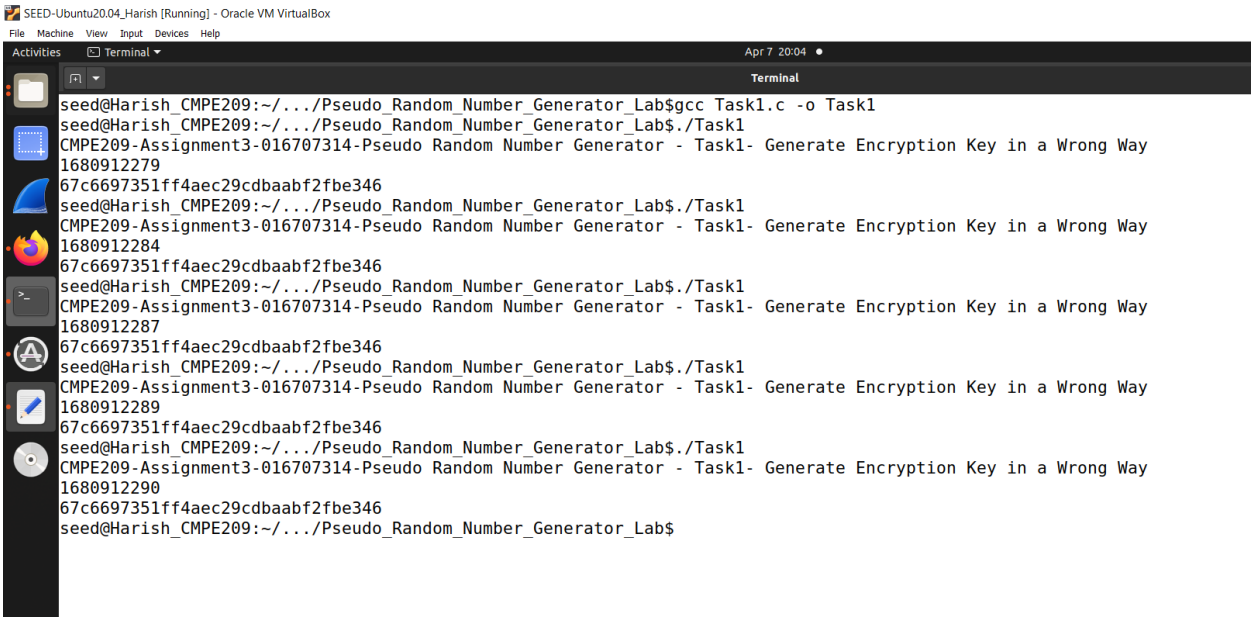


Fig. 57: Same random numbers

Observation:
It is found that the random number and the number of seconds are different each time. This is because the srand(time(NULL)) function is used to set the seed for the random number generation function rand(); time is a function of the C language to obtain the current system time, in seconds, representing the current time since the Unix standard time stamp (How many seconds elapsed at 0:00:00 on January 1, 1970, GMT); the time of each operation is different, so the random number obtained is also different. When commenting out srand (time(NULL)), since no seed is set, the default random number seed 0 will be used, so the random number generated is the same every time the program is run.

2. Task3 – Measure the Entropy of Kernel:

a. Figure 58 shows the Kernel Entropy at the current moment.

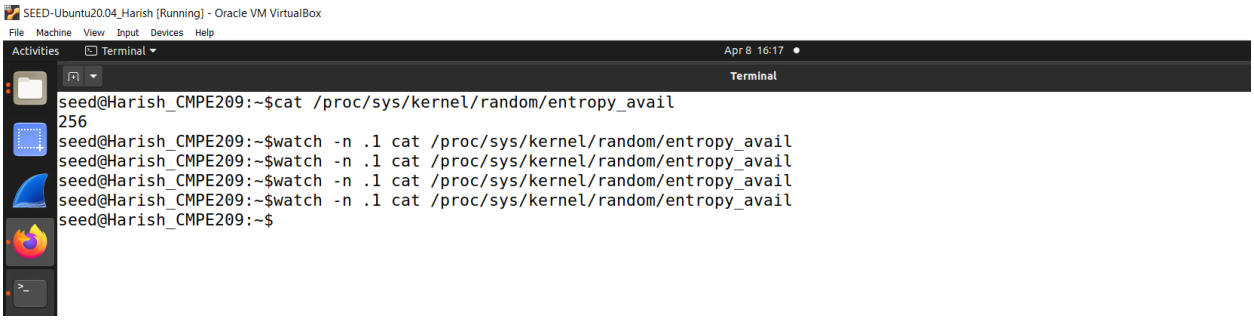


Fig. 58: Kernel Entropy Current Moment

b. Figure 59 shows the Entropy using watch.

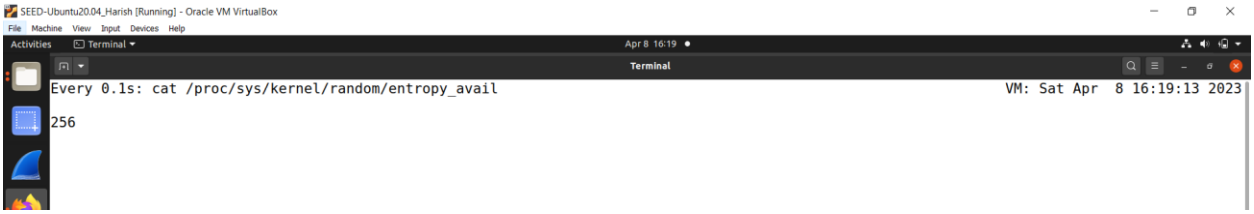


Fig. 59: Entropy using watch

Observations:

It is found that every time you move the mouse, hit the keyboard, etc., it will cause a change in entropy. When I move the mouse or type something, the value increases fast.

4. MD5 Collision Attack:

1. Task1: Generating Two Different Files with the same MD5 Hash:

a. Figure 60 shows the creation of prefix text.

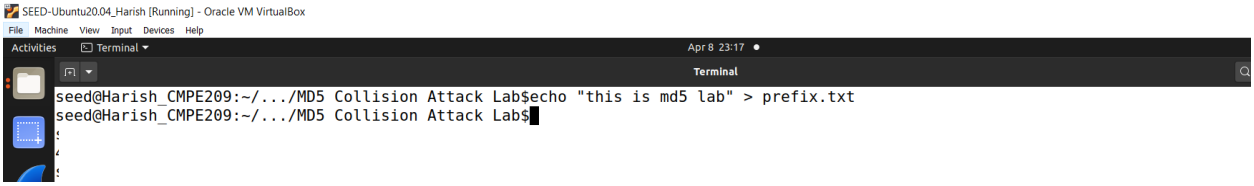


Fig. 60: Creation of prefix text

b. Figure 61 shows the created prefix text.

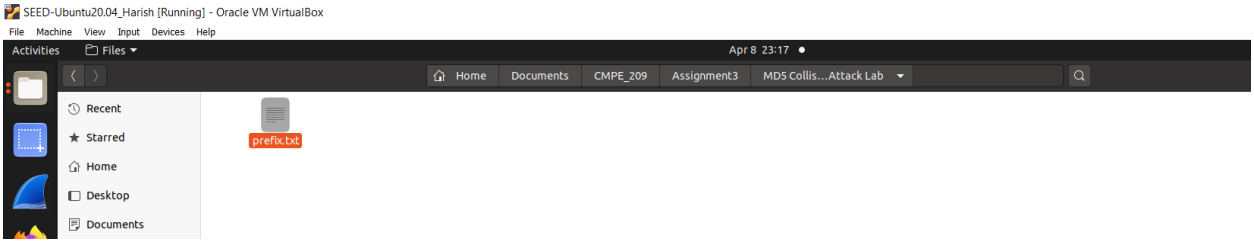


Fig. 61: Created Prefix Text

c. Figure 62 shows the generation of out1 bin and out2 bin.

Fig. 62: Generation of out1 bin and out2 bin

d. Figure 63 shows the generated out1 bin and out2 bin.

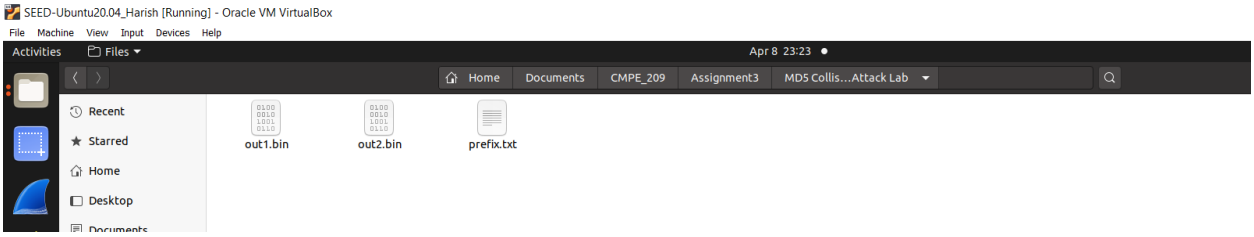


Fig. 63: Generated out1 bin and out2 bin

e. Figure 64 shows the out1bin text.

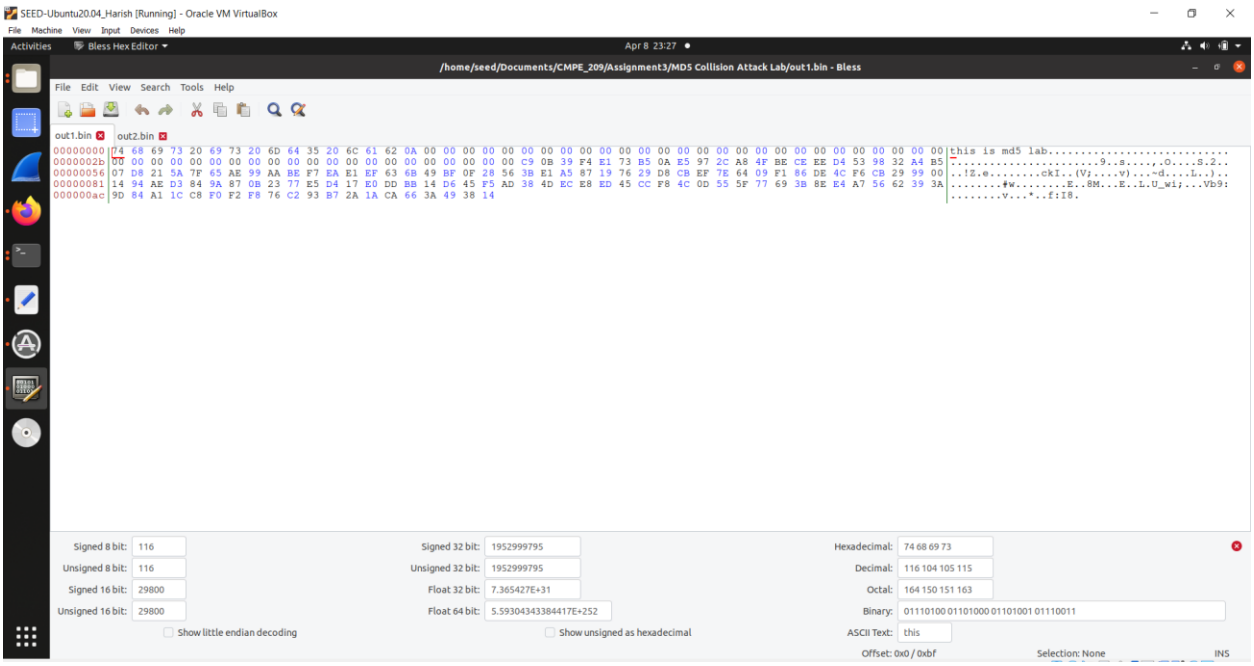


Fig. 64: out1 bin text

f. Figure 65 shows the out2 bin text.

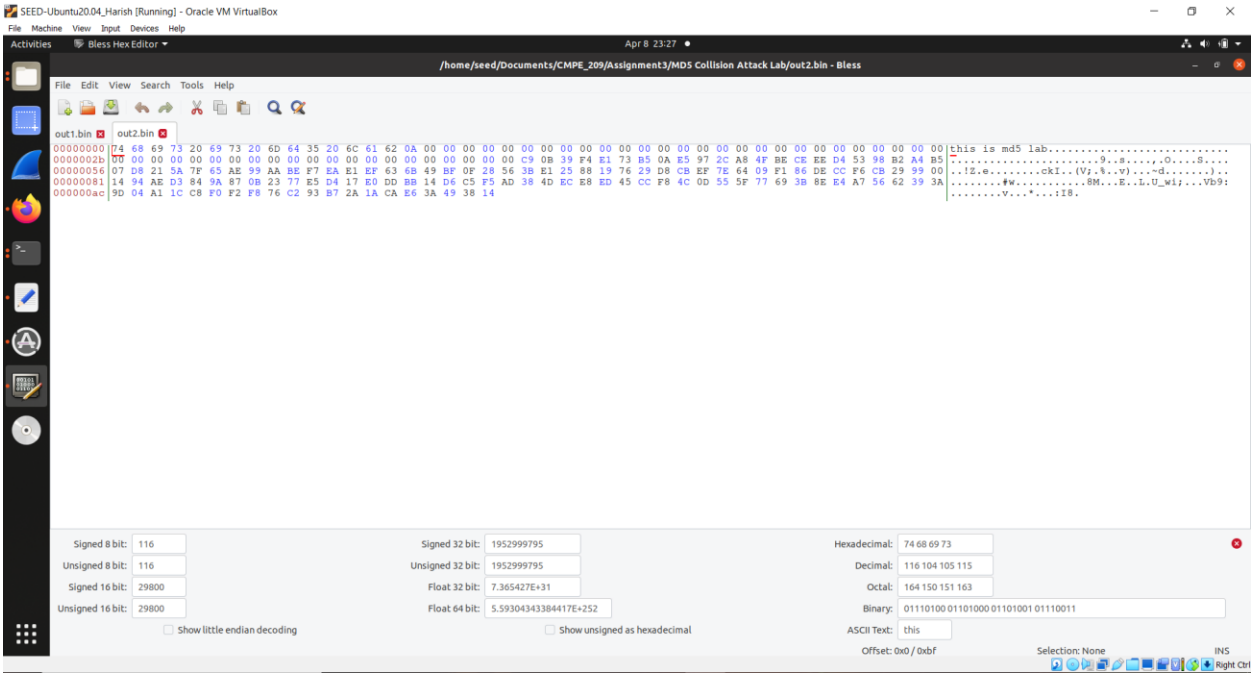


Fig. 65: out2 bin text

g. Figure 66 shows the difference of 2 files. It shows that the files are different.

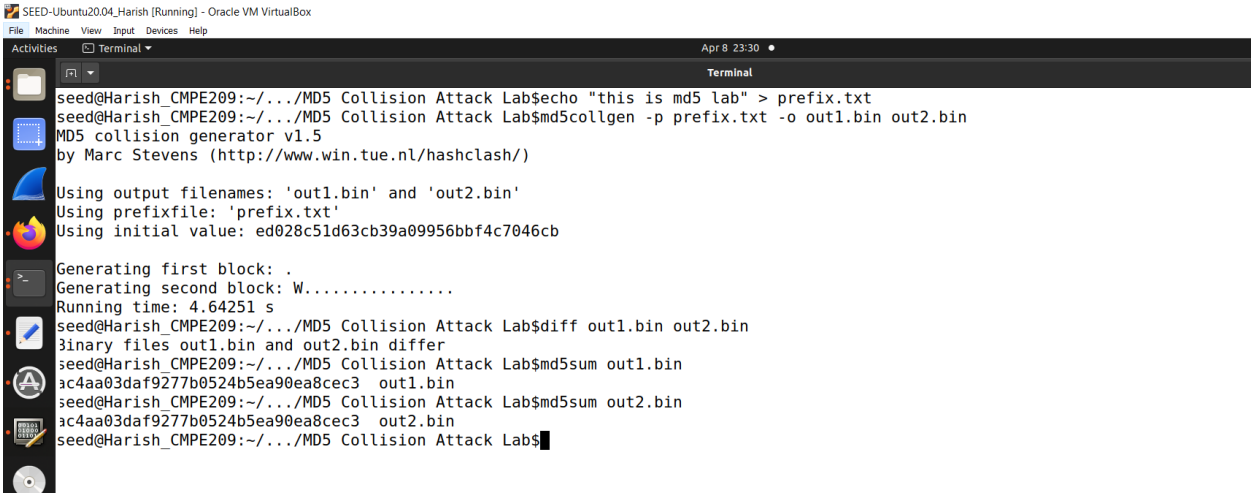


Fig. 66: Difference of 2 files

Answer 1: If the length of the prefix file is not a multiple of 64 and let's say it is less than 64, then the prefix length is made 64 bytes by padding the required number of 0's. If the length is more than 64, then the prefix is divided into 64-byte blocks and then hashed accordingly.

Answer 2:

a. Figure 67 shows the prefix text creation.

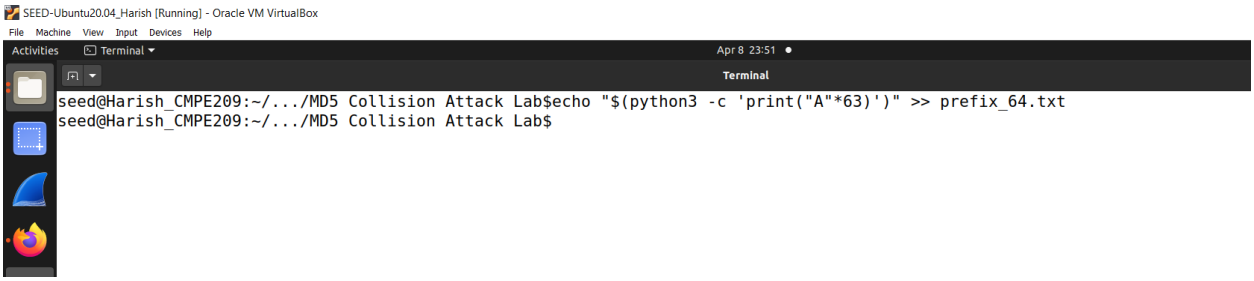


Fig. 67: Prefix Text Creation

b. Figure 68 shows the created prefix.

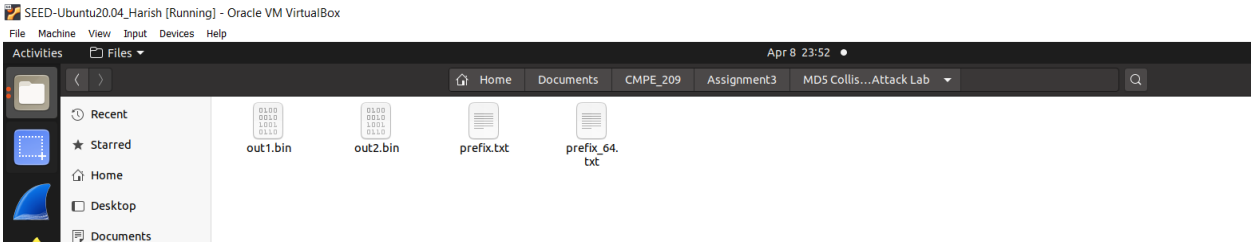


Fig. 68: Created Prefix

c. Figure 69 shows the prefix text content.

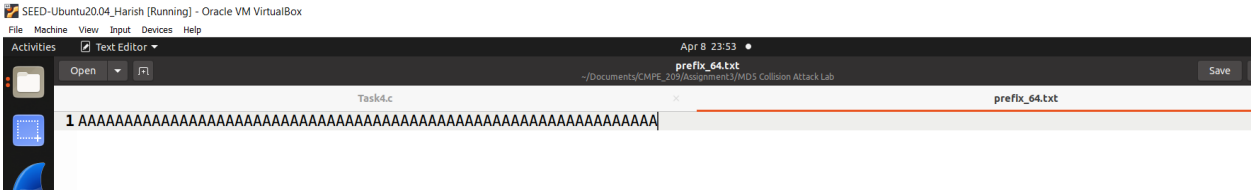


Fig. 69: Prefix Text content

d. Figure 70 shows the generation of pf1 and pf2 bin files.

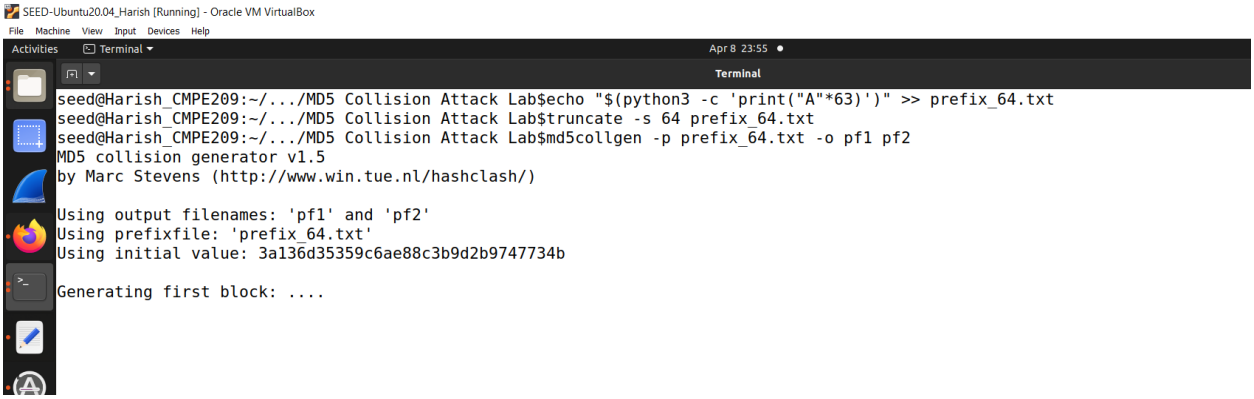


Fig. 70: Creation of pf1 and pf2 bin file

e. Figure 71 shows the generated pf1 and pf2 file.

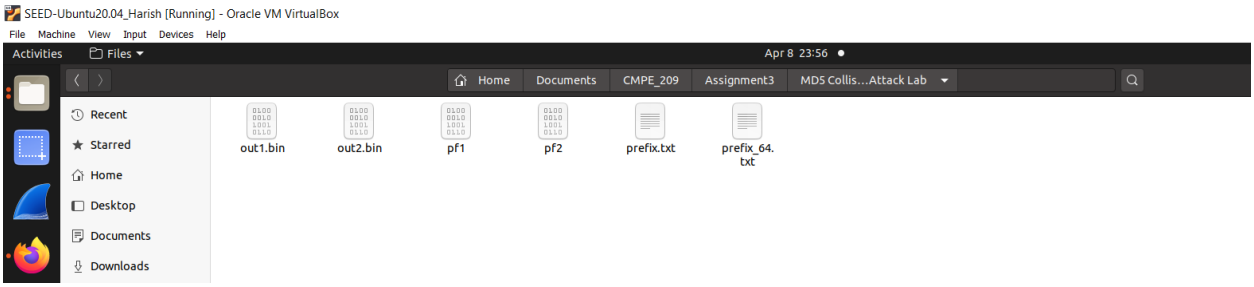


Fig. 71: Generated pf1 and pf2

f. Figure 72 shows pf1 text.

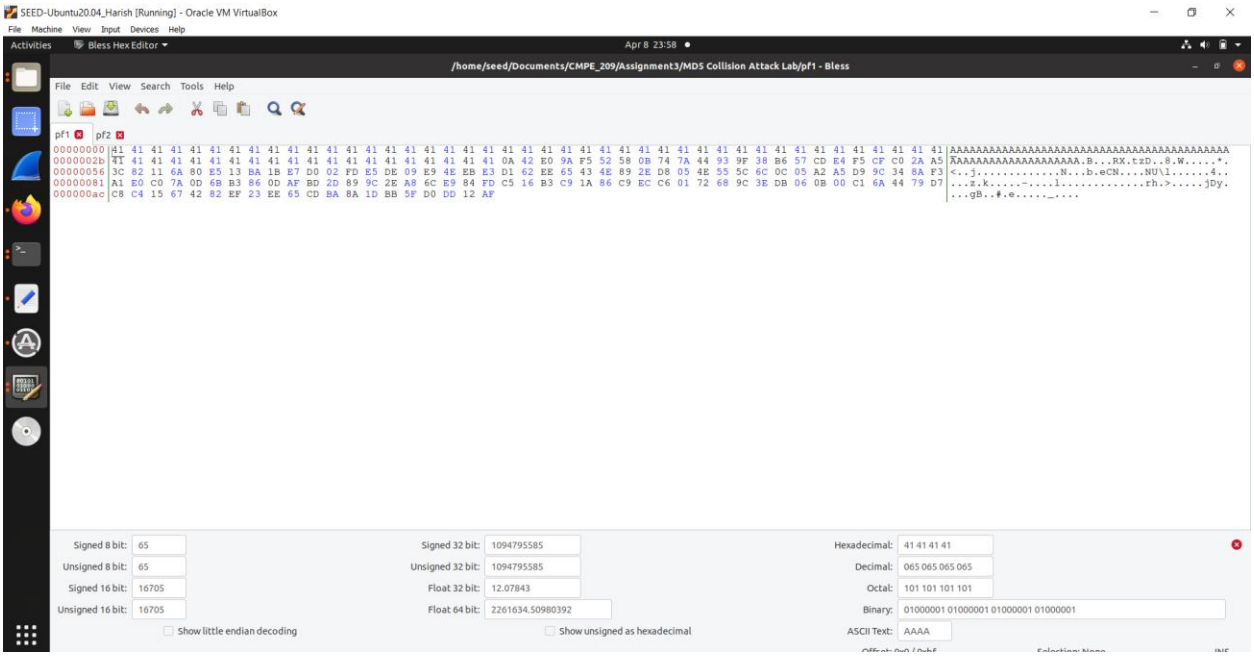


Fig. 71: pf1 text

g. Figure 73 shows the pf2 text.

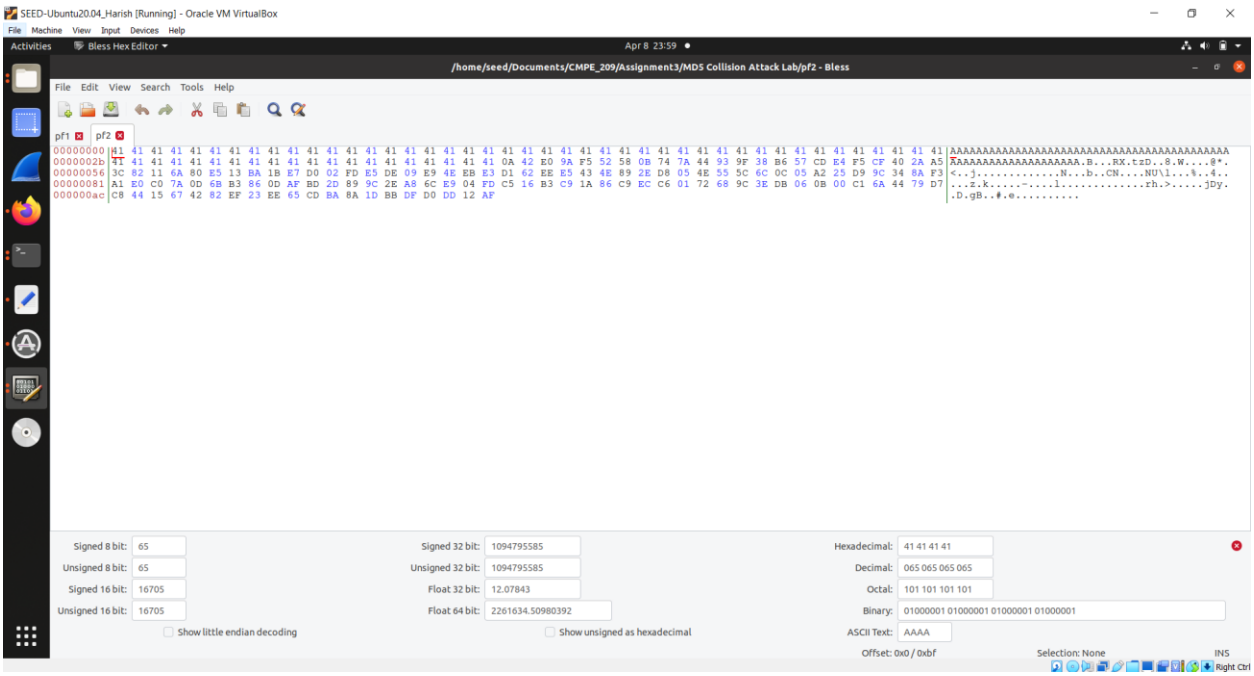


Fig. 73: pf2 text

h. Figure 74 shows the pf1 and pf2 differencces.

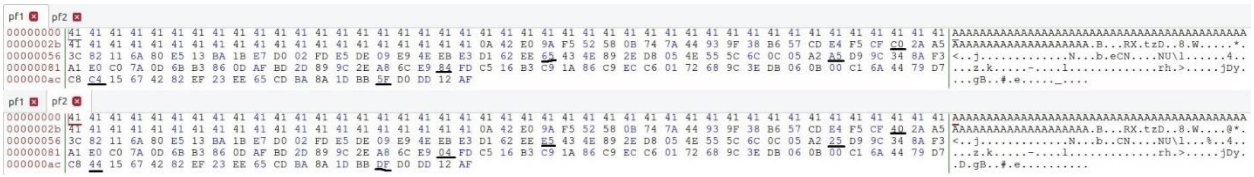


Fig. 74: Differences in both files

i. Figure 75 shows the python code for checking differences.

Fig. 75: Python for checking differences

j. Figure 76 shows the differences in bytes.

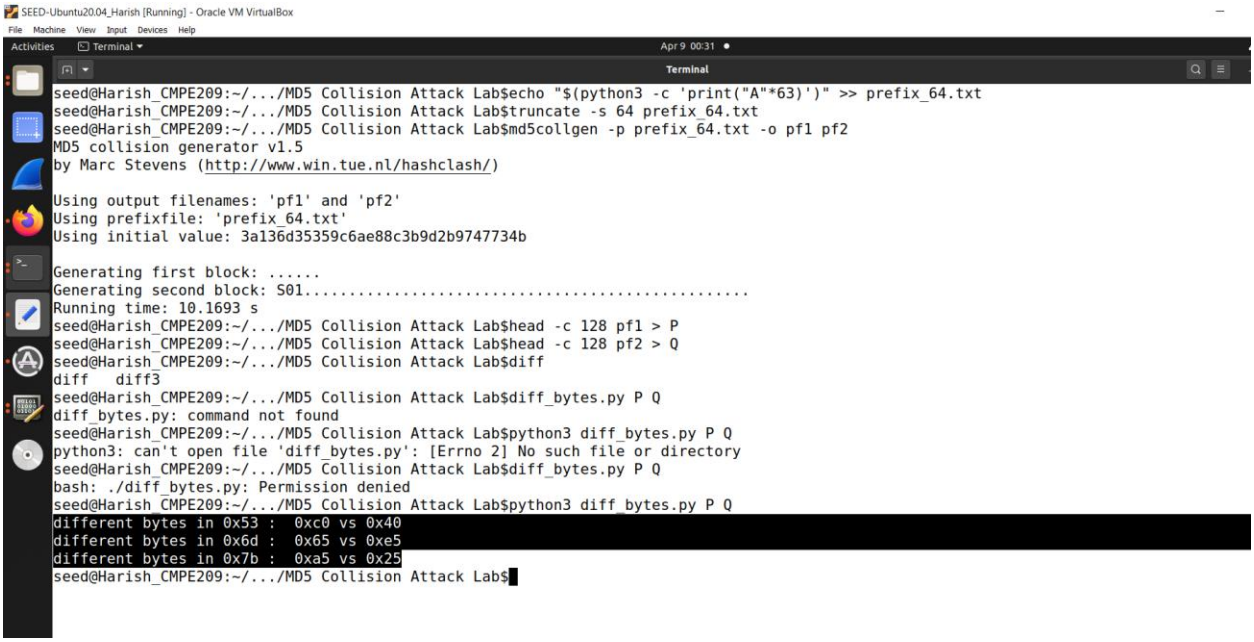


Fig. 76: Differences in bytes

Answer 3: No, not all bytes are different. In the previous case, the bytes differed only in individual positions. After many trials, the places where these differences were found were not fixed. Figure 77 shows the differences.

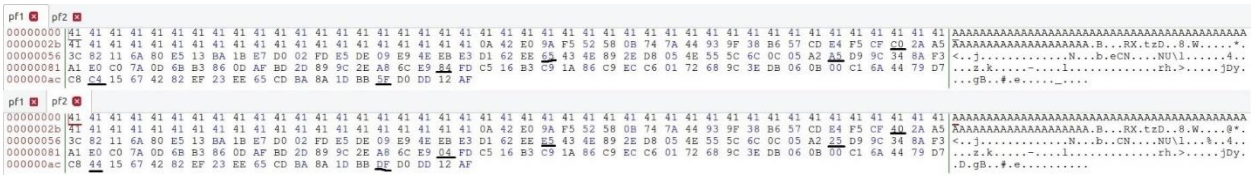


Fig. 77: A few differences

8. SDN Mininet Lab:

a. Installing the dependencies. Figure 78 shows apt -get update command.

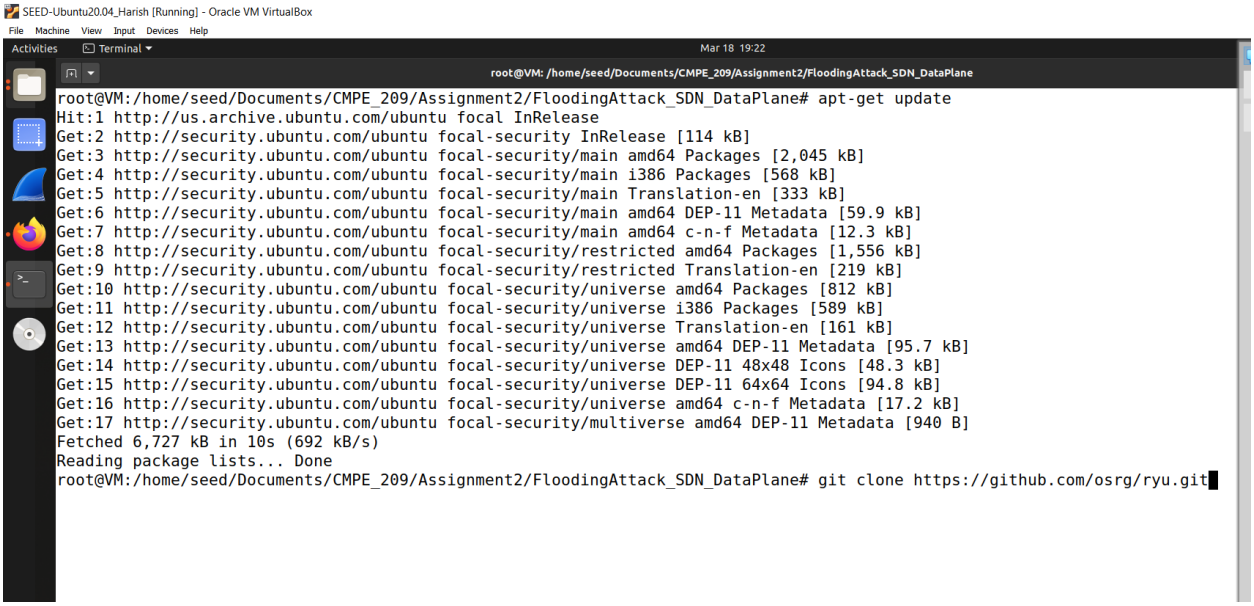


Fig. 78: apt -get update command

b. Figure 79 shows the apt install command.

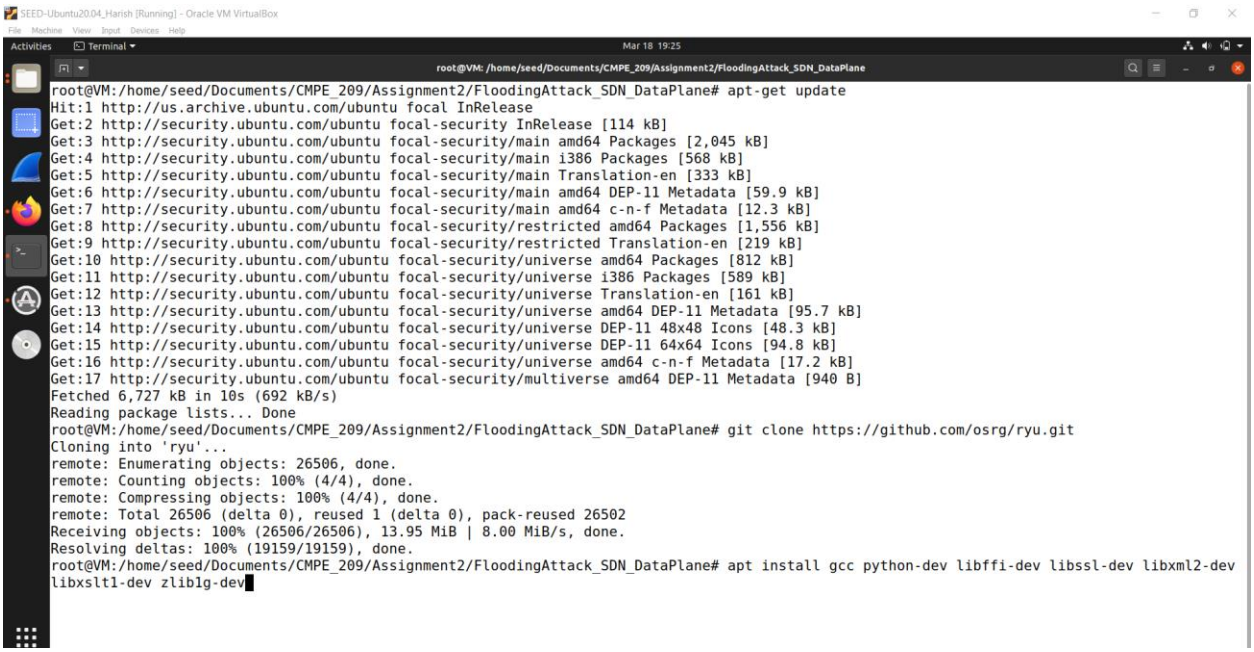


Fig. 79: apt install command

c. Figure 80 shows the output for the previous command and apt install python3 pip command.

Fig. 80: Output for previous command

d. Figure 81 shows the output for previous command and cd ryu command.

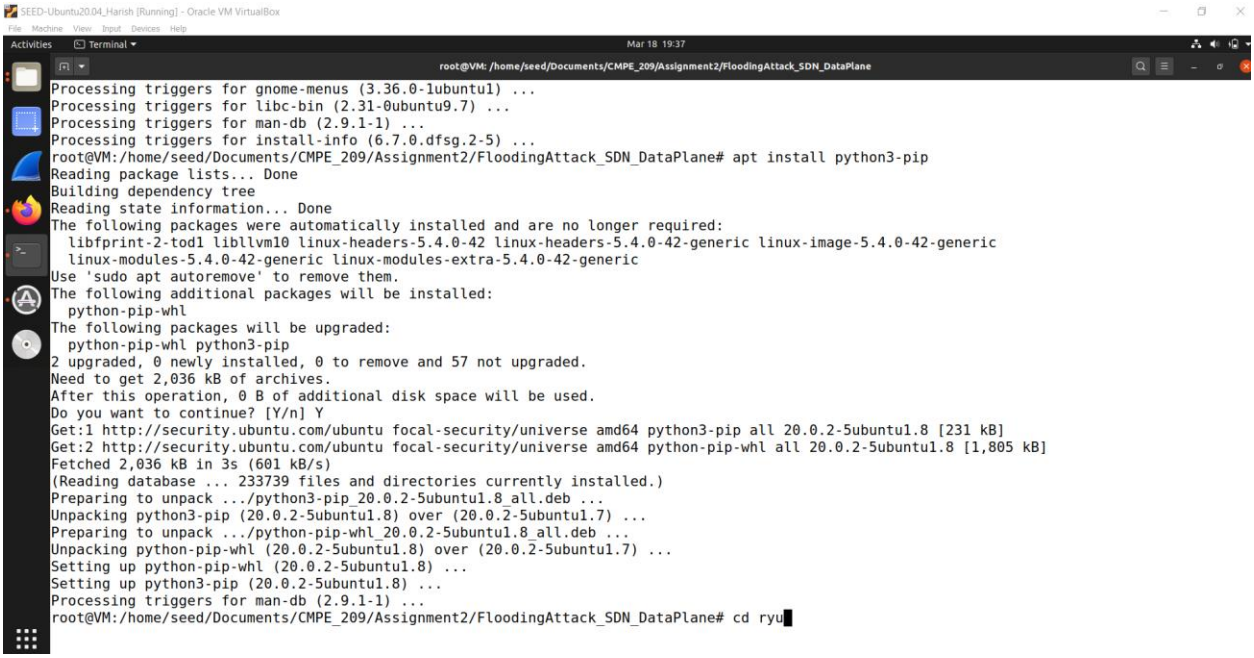


Fig. 81: Output for previous command and cd ryu command

e. Figure 82 shows the output for previous command and pip3 install command.

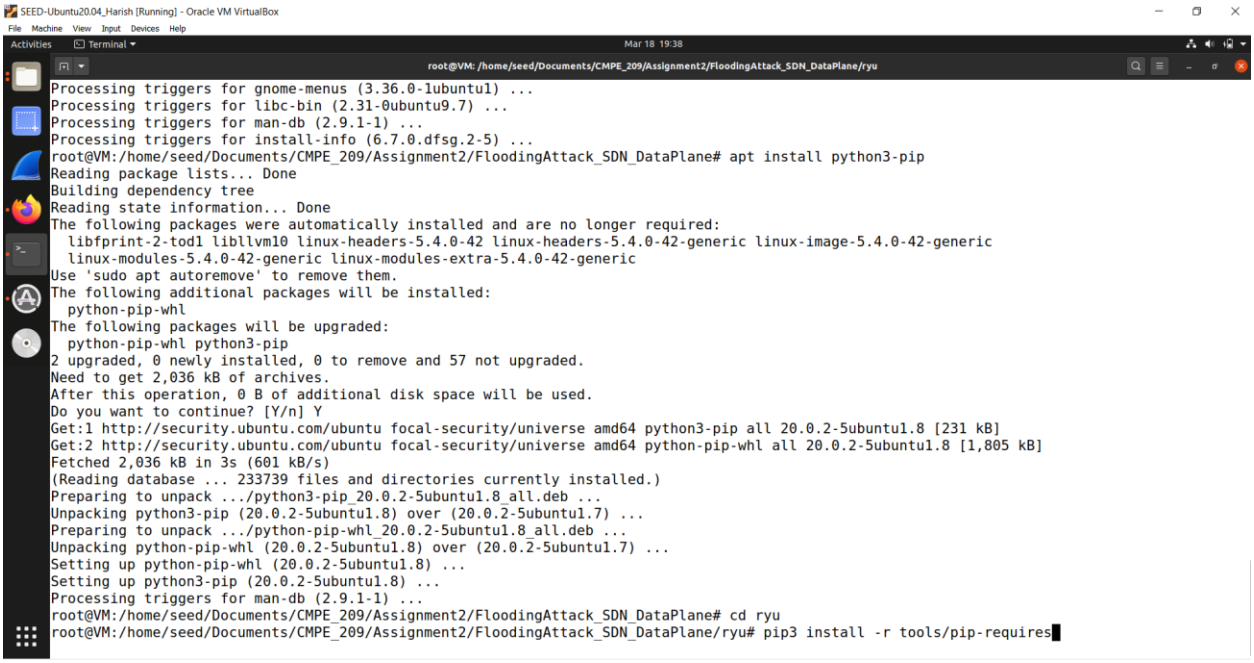


Fig. 82: Output for previous command and pip3 install command

f. Figure 83 shows the output for previous command and python setup install command.

Fig. 83: Output for previous command and python setup install command

g. Figure 84 shows the output for previous command.

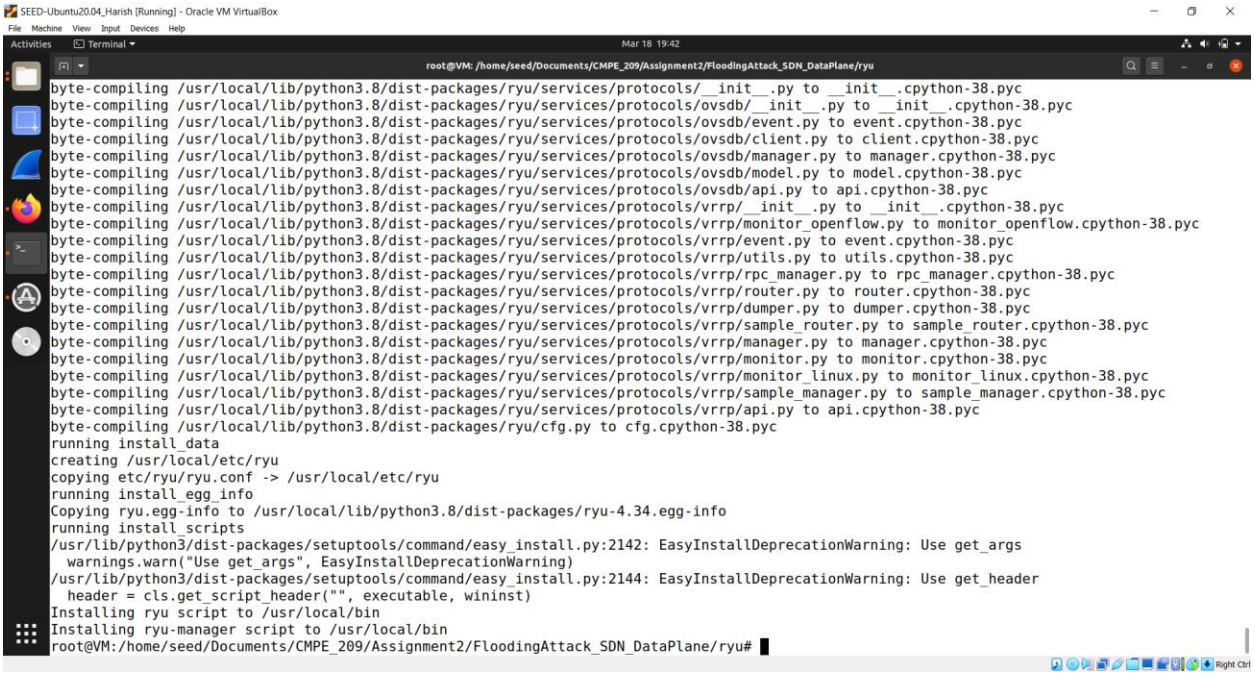


Fig. 84: Output for previous command

h. Figure 85 shows the install mininet command.

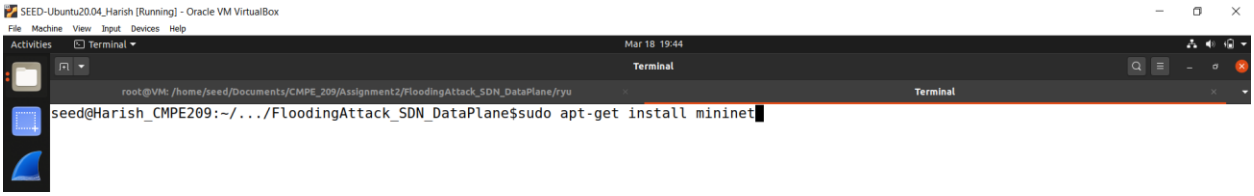


Fig. 85: Install Mininet command

i. Figure 86 shows the output of previous command sudo apt -get install hping3 command.

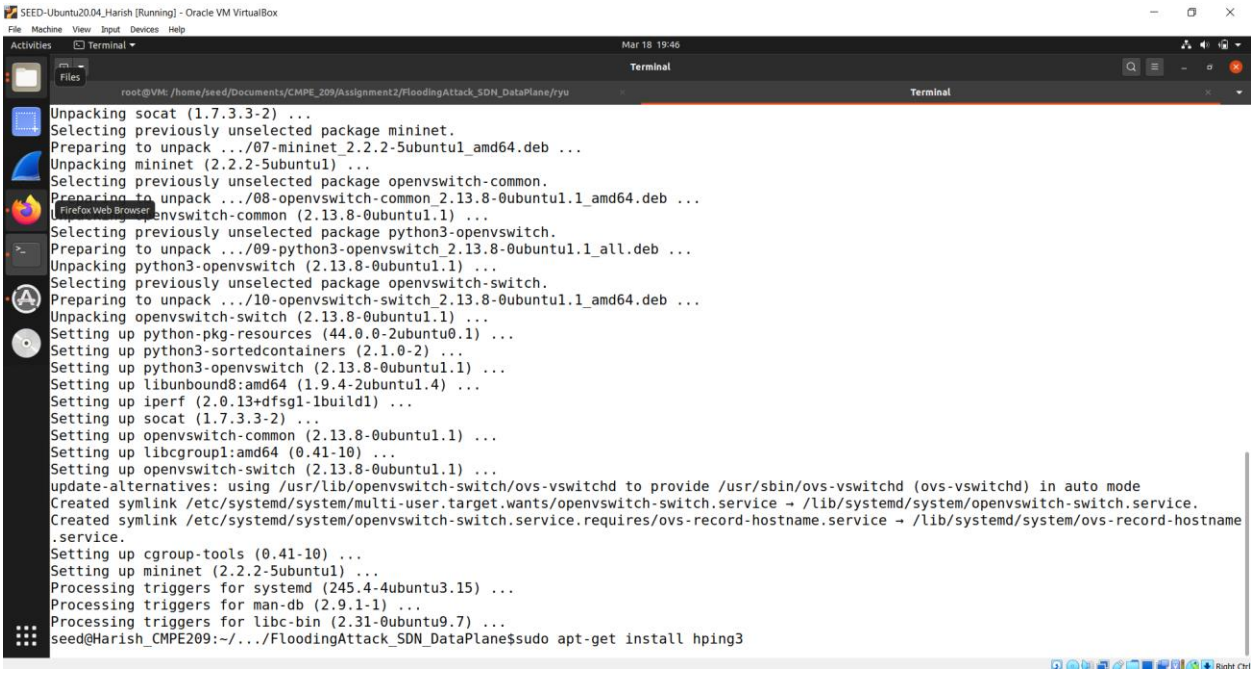


Fig. 86: sudo apt-get install hping3

j. Figure 87 shows the output for the previous command.

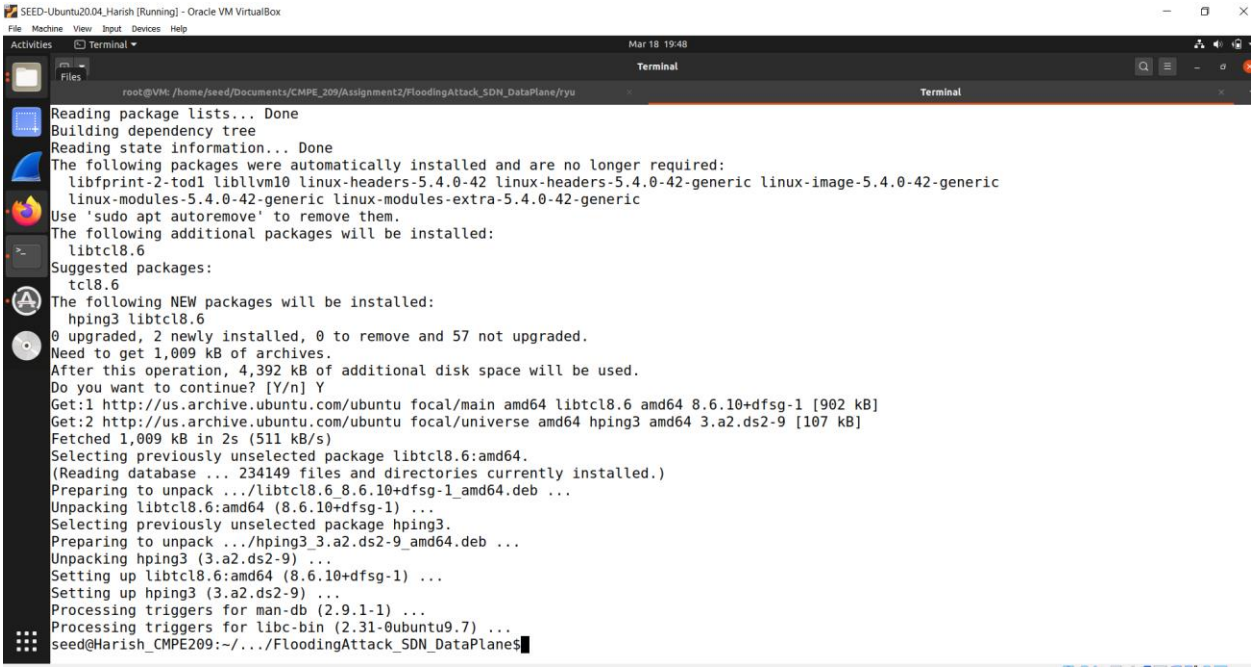


Fig. 87: Output for previous command

k. Figure 88 shows the command for running ryu.

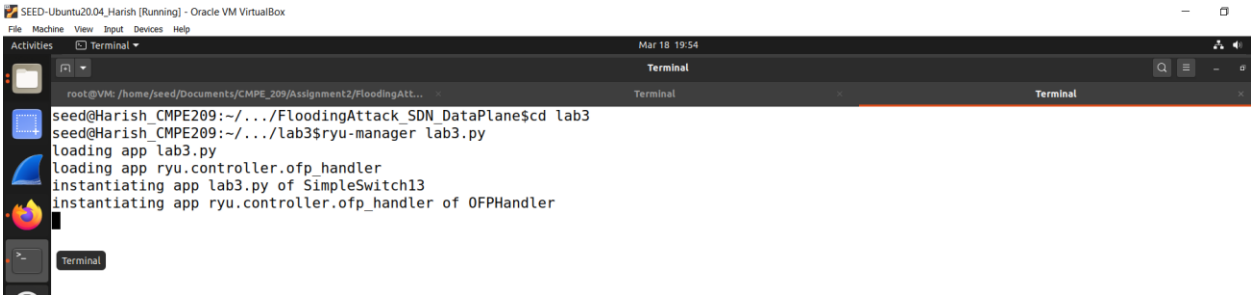


Fig. 88: Running ryu

l. Figure 89 shows the command for running mininet topology.

Fig. 89: Running mininet topology

m. Figure 90 shows the command for run ping all.

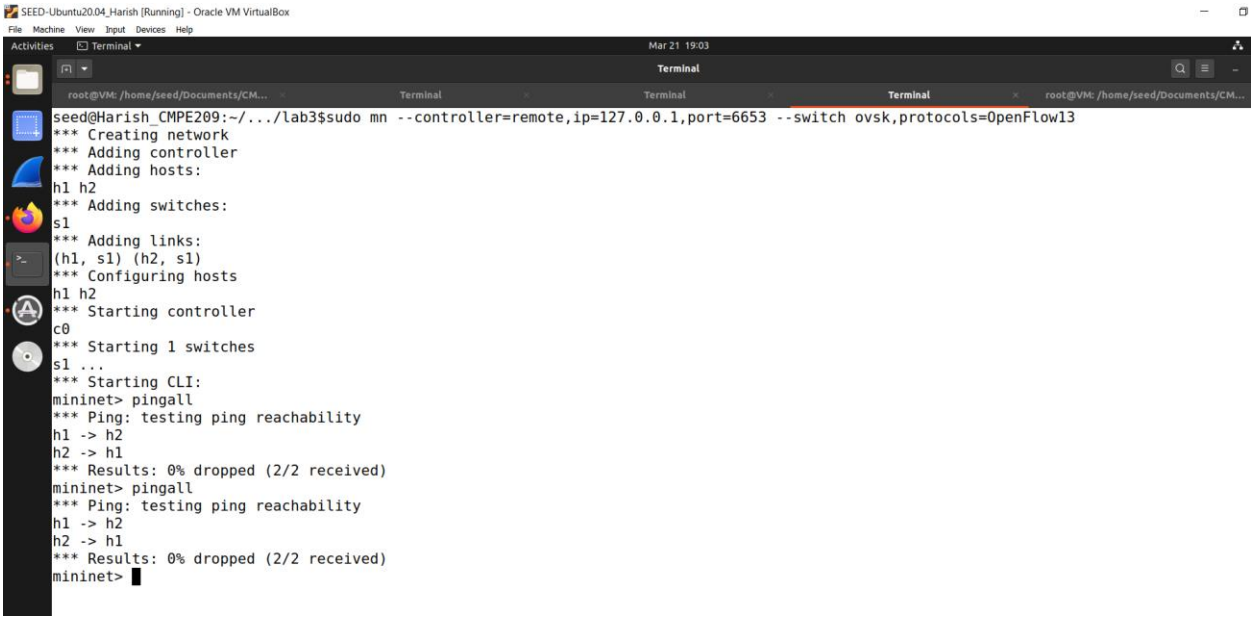


Fig. 90: Run Ping All command

1. Task1:

a. Figure 91 shows the current flow rules.

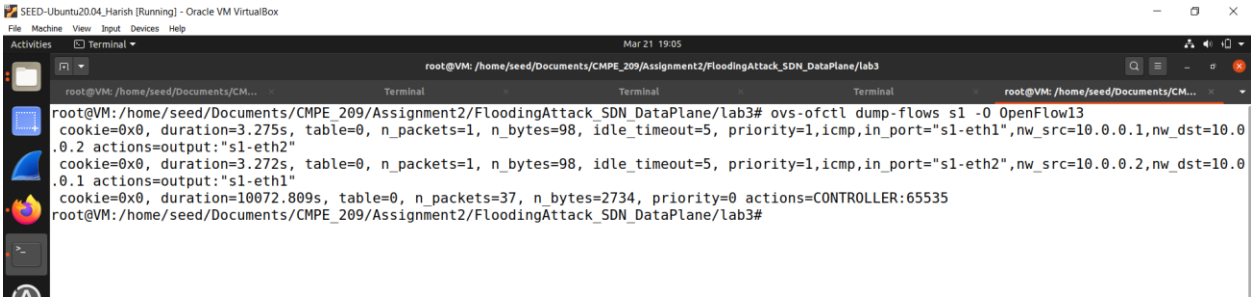


Fig. 91: Current Flow Rules

2. Task2:

a. Figure 92 shows flood packets to h2.

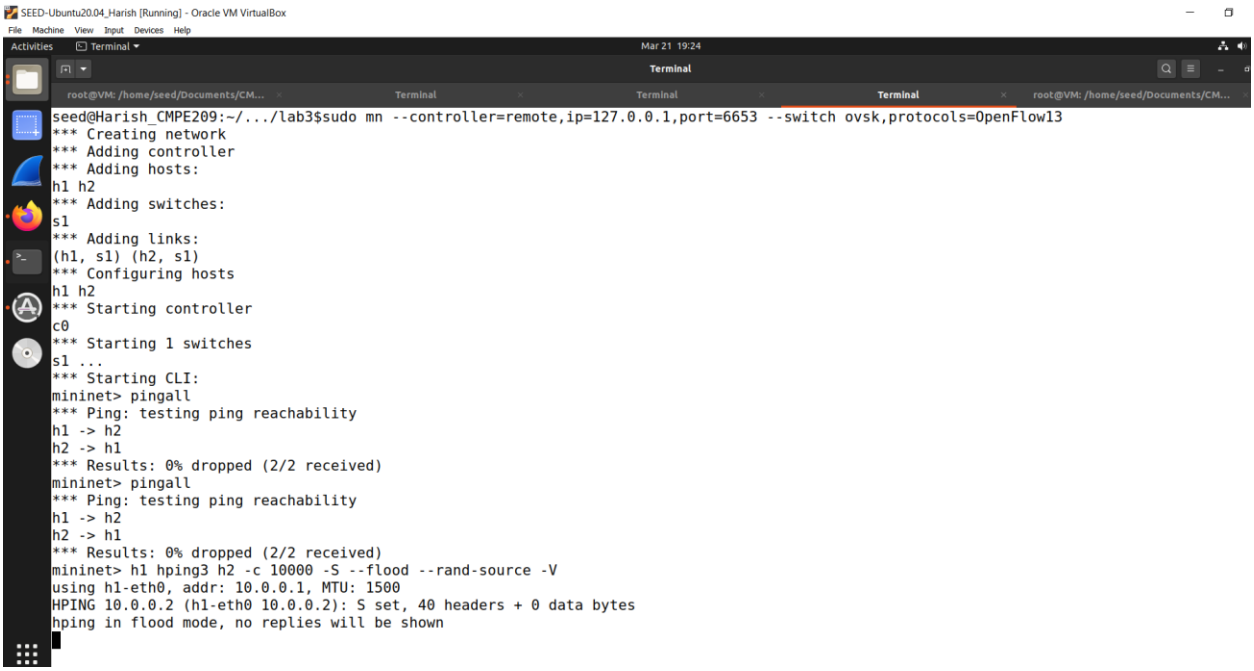


Fig. 92: Flood packets to h2

3. Task3:

a. Figure 93 shows the flow entries in S1.

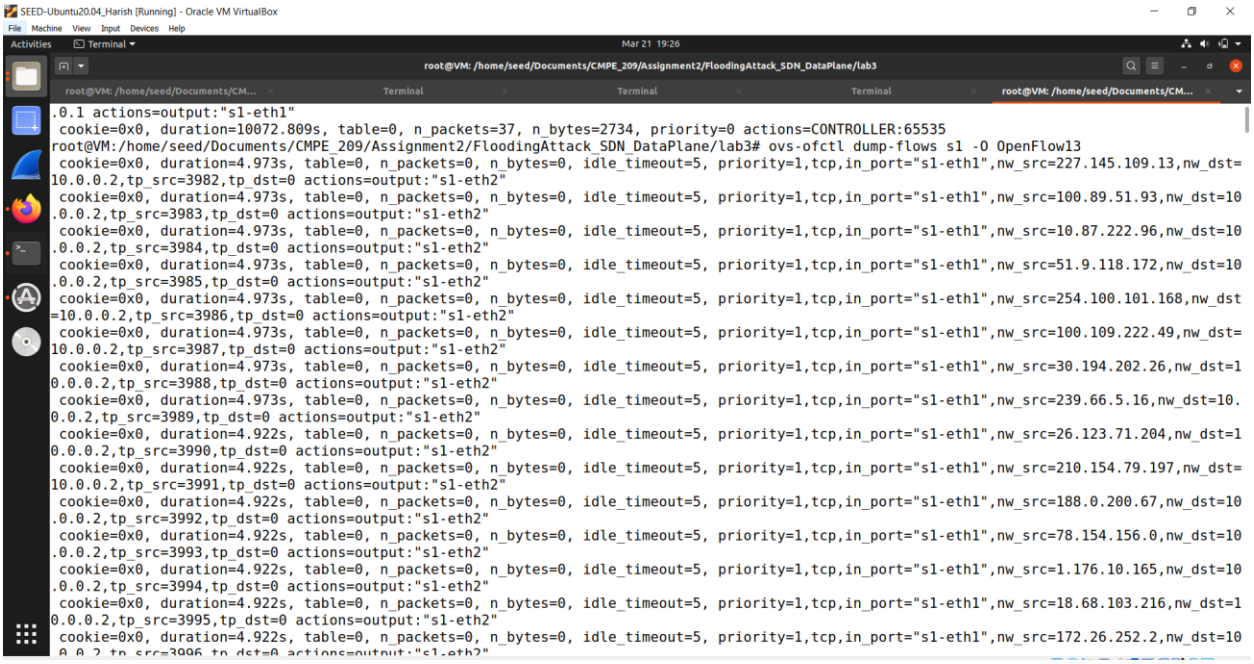


Fig. 93: Flow Entries in S1

b. Figure 94 shows the continued flow entries in S1.

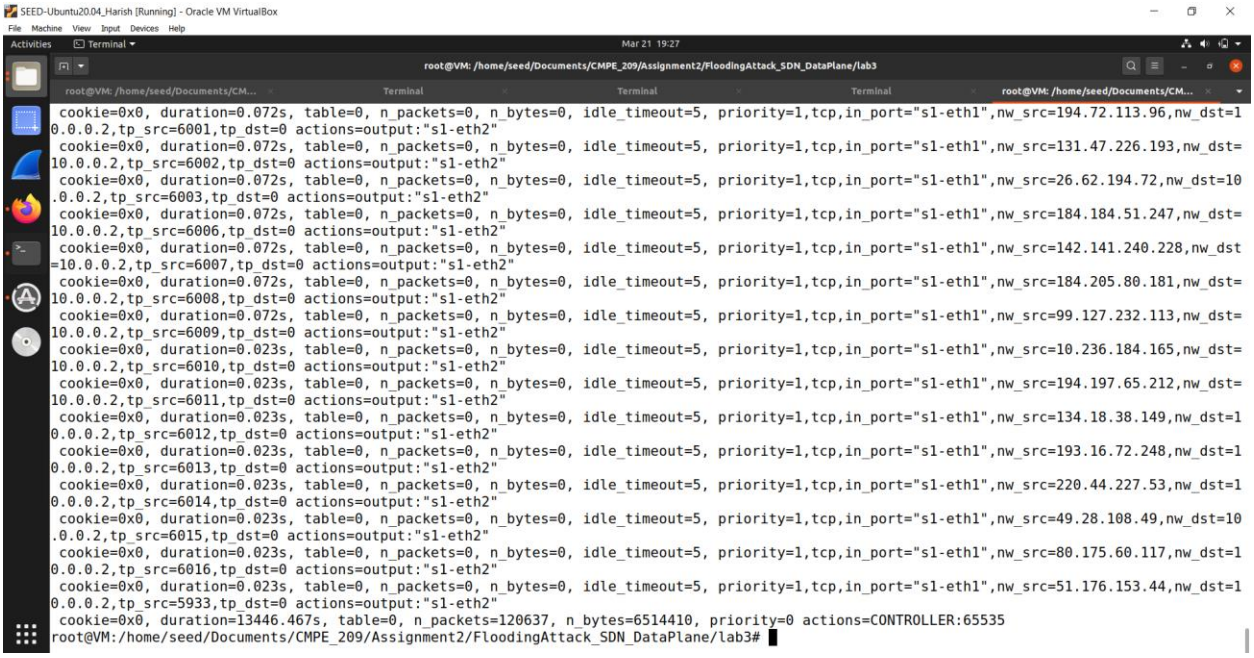


Fig. 94: Continued flow entries in S1

c. Figure 95 shows the destination host unreachable messages.

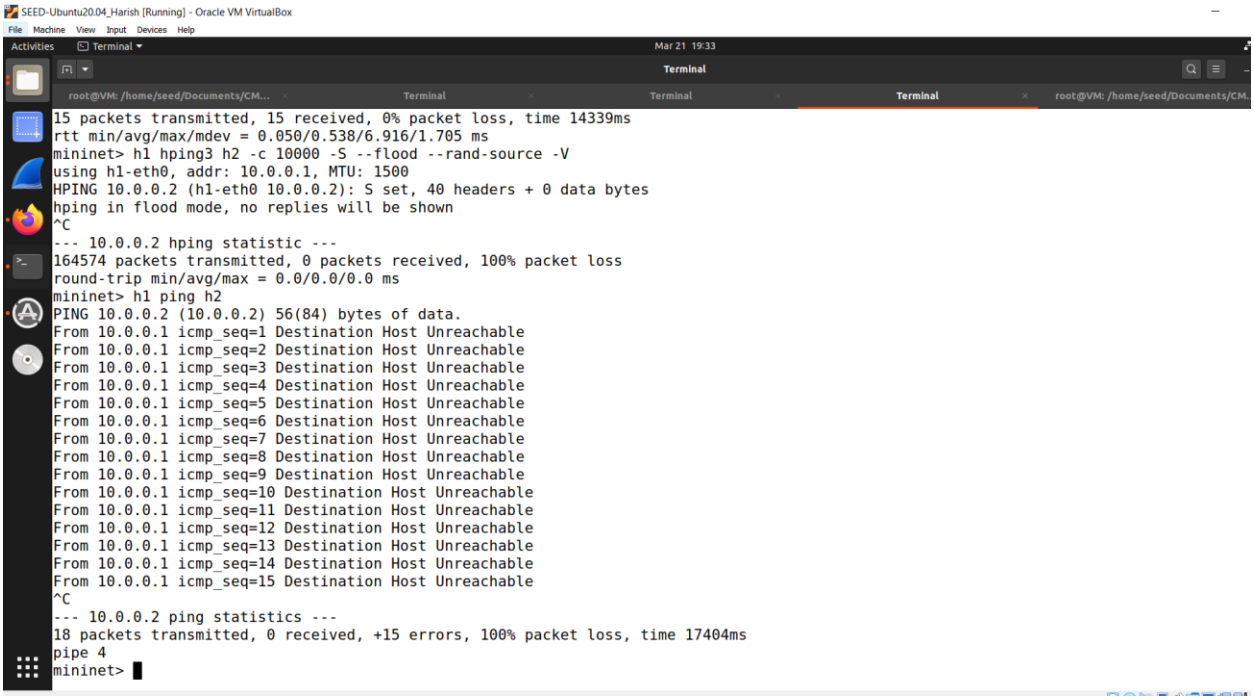


Fig. 95: Destination Host Unreachable messages

d. Figure 96 shows the packets sent properly.

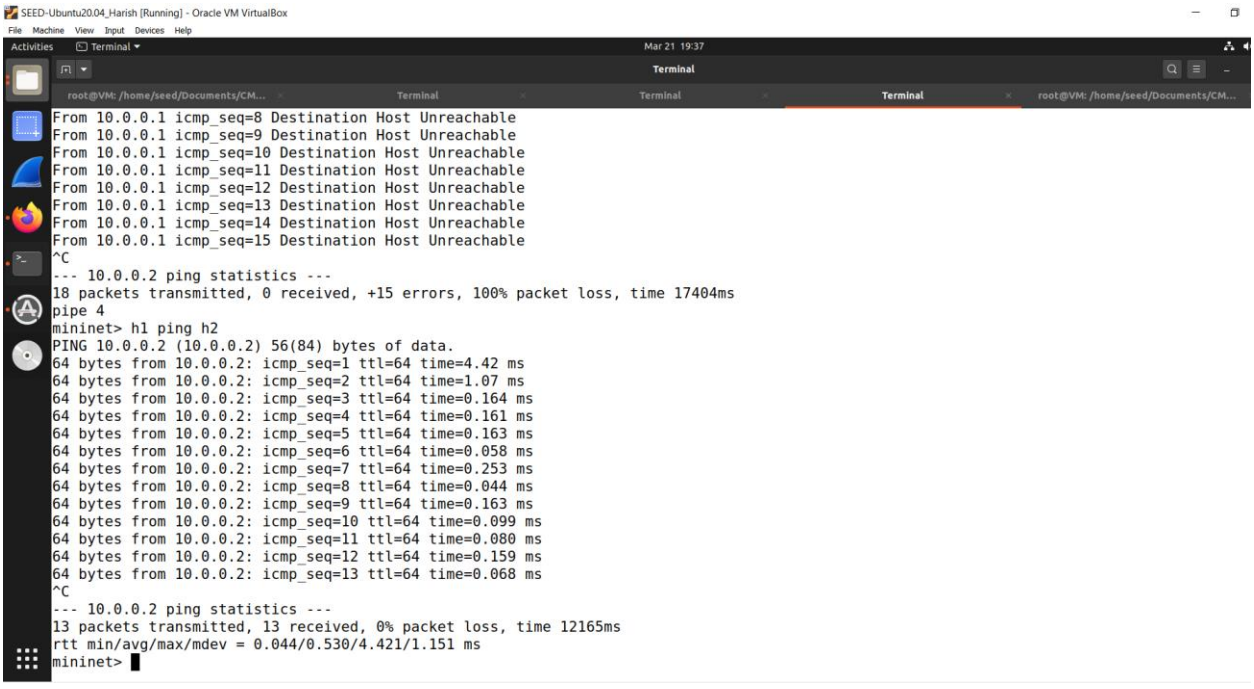


Fig. 96: Packets sent properly

e. Figure 97 shows the flow table rules of S1.

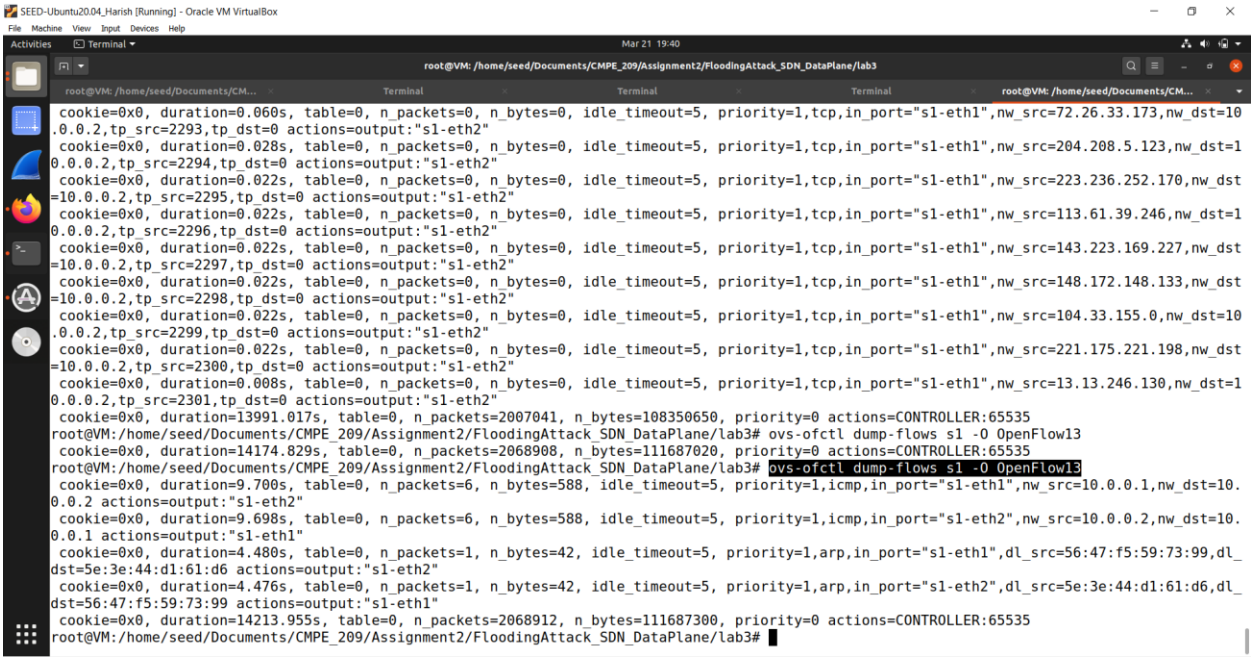


Fig. 97: Flow table rules of S1

Observation/Conclusion:

When the flow table of OVS switches is full, any additional flow-rule installation will be failed due to insufficient space in the flow table. A switch that cannot install a flow-entry will send an OFPT_ERROR message to the controller along with OFPFMFC_TABLE_FULL. The switch then drops the packet since it is unable to receive instructions to install a flow-entry due to the resource exhaustion. This is a DoS attack in the data plane of SDN.

APPENDIX

freq.py:

```
#!/usr/bin/env python3

from collections import Counter
import re

TOP_K  = 20
N_GRAM = 3

# Generate all the n-grams for value n
def ngrams(n, text):
    for i in range(len(text) -n + 1):
        # Ignore n-grams containing white space
        if not re.search(r'\s', text[i:i+n]):
           yield text[i:i+n]

# Read the data from the ciphertext
with open('ciphertext.txt') as f:
    text = f.read()

# Count, sort, and print out the n-grams
for N in range(N_GRAM):
   print("-----------------------------------")
   print("{}-gram (top {}):".format(N+1, TOP_K))
   counts = Counter(ngrams(N+1, text))        # Count
   sorted_counts = counts.most_common(TOP_K)  # Sort
   for ngram, count in sorted_counts:
      print("{}: {}".format(ngram, count))   # Print
```

cipher.txt
```
ytn xqavhq yzhu  xu qzupvd ltmat qnncq vgxzy hmrty vbynh ytmq ixur qyhvurn
vlvhpq yhme ytn gvrrnh bnniq imsn v uxuvrnuvhmvu yxx

ytn vlvhpq hvan lvq gxxsnupnp gd ytn pncmqn xb tvhfnd lnmuqynmu vy myq xzyqny
vup ytn veevhnuy mceixqmxu xb tmq bmic axcevud vy ytn nup vup my lvq qtvenp gd
ytn ncnhrnuan xb cnyxx ymcnq ze givasrxlu eximymaq vhcavupd vaymfmqc vup
v uvymxuvi axufnhqvymxu vq ghmnb vup cvp vq v bnfnh phnvc vgxzy ltnytnh ytnhn
xzrty yx gn v ehnqmpnuy lmubhnd ytn qnvqxu pmpuy ozqy qnnc nkyhv ixur my lvq
nkyhv ixur gnavzqn ytn xqavhq lnhn cxfnp yx ytn bmhqy lnnsnup mu cvhat yx
vfxmp axubimaymur lmyt ytn aixqmur anhncxud xb ytn lmuynh xidcemaq ytvusq
ednxuratvur

xun gmr jznqymxu qzhhxzupmur ytmq dnvhq vavpncd vlvhpq mq txl xh mb ytn
```

anhncxud lmii vpphnqq cnyxx nqenamviid vbynh ytn rxipnu rixgnq ltmat gnavcn v ozgmivuy axcmurxzy evhyd bxh ymcnq ze ytn cxfncnuy qenvhtnvpnp gd exlnhbzi txiidlxxp lxcnu ltx tnienp hvmqn cmiimxuq xb pxiivhq yx bmrty qnkzvi tvhvqqcnuy vhxzup ytn axzuyhd

qmruvimur ytnmh qzeexhy rxipnu rixgnq vyynupnnq qlvytnp ytncqnifnq mu givas qexhynp iveni emuq vup qxzupnp xbb vgxzy qnkmqy exlnh mcgvivuanq bhxc ytn hnp avheny vup ytn qyvrn xu ytn vmh n lvq aviinp xzy vgxzy evd munjzmyd vbynh myq bxhcnh vuatxh avyy qvpinh jzmy xuan qtn invhunp ytvy qtn lvq cvsmur bvh inqq ytvu v cvin axtxqy vup pzhmur ytn anhncxud uvyvimn exhycvu yxxs v gizuy vup qvymqbdmur pmr vy ytn viicvin hxqynh xb uxcmuvynp pmhnayxhq txl axzip ytvy gn yxeenp

vq my yzhuq xzy vy invqy mu ynhcq xb ytn xqavhq my ehxgvgid lxuy gn

lxcnu mufxifnp mu ymcnq ze qvmp ytvy viytxzrt ytn rixgnq qmrumbmnp ytn mumymvymfnq ivzuat ytnd unfnh muynupnp my yx gn ozqy vu vlvhpq qnvqxu avcevmru xh xun ytvy gnavcn vqqxamvynp xuid lmyt hnpavheny vaymxuq muqynvp v qexsnqlxcvu qvmp ytn rhxze mq lxhsmur gntmup aixqnp pxxhq vup tvq qmuan vcvqqnp cmiimxu bxh myq inrvi pnbnuqn bzup ltmat vbynh ytn rixgnq lvq bixxpnp lmyt ytxzqvupq xb pxuvymxuq xb xh inqq bhxc enxein mu qxcn axzuyhmnq

ux avii yx lnvh givas rxluq lnuy xzy mu vpfvuan xb ytn xqavhq ytxzrt ytn cxfncnuy lmii vicxqy anhyvmuid gn hnbnhnuanp gnbxhn vup pzhmur ytn anhncxud nqenamviid qmuan fxavi cnyxx qzeexhynhq imsn vqtind ozpp ivzhv pnhu vup umaxin smpcvu vhn qatnpzinp ehnqnuynhq

vuxytnh bnvyzhn xb ytmq qnvqxu ux xun hnviid suxlq ltx mq rxmur yx lmu gnqy emayzhn vhrzvgid ytmq tveenuq v ixy xb ytn ymcn muvhrzvgid ytn uvmigmynh uvhhvymfn xuid qnhfnq ytn vlvhpq tden cvatmun gzy xbynu ytn enxein bxhnavqymur ytn hvan qxaviinp xqavhxixrmqyq avu cvsn xuid npzavynp rznqqnq

ytn lvd ytn vavpncd yvgzivynq ytn gmr lmuunh pxnquy tnie mu nfnhd xytnh avynrxhd ytn uxcmunn lmyt ytn cxqy fxynq lmuq gzy mu ytn gnqy emayzhn avynrxhd fxynhq vhn vqsnp yx imqy ytnmh yxe cxfmnq mu ehnbnhnuymvi xhpnh mb v cxfmn rnyq cxhn ytvu enhanuy xb ytn bmhqyeivan fxynq my lmuq ltnu ux cxfmn cvuvrnq ytvy ytn xun lmyt ytn bnlnqy bmhqyeivan fxynq mq nimcmuvynp vup myq fxynq vhn hnpmqyhmgzynp yx ytn cxfmnq ytvy rvhunhnp ytn nimcmuvynp gviixyq qnaxupeivan fxynq vup ytmq axuymuznq zuymi v lmuunh ncnhrnq

my mq vii ynhhmgid axubzqmur gzy veevhnuyid ytn axuqnuqzq bvfxhmyn axcnq xzy vtnvp mu ytn nup ytmq cnvuq ytvy nupxbqnvqxu vlvhpq atvyynh mufvhmvgid mufxifnq yxhyzhnp qenazivymxu vgxzy ltmat bmic lxzip cxqy imsnid gn fxynhq qnaxup xh ytmhp bvfxhmyn vup ytnu njzviid yxhyzhnp axuaizqmxuq vgxzy ltmat bmic cmrty ehnfvmi

mu my lvq v yxqqze gnylnnu gxdtxxp vup ytn nfnuyzvi lmuunh gmhpcvu
mu lmyt ixyq xb nkenhyq gnyymur xu ytn hnfnuvuy xh ytn gmr qtxhy ytn
ehmwn lnuy yx qexyimrty ivqy dnvh unvhid vii ytn bxhnavqynhq pnaivhnp iv
iv ivup ytn ehnqzceymfn lmuunh vup bxh ylx vup v tvib cmuzynq ytnd lnhn
axhhnay gnbxhn vu nufnixen quvbz lvq hnfnvinp vup ytn hmrtybzi lmuunh
cxxuimrty lvq ahxlunp

ytmq dnvh vlvhpq lvyatnhq vhn zunjzviid pmfmpnp gnylnnu ythnn gmiigxvhpq
xzyqmpn nggmur cmqqxzhm ytn bvfxhmyn vup ytn qtven xb lvynh ltmat mq
ytn gvrrnhq ehnpmaymxu lmyt v bnl bxhnavqymur v tvmi cvhd lmu bxh rny xzy

gzy vii xb ytxqn bmicq tvfn tmqyxhmavi xqavhfxymur evyynhuq vrvmuqy ytnc ytn
qtven xb lvynh tvq uxcmuvymxuq cxhn ytvu vud xytnh bmic vup lvq viqx
uvcnp ytn dnvhq gnqy gd ytn ehxpzanhq vup pmhnayxhq rzmipq dny my lvq uxy
uxcmuvynp bxh v qahnnu vayxhq rzmip vlvhp bxh gnqy nuqncgin vup ux bmic tvq
lxu gnqy emayzhn lmytxzy ehnfmxzqid ivupmur vy invqy ytn vayxhq uxcmuvymxu
qmuan ghvfntnvhy mu ytmq dnvh ytn gnqy nuqncgin qvr nupnp ze rxmur yx
ythnn gmiigxvhpq ltmat mq qmrumbmavuy gnavzqn vayxhq cvsn ze ytn vavpncdq
ivhrnqy ghvuat ytvy bmic ltmin pmfmqmfn viqx lxu ytn gnqy phvcv rxipnu rixgn
vup ytn gvbyv gzy myq bmiccvsnh cvhymu capxuvrt lvq uxy uxcmuvynp bxh gnqy
pmhnayxh vup vevhy bhxc vhrx cxfmnq ytvy ivup gnqy emayzhn lmytxzy viqx
nvhumur gnqy pmhnayxh uxcmuvymxuq vhn bnl vup bvh gnylnnu

result_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

WOMEN INVOLVED IN TIMES UP SAID THAT ALTHOUGH THE GLOBES SIGNIFIED THE INITIATIVES LAUNCH THEY NEVER INTENDED IT TO BE JUST AN AWARDS SEASON CAMPAIGN OR ONE THAT BECAME ASSOCIATED ONLY WITH REDCARPET ACTIONS INSTEAD A SPOKESWOMAN SAID THE GROUP IS WORKING BEHIND CLOSED DOORS AND HAS SINCE AMASSED MILLION FOR ITS LEGAL DEFENSE FUND WHICH AFTER THE GLOBES WAS FLOODED WITH THOUSANDS OF DONATIONS OF OR LESS FROM PEOPLE IN SOME COUNTRIES

NO CALL TO WEAR BLACK GOWNS WENT OUT IN ADVANCE OF THE OSCARS THOUGH THE MOVEMENT WILL ALMOST CERTAINLY BE REFERENCED BEFORE AND DURING THE CEREMONY
ESPECIALLY SINCE VOCAL METOO SUPPORTERS LIKE ASHLEY JUDD LAURA DERN AND NICOLE KIDMAN ARE SCHEDULED PRESENTERS

ANOTHER FEATURE OF THIS SEASON NO ONE REALLY KNOWS WHO IS GOING TO WIN BEST PICTURE ARGUABLY THIS HAPPENS A LOT OF THE TIME INARGUABLY THE NAILBITER NARRATIVE ONLY SERVES THE AWARDS HYPE MACHINE BUT OFTEN THE PEOPLE FORECASTING
THE RACE SOCALLED OSCAROLOGISTS CAN MAKE ONLY EDUCATED GUESSES

THE WAY THE ACADEMY TABULATES THE BIG WINNER DOESNT HELP IN EVERY OTHER CATEGORY THE NOMINEE WITH THE MOST VOTES WINS BUT IN THE BEST PICTURE CATEGORY VOTERS ARE ASKED TO LIST THEIR TOP MOVIES IN PREFERENTIAL ORDER IF A MOVIE GETS MORE THAN PERCENT OF THE FIRSTPLACE VOTES IT WINS WHEN NO MOVIE MANAGES THAT THE ONE WITH THE FEWEST FIRSTPLACE VOTES IS ELIMINATED AND ITS VOTES ARE REDISTRIBUTED TO THE MOVIES THAT GARNERED THE ELIMINATED BALLOTS
SECONDPLACE VOTES AND THIS CONTINUES UNTIL A WINNER EMERGES

IT IS ALL TERRIBLY CONFUSING BUT APPARENTLY THE CONSENSUS FAVORITE COMES OUT AHEAD IN THE END THIS MEANS THAT ENDOFSEASON AWARDS CHATTER INVARIABLY INVOLVES TORTURED SPECULATION ABOUT WHICH FILM WOULD MOST LIKELY BE VOTERS

SECOND OR THIRD FAVORITE AND THEN EQUALLY TORTURED CONCLUSIONS ABOUT WHICH FILM MIGHT PREVAIL

IN IT WAS A TOSSUP BETWEEN BOYHOOD AND THE EVENTUAL WINNER BIRDMAN
IN WITH LOTS OF EXPERTS BETTING ON THE REVENANT OR THE BIG SHORT THE
PRIZE WENT TO SPOTLIGHT LAST YEAR NEARLY ALL THE FORECASTERS DECLARED LA
LA LAND THE PRESUMPTIVE WINNER AND FOR TWO AND A HALF MINUTES THEY WERE
CORRECT BEFORE AN ENVELOPE SNAFU WAS REVEALED AND THE RIGHTFUL WINNER
MOONLIGHT WAS CROWNED

THIS YEAR AWARDS WATCHERS ARE UNEQUALLY DIVIDED BETWEEN THREE BILLBOARDS
OUTSIDE EBBING MISSOURI THE FAVORITE AND THE SHAPE OF WATER WHICH IS
THE BAGGERS PREDICTION WITH A FEW FORECASTING A HAIL MARY WIN FOR GET OUT

BUT ALL OF THOSE FILMS HAVE HISTORICAL OSCARVOTING PATTERNS AGAINST THEM THE
SHAPE OF WATER HAS NOMINATIONS MORE THAN ANY OTHER FILM AND WAS ALSO
NAMED THE YEARS BEST BY THE PRODUCERS AND DIRECTORS GUILDS YET IT WAS NOT
NOMINATED FOR A SCREEN ACTORS GUILD AWARD FOR BEST ENSEMBLE AND NO FILM HAS
WON BEST PICTURE WITHOUT PREVIOUSLY LANDING AT LEAST THE ACTORS NOMINATION
SINCE BRAVEHEART IN THIS YEAR THE BEST ENSEMBLE SAG ENDED UP GOING TO
THREE BILLBOARDS WHICH IS SIGNIFICANT BECAUSE ACTORS MAKE UP THE ACADEMYS
LARGEST BRANCH THAT FILM WHILE DIVISIVE ALSO WON THE BEST DRAMA GOLDEN GLOBE
AND THE BAFTA BUT ITS FILMMAKER MARTIN MCDONAGH WAS NOT NOMINATED FOR BEST
DIRECTOR AND APART FROM ARGO MOVIES THAT LAND BEST PICTURE WITHOUT ALSO
EARNING BEST DIRECTOR NOMINATIONS ARE FEW AND FAR BETWEEN

AES128CBC:
decrypted text:
This is the plain text used for encryption

AES128CFB:
decrypted text:
This is the plain text used for encryption

BFCBC:
decrypted text:
This is the plain text used for encryption

RSA public key encryption:

---

bn_sample.c

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
```

```
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();
BIGNUM *a = BN_new();
BIGNUM *b = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
// Initialize a, b, n
BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
BN_dec2bn(&b, "273489463796838501848592769467194369268");
BN_rand(n, NBITS, 0, 0);
// res = a*b
BN_mul(res, a, b, ctx);
printBN("a * b = ", res);
// res = aˆb mod n
BN_mod_exp(res, a, b, n, ctx);
printBN("aˆc mod n = ", res);
return 0;
}
```

Task1.c

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{
   char *number_str = BN_bn2hex(a);
   printf("%s %s\n", msg, number_str);
   OPENSSL_free(number_str);
}
int main()
{
   printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task1 -
Deriving the Private Key\n");
   BN_CTX *ctx = BN_CTX_new();
   BIGNUM *p = BN_new();
   BIGNUM *q = BN_new();
```

```
    BIGNUM *phi = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *p_minus_1 = BN_new();
    BIGNUM *q_minus_1 = BN_new();

    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    BN_sub(p_minus_1, p, BN_value_one());
    BN_sub(q_minus_1, q, BN_value_one());
    BN_mul(n, p, q, ctx);
    BN_mul(phi, p_minus_1, q_minus_1, ctx);

    BN_mod_inverse(d, e, phi, ctx);

    printBN("public key e = ", e);
    printBN("public key n = ", n);
    printBN("private key d = ", d);
    return 0;
}
```

Task2.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task2 -
Encrypting a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M = BN_new(); //Message
```

```
    BIGNUM *p = BN_new(); //Plain Text
    BIGNUM *c = BN_new(); //cypher text

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&M, "4120746f702073656372657421"); //"A top secret!"

    BN_mod_exp(c, M, e, n, ctx);
    BN_mod_exp(p, c, d, n, ctx);
    printBN("Encryption result: ", c);
    printBN("Plain Text: ", p); //For verification
    return 0;
}
```

Task3.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{ /* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task3 -
Decrypting a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *p = BN_new(); //plain text
    BIGNUM *C = BN_new(); //cypher text

    // Assign values
    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
```

```
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&C,
"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    // decrypt C: C^d mod n
    BN_mod_exp(p, C, d, n, ctx);
    printBN("Decryption result: ", p);
    return 0;
}
```

Task4.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{ /* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task4 -
Signing a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M1 = BN_new();
    BIGNUM *M2 = BN_new();
    BIGNUM *sign1 = BN_new();
    BIGNUM *sign2 = BN_new();

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&M1, "49206f776520796f75202432303030"); // hex encode for "I owe you $2000"
    BN_hex2bn(&M2, "49206f776520796f75202433303030"); // hex encode for "I owe you $3000"

    // encrypt M: M^d mod n
```

```
    BN_mod_exp(sign1, M1, d, n, ctx);
    BN_mod_exp(sign2, M2, d, n, ctx);
    printBN("Signature of M1:", sign1);
    printBN("Signature of M2:", sign2);
    return 0;
}
```

Pseudo Random Number Generation:

Task1.c:

```
//Generate Encryption Key in a Wrong Way
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
void main()
{
printf("CMPE209-Assignment3-016707314-Pseudo Random Number Generator - Task1- Generate Encryption Key in a Wrong Way\n");
int i;
char key[KEYSIZE];
printf("%lld\n", (long long) time(NULL));
//srand (time(NULL));
for (i = 0; i< KEYSIZE; i++){
key[i] = rand()%256;
printf("%.2x", (unsigned char)key[i]);
}
printf("\n");
}
```

MD5 Collison Attack Lab:

```
#!/usr/bin/python3
from sys import argv

_, first, second = argv

with open(first, 'rb') as f:
    f1 = f.read()
with open(second, 'rb') as f:
    f2 = f.read()

for i in range(min(len(f1), len(f2))):
    if f1[i] != f2[i]:
        print("different bytes in", hex(i), ": " ,hex(f1[i]) + ' vs ' + hex(f2[i]))
```

prefix.txt:

this is md5 lab

Mininet

lab3.py

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types


from ryu.lib.packet import in_proto
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from ryu.lib.packet import tcp
from ryu.lib.packet import udp

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
```

```
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.  The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id, priority=priority, match=match,
instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)



    def add_flow1(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id, priority=priority, idle_timeout=5,
match=match, instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority, idle_timeout=5,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
    dst = eth.dst
    src = eth.src

    dpid = format(datapath.id, "d").zfill(16)
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        if eth.ethertype == ether_types.ETH_TYPE_IP:
            ip = pkt.get_protocol(ipv4.ipv4)
            srcip = ip.src
            dstip = ip.dst
            protocol = ip.proto

            # if ICMP Protocol
            if protocol == in_proto.IPPROTO_ICMP:
```

```
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol)

            #  if TCP Protocol
            elif protocol == in_proto.IPPROTO_TCP:
                t = pkt.get_protocol(tcp.tcp)
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol, tcp_src=t.src_port, tcp_dst=t.dst_port,)

            #  If UDP Protocol
            elif protocol == in_proto.IPPROTO_UDP:
                u = pkt.get_protocol(udp.udp)
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol, udp_src=u.src_port, udp_dst=u.dst_port,)
        # verify if we have a valid buffer_id, if yes avoid to send both
        if eth.ethertype == ether_types.ETH_TYPE_ARP:
            match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_ARP, in_port=in_port, eth_dst=dst,
eth_src=src)


        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow1(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow1(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```