Technical Portfolio: Cryptanalysis & Secure Design

Vulnerability Analysis & Exploit Development

Rishi Vardhan Majji
Sophomore — Indian Institute of Technology Bombay

Abstract
This portfolio compiles detailed technical reports on cryptographic protocol security, including practical implementations of ECDSA private key recovery via nonce reuse, Merkle–Damgård hash collisions, and timing side-channel analysis.

Academic Session: Autumn 2025
Guide: Prof. Sruthi Sekar

Challenge 1

Rishi Vardhan Majji
24B0969

August 13, 2025

1 Description

The one-time pad, as you have learnt, is perfectly secure. However, its perfect secrecy depends on the key being used exactly once. If the same key is reused, the resulting "two-time pad" is no longer secure as XORing the two ciphertexts gives you the XOR of the plaintexts, which is a lot of information about the two messages. In fact, if the messages are in natural language, it may even allow you to extract both the messages fully given their XOR. In this challenge, you'll do exactly that. ciphertext1.enc and ciphertext2.enc contain two one-time pad encrypted ciphertexts en crypted with the same key. One of them is a "flag" used in a Capture The Flag competition (hint: it starts with cs409) and the other one is a normal English sentence. Use the redundancy of the English language to figure out the entire flag. Feel free to look at encrypt.py to understand how to use the strxor function to XOR two byte objects in python.

2 Approach

As using XOR operation on the ciphertexts gives us the result of XOR operation on the original Flag and Message, using XOR on this and the part **"cs409{"** (elongated to match the length of message, we can get the first 6 letters of the message.Which would be **"Crypta"**.

Now we can anticipate what the "Crypta" thing could actually be, and its **Cryptanalysis**. Using this method,we continuously predict the flag and message by their initial parts and get it.

```
with open("ciphertext1.enc", 'rb') as f:
    c1=f.read()
```

```
with open("ciphertext2.enc", 'rb') as f:
    c2=f.read()

c = strxor(c1,c2)
```

Get the XOR of the ciphertexts which is the same as that of the flag and message, call it **c**

```
text1=b'cs409{'
text2=b''

if len(text1)< len(c):
    text1=text1.ljust(len(c))
else :
    text1=text1[:len(c)]

if len(text2)< len(c):
    text2=text2.ljust(len(c))
else :
    text2=text2[:len(c)]
```

Start with the texts you know initially and extend them to the length of the ciphertexts.

```
c3=strxor(text1,c)
c4=strxor(text2,c)
print(c3)
print(c4)
```

We are going to XOR text 1 with c (which is XOR of text1 and text2) and the same with text2. So we should get text2 as c3 and text1 as c4.We then print these and analyse what could come next in text 1 and 2 simultaneously.

The process looks somewhat like this

```
b'Crypta'
b'cs409{one_time'
b'Cryptanalysis freq'
b'cs409{one_time_pad_key_re'
b'Cryptanalysis frequently invo'
b'cs409{one_time_pad_key_reuse_compr'
b'Cryptanalysis frequently involves statistical att'
b'cs409{one_time_pad_key_reuse_compromises_security!!!'
b'Cryptanalysis frequently involves statistical attacks'
```

We slowly reach

```
text1=b'cs409{one_time_pad_key_reuse_compromises_security!!!}'
text2=b'Cryptanalysis frequently involves statistical attacks'
```

Challenge 2

Rishi Vardhan Majji
24B0969

August 2025

1 Description

Recall that a finite group is a finite set G along with a binary operation $\cdot$: G$\times$G $\rightarrow$ G such that the binary operation $\cdot$ satisfies the following conditions:
1. (Associativity) x $\cdot$ (y $\cdot$ z) = (x $\cdot$ y) $\cdot$ z for any x,y,z ϵ G.

2. (Identity) There exists an element e ϵ G such that x $\cdot$ e = e $\cdot$ x = x for

any x ϵ G. e is called the identity element of G.
3. (Inverse) For every x ϵ G, there exists an element y ϵ G such that

x $\cdot$ y = y $\cdot$ x = e. The element y is called the inverse of x.
For instance, the set 0,1 with the binary operation of XOR forms a finite

group. The identity element is 0, the inverse of 0 is 0 and the inverse of 1 is 1. Another example of a finite group is Zn, the set of all integers modulo n for some natural number n. Given a finite group G and a positive integer m, the group Gm consisting of m-tuples of elements from G is also a group, with the group operation being co-ordinate wise application of the group operation of G.// As discussed in class, the one-time pad can actually be used for messages in any finite group. When using the group Gm, this requires that key is in the form a string of m "characters" each of which is uniformly sampled from G, independent of each other.
In this challenge, you will see a (faulty) implementation of the one-time pad over the group Zm 128 (for some m). Each message character is converted to its 7-bit ASCII representation and treated as a group element of Z128. You do not know the key used in the encryption, but you suspect that it is not uniformly random. Look at the encryption script in encryptor.py and f igure out a way to retrieve the original message.

2 Approach

Here, as per the encryptor

```
key = chr(random.randint(0,127)).encode()
    for _ in range(1, len(plaintext)):
        key += chr(hashlib.sha256(key).digest()[0]%128).encode()
```

It chooses a random integer in 0 to 127 and creates a key based on that integer. The point is that a hash is like a finger print, so for a particular integer you start with, you always get the same key at the end. So there are only 128 keys possible for this encryption, so we are going to simulate the key generating function and brute force it.

```
for i in range(0,128):
    print(i)
    print("..........")
    key=chr(i).encode()
    for a in range(1, len(encflag)):
        key += chr(hashlib.sha256(key).digest()[0]%128).encode()
```

So we are looping this for all integers from 0 to 127, creating every key.

```
    flag=b""
    for i in range(len(key)):
        flag+= chr((encflag[i]+127*key[i])%128).encode()
    print (flag)
```

And we are decrypting the cipher using all these keys , and we can just search for the one which is sensible.

```
0
..........
b'hl`w@\x1f!\t\x06JU%qB Hf=00~#:DK(1x!*)x!('
1
..........
b'g\x0fU5i\x1cvX\\fJ3Ev\x1f\x0c0,^e\x14\x7f~a%.M\x06z8\x11Ul*'
2
..........
b'f\x7fTuE[uaRn6Ux=(KK?\x00H\t\x05yYJQY\x15<\x06B|YH'
3
```

```
..........
b'eRi7\x1c\x06T>I(;\x12hU0e\x1b\n`b>W?>\x05R\x16\x02\x04\x10\x1f\x13\x19@'
4
..........
b"du_1Q\x17_O{9dO<YdB\x13&%_>\x14\x01S\x0f]mv8\x0cDi+'"
5
..........
b'cs409{algebra_enters_the_picture!}'
```

And as you can see that we got our flag for the key made from integer **5**. Hence, our key is **cs409{algebra_enters_the_picture!}**

Challenge 3

Rishi Vardhan Majji

August 2025

1 Description

True one-time pad encryption, as you have studied, is perfectly secure, that is, there is no way for any (potentially unbounded) adversary to distinguish between the encryption of string under one time pad and a string sampled uniformly at random (of the appropriate length) with a non-zero advantage.
Bob, however, felt that the byte 0x00 in the key could potentially lead to security issues because it does not change the message character at all during the encryption. So instead of sampling each key-byte from the interval [0x00, 0xff], he decided to instead sample the key-byte uniformly from the interval [0x01, 0xff]. Show that Bob's encryption scheme is no longer perfectly secure by distinguishing between the encryption of a string you provide and the encryption of a random message. (This is slightly different from IND-onetime security as defined in class, but it is also equivalent to perfect secrecy.)

2 Approach

As the byte 0x00 is removed from the interval from where the key takes its bytes from, The key can never have 0x00 byte in it, meaning that the ciphertext can never have any byte as the same as the message, because for that to happen, the key should be having the 0x00 byte at that location.
So we are going to give an input message such that we have a high chance of finding the same bit of our original message in the randomly generated message.
So lets just give an input of **'00'** (hex) , so if any of the two encryptions have

the byte '00' in them, then it should be the one made from a random message.
But we cannot be sure as just one bit is not enough as the probability of the bit

being '0x00' is 1/256 . So we are going to give a large input -¿ A huge string of '00' bits long enough to have a good probability of getting atleast one 0x00 bit in the randomly generated encrypted message and so that we can distinguish between them.

```
L=5000
```

```
payload = "00"*L
```

Making the input long enough for high chances of distinguishability.

```
def func(c):
        x=bytes.fromhex(c)
        return b'\x00' in x
```

Creating a function to check if the message has the byte '00' in it

```
if func(c1) and not func(c2) :guess=2
elif func(c2) and not func(c1): guess=1
else :guess=1
```

If c1 has it, then it must be the fake one, so c2 would be our encrypted msg and vice versa.
And if both of them dont have it(Rare),then we just guess it to 1. **(So we take a high value of 'L' to reduce the probability of this happening)**

And we end up geting the key cs409{y0u_h4d_fu11_4dv4nt4g3}

Challenge 4

Rishi Vardhan Majji

August 2025

1 Description

Suppose you are given a long key generated using the method from the previous challenge (uni formly sampling each key byte from [0x01, 0xff]). Could you still repurpose this key somehow to make a perfectly secure encryption scheme?
One way to do so is as follows: Say we denote the plaintext by m1,m2,...,mn where each mi is a single byte. Considering each byte as a number in base-256, we can think of the message as a number

$$m = m_1 * 256^{n-1} + m_2 * 256^{n-2} + \ + m_{n-1} * 256 + m_n$$

in decimal. Converting this number to base-255, say we get the base-255 representation as p1p2...pn', that is,

$$m = p_1 * 255^{n'-1} + p_2 * 255^{n'-2} + \ + p_{n'-1} * 255 + p_{n'}$$

, where pi ϵ [0,254] for each i ϵ [1,n'] (where n' is fixed ahead of time– what should it be?). Define the ciphertext as follows: say that for i ϵ [1,n'], we have ci = (pi + ki -1) (mod 255) where ki is the ith byte of the key interpreted as an integer in [1,255]. Interpret c = c1c2...cn' as a number in base-255 and convert it back to base-256 to get a byte sequence which forms the ciphertext. Can you argue that this method of encryption is perfectly secure?

2 Approach

We Just reverse the whole process

```
with open ("ciphertext.enc",'rb') as f:
    x=f.read()
with open ("keyfile",'rb') as f:
    k=f.read()
```

Read the key and cipherthext to x and f.

```
value=0
a=1
for i in range(len(x),0,-1):
    value+=x[i-1]*a
    a*=256
```

From the ciphertext, as we want to convert it to base 255 from 256, we get the value of it.
vspace1em

```
c=[]
for i in range(len(x)):
    c.append(value%255)
    value//=255

c[:]=c[::-1]
```

And then we get the base 255 coefficients from the value, put it in list c.

```
p=[]
for i in range(len(c)):
    p.append((c[i]-k[i]+1)%255)
```

And then, we create list p, and revert the process we used to generate cipher to get the coefficients related to our original message.

```
value=0
a=1
for i in range(len(c),0,-1):
    value+=p[i-1]*a
    a*=255
```

Now,as we want the coefficients of base 256, we convert the base 255 version to a value.

```
m=[]
for i in range(len(c)):
    m.append(value%256)
    value//=256

m[:]=m[::-1]
```

And then, from the value, we get the coefficients of base 256 for our original message.

```
byte_seq = bytes(m)
flag = byte_seq.decode('utf-8')
```

And now, we just decode the bytes to get out message.

Lab2 Report

Rishi Vardhan Majji
24B0969

September 2025

1 Challenge 1: openssl Decryption

You've been provided a file ciphertext.bin which contains an encryption of the flag encrypted using aes-128-cbc with the key given in key.hex and the IV given in iv.hex.

1.1 Approach

Using openssl, this qsn can be solved using the terminal.

enc = uses the symmetric cipher functions
-d = decrypt
-aes-128-cbc = specifies the encryption scheme
-in = input file
-K = specifies the key value
-out = output flag text file

```
openssl enc -aes-128-cbc -d \
 -in ciphertext.bin \
 -K "$(cat key.hex | tr -d '\n')" \
 -iv "$(cat iv.hex | tr -d '\n')" \
 -out -
```

```
flag-->cs409{op3n551_2_d3crypt10n_1_4m}
```

we get the flag **cs409{op3n551_2_d3crypt10n_1_4m}**

2 Challenge 2: The Electron Code

2.1 Description

The ECB (Electronic Code Book) mode of encryption is a "bad" mode of encryption because it always maps the same block of plaintext to the same block of ciphertext.
Thus, if you have a long message such that the same block repeats twice in the plaintext, the same block would repeat in the exact same position in the ciphertext too. This leaks information about the plaintext.

2.2 Approach

Since identical plaintext block produces the same ciphertext block,and each character is modified to repeat **16** times.
The byte block sizze of AES, so each **16** sized block of the cipher text corresponds to the a single character from the HEADER.
as we know the HEADER, we try to create a mapping for each character in the header to its corresponding ciphertext block.
Now, as the flag is encrypted using the same encryption, we can use the mapping we found from the HEADER part and find the corresponding character in the flag.

......The code goes as follows......

Mentioning the HEADER and the AES block size.

```
from Crypto.Cipher import AES

n = AES.block_size
HEADER = "_Have you heard about the \\{quick\\} brown fox which jumps over the
lazy dog?\n__The decimal number system uses the digits 0123456789!\n___The flag
is: "

ciphertext = open("ciphertext.bin","rb").read()
```

Initiating the map and filling it with the characters in HEADER

```
mymap = {}
for i in range(len(HEADER)):
    mymap[ciphertext[i*n:(i+1)*n]] = HEADER[i]
```

Looping through the ciphertext for the flag and finding the corresponding characters of it using the map.

```
    flag = ""
    for i in range(len(HEADER), len(ciphertext)//n):
        flag += mymap[ciphertext[i*n:(i+1)*n]]

    print(flag)


    #flag--> cs409{r3dund4nt_l34k4g35}
```

We get the flag **cs409{r3dund4nt_l34k4g35}**

3 Challenge 3: The Catastrophic Equality

3.1 Approach

In this challenge, the server uses AES in CBC mode, and the IV happens to be the same as the secret key.
First, we send some normal parameters to the server to get a real ciphertext block. Then we trick the server by sending a special ciphertext in the form C0 || zero_block || C0.
When the server tries to decrypt it, it ends up leaking part of the plaintext. Using this leaked data, we can figure out the key because P0 XOR P2 gives us the key (As IV = key).
After we get the key, we can locally encrypt a new parameter string that includes admin=true using the same key and IV. Finally, we send this ciphertext to the server, get admin access, and decrypt the flag to read it.

The code goes as follows...........

```
n = AES.block_size  # 16 bytes
```

We set `n` to 16 as AES works with 16-byte blocks.

```
ct_hex = choice1("a=b")
ct = bytes.fromhex(ct_hex)
C0 = ct[:n]
```

We send some normal parameters (‘a=b‘) to the server using `choice1`. It gives us a ciphertext. We convert it from hex to bytes and take the first block, ‘C0‘, which we’ll use in the next step.

```
# Make the server decrypt and leak plaintext by sending: C0 || zero_block || C0
mal = C0 + (b"\x00" * n) + C0
ok, leaked_hex = choice2(mal.hex())

leaked = bytes.fromhex(leaked_hex)

P0 = leaked[0:n]
P2 = leaked[2*n:3*n]
```

We create a special ciphertext: ‘C0 || zero_block || C0‘. When the server tries to decrypt it, it rejects it but leaks part of the plaintext. We convert the leaked hex back to bytes and split it: ‘P0‘ is the first block, ‘P2‘ is the third block. These will help us recover the key.

```
# Recover key (because IV == key): key = P0 XOR P2
key = strxor(P0, P2)
```

Since the IV is the same as the key, we can recover the key with a simple XOR: ‘key = P0 XOR P2‘.

```
# Locally encrypt a params string that contains admin=true using key as IV too.
payload = b"a=1&admin=true"
made = AES.new(key, AES.MODE_CBC, iv=key).encrypt(pad(payload, n))
```

We make a new parameters string containing ‘admin=true‘. We pad it to 16 bytes and encrypt it locally with AES in CBC mode, using the recovered key both as the key and as the IV. This gives us a crafted ciphertext that should give us admin access.

```
# Submit made ciphertext and print only the flag if we get admin.
got_admin, flag = choice2(made.hex())
flag_cipher = bytes.fromhex(flag)
flag = unpad(AES.new(key,AES.MODE_CBC,iv=key).decrypt(flag_cipher), n).decode()
print(flag)
```

We send the crafted ciphertext to the server using ‘choice2‘. If it works, the server returns the flag in encrypted form. We decrypt it locally with the same key and IV, remove padding, and print the flag.

```
    flag-->cs409{fu11_k3y_recovery_ftw_1mpl3m3nt_w1th_c4r3}
```

4 Challenge 4: Never Painted by the Numbers

4.1 Description

In this challenge, you will interact with a server which essentially acts as an echo server- it outputs back to you exactly what you input to it– except that the response is encrypted.
However, if you input !flag to it, the server instead outputs the flag to you– again, encrypted. The encryption is done using CTR mode, using a key you don't know. But you suspect that the implementation of the CTR mode encryption in the server has some vulnerability

4.2 Approach

CTR turns AES into a bytewise keystream XOR, so reusing the same key+nonce just shifts the same keystream around.
The server echoes our chosen plaintexts and encrypts them with predictable counters, so we can get ciphertexts for plaintexts we already know.
XORing a known plaintext with its ciphertext directly gives that segment of the keystream.
The flag is encrypted with the same keystream sequence but starting at some unknown offset, so we don't need every counter, just the right part
By sliding the keystream made across the flag ciphertext and XORing, one alignment will produce readable text.
When that alignment yields the expected flag pattern **cs409{** we've recovered the flag.

The code goes as follows...............

```
x=1000
known_plaintext=b'0'*x
```

Start by creating a long text thats known to us, we use long to get the key stream as long as possible so we can find our required chunk in it.

```
enc_inp_hex,enc_out_hex=send_to_server(known_plaintext.decode())
enc_inp=bytes.fromhex(enc_inp_hex)
```

Giving it to the server to get two encryptions with different ctr and convert to its byte form.

```
 k=strxor(enc_inp,known_plaintext)
```

This gives us the key stream used to encrypt this long message.

```
enc_flag_in_hex,enc_flag_out_hex=send_to_server("!flag")
flag_ciph=bytes.fromhex(enc_flag_out_hex)
```

Giving **!flag** as input to get the encryption of the required flag, as the required cipher text is in the second one(made by output encryption) we store it.

```
for i in range(len(k)-len(flag_ciph)):
    text=strxor(flag_ciph,k[i:i+len(flag_ciph)])
```

We take the flag ciphertext and try XORing it with slices of the long keystream k. keeping each slice length the same length as the flag.

```
        try:
            text = text.decode()
        except UnicodeDecodeError:
            text = text.decode(errors="ignore")
```

There will be many texts(the ones we dont need) who will have random bytes which arent valid in ASCII, so we try to ignore them.

```
        if "cs409{" in text:
            print(text)
            break

    #flag-> cs409{y0u_kn0w_th3_gr34t35t_f1lm5_of_4ll_t1m3_w3re_n3v3r_m4d3}
```

We print the text if it has the beginning portion **CS409{** in it, as it will be our flag

5 Bonus Challenge: Canis Lupus Familiaris

5.1 Description

A padding oracle attack is a cryptographic attack that targets the padding scheme used in block cipher modes, particularly in Cipher Block Chaining (CBC) mode.
In practice, padding is often added to plaintext messages to ensure that the length of the message is a multiple of the block size of the cipher being used. A padding oracle is a term used to describe a vulnerability that arises when an attacker can determine whether a given ciphertext has a valid padding or not. The padding is typically added to ensure that the plaintext can be properly aligned into blocks before encryption. The padding scheme we often use is PKCS#7.
The padding oracle attack can be extremely dangerous in the sense that one could completely recover the plaintext given access to a padding oracle
In this challenge, you will interact with a server which will act as a padding oracle

5.2 Approach

AES in CBC mode encrypts each plaintext block P_i by first XORing it with the previous ciphertext block C_{i-1} (or the IV for the first block), then applying the block cipher:

$$C_i = E_K(P_i \oplus C_{i-1}).$$

During decryption we first get the intermediate block $I := D_K(C_i)$, then recover the plaintext as

$$P_i = I \oplus C_{i-1}.$$

We exploit a *padding oracle* that tells us whether the decrypted plaintext has valid PKCS#7 padding. To recover a single byte $I[j]$:

- pick a padding value pad (1, 2, 3, ...), and build a forged previous block C' where we set the current test byte $C'[j] = I_{\text{guess}} \oplus \text{pad}$;
- for any bytes to the right of j that we already solved, adjust them as $C'[k] = I[k] \oplus \text{pad}$ so the padding stays valid;
- send (C', C_i) to the oracle — if it says "Valid Padding!", the guess I_{guess} is correct.

Once we know $I[j]$, the real plaintext byte is recovered as

$$P_i[j] = I[j] \oplus C_{i-1}[j].$$

Repeat this from the last byte to the first, and for every block. After decrypting all blocks, remove the PKCS#7 padding to get the full plaintext, including the hidden flag.

The code is as follows.................
....

```
n = 16  # AES block size
```

AES works with 16-byte blocks, so we set `n` to 16. We use this whenever we split data or remove padding.

```
def split_blocks(data: bytes, size: int = n):
    return [data[i:i+size] for i in range(0, len(data), size)]
```

This helper just cuts bytes into 16-byte pieces. We use it to turn the ciphertext into blocks so we can handle one block at a time.

```
def decrypt_block(prev_block: bytes, curr_block: bytes) -> bytes:
    intermediate = [0] * n   # will hold I = D_k(C) values
    plaintext = [0] * n      # will hold recovered P bytes
```

This function tries to recover one plaintext block. - `intermediate` stores the AES-decrypted bytes of the current ciphertext block (before XOR). - `plaintext` will store the real bytes we want after XORing with the previous block.

```
# work from last byte to first
for pos in range(n - 1, -1, -1):
    pad_val = n - pos
```

We solve bytes from the end of the block to the start. At each position we want the padding to look like `pad_val` (e.g. 1, 2, 3, ...).

```
for I_guess in range(256):
    forged = bytearray(b'\x00' * n)
```

We try all 0..255 values for the intermediate byte (call it `I_guess`) and build a fake previous block called `forged`.

```
for j in range(pos + 1, n):
    forged[j] = intermediate[j] ^ pad_val
```

For bytes we already found (the tail of the block), we set the forged block so those positions will decrypt to `pad_val`. This keeps the tail valid while we test the current byte.

```
forged[pos] = I_guess ^ pad_val
```

To test our guess, we set the current byte in the forged block to I_guess$\oplus$pad_val. If the real intermediate byte equals `I_guess`, the server will see valid padding.

```
if validate_padding(forged.hex(), curr_block.hex()):
    intermediate[pos] = I_guess
    plaintext[pos] = intermediate[pos] ^ prev_block[pos]
    break
```

We send the forged block and the current ciphertext to the server. If it replies "Valid Padding!", our guess is right. We then record the intermediate byte and compute the real plaintext byte by XORing with the previous block.

```
blocks = [IV] + split_blocks(flag_enc, n)
recovered = b""
for i in range(1, len(blocks)):
    recovered += decrypt_block(blocks[i-1], blocks[i])
```

We split the whole encrypted flag into blocks (IV first). Then we decrypt each ciphertext block using the previous block and collect the plaintext blocks into `recovered`.

```
flag = unpad(recovered, n).decode()
print(flag)
```

Finally, we remove PKCS#7 padding and print the flag as a readable string.

```
    flag-->cs409{sid3_ch4nn3l_danger!}
```

Lab3 Report

Rishi Vardhan Majji
24B0969

October 2025

1 Challenge 1: openssl Sorcery

You have been provided a message.txt file on which you must perform all the MAC computations. If you ever need any key or IV, use the ones provided in key.hex and iv.hex respectively.
MAC1: Compute the HMAC of message.txt with the SHA256 digest.
MAC2: Compute the CMAC of message.txt with AES-128 cipher in CBC mode.
MAC3: Compute the GMAC of message.txt with AES-128 cipher in GCM mode.

CODE:

```
# Read key and IV from files
KEY=$(cat key.hex)
IV=$(cat iv.hex)

# Compute HMAC-SHA256 of message.txt using the key
MAC1=$(openssl mac -digest SHA256 -macopt hexkey:$KEY -in message.txt HMAC \
| tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Compute CMAC-AES-128-CBC of message.txt using the key
MAC2=$(openssl mac -cipher AES-128-CBC -macopt hexkey:$KEY -in message.txt CMAC \
| tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Compute GMAC-AES-128-GCM of message.txt using key and IV
MAC3=$(openssl mac -cipher AES-128-GCM -macopt hexiv:$IV -macopt hexkey:$KEY -in\
message.txt GMAC | tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Print final flag in cs409{MAC1_MAC2_MAC3} format
printf 'cs409{%s_%s_%s}\n' "$MAC1" "$MAC2" "$MAC3"
```

2 Challenge 2: Extension of Deception

2.1 Description

You are provided the MAC digest of DATA (refer to the server script) and you need to submit the MAC digest of a string that starts with DATA and contains admin=true as a substring

2.2 Approach

We have the MAC of the original message $\rightarrow t_1$.
Need MAC of `originalmsg||&admin=true` $\rightarrow$ it's just continuing the MAC over the second block.
If $m_1||m_2 \rightarrow t_1||t_2$, we need to make $t_2 = \text{func}(t_1 \oplus m_2)$. We can get that without using m_1 by asking the oracle to MAC a single block chosen as $m_2 \oplus t_1 \oplus IV$.

The code goes as follows......................

```
original_mac_bytes = bytes.fromhex(original_mac)
iv_bytes = bytes.fromhex(iv)
```

we define the data we want to append

```
append_data = b"&admin=true"
append_block = pad(append_data, AES.block_size)  # this is m2 (padded)
```

Now craft the message for the oracle (Möbius Hacker)
mobius_data_bytes $= m2 \oplus t1 \oplus iv$

```
mobius_data_bytes = strxor(strxor(append_block, original_mac_bytes), iv_bytes)
mobius_data = mobius_data_bytes.hex()
```

now the oracle will compute $func(m2 \oplus t1) = t2$

What we are doing above–

1. Convert the server strings `original_mac` and `iv` from hex to bytes: `original_mac_bytes`, `iv_bytes`.
2. Set `append_data = b"&admin=true"` and pad it to a full AES block: this gives `append_block` $(= m_2)$.
3. Compute `mobius_data_bytes` $= m_2 \oplus t_1 \oplus IV$ using `strxor`. Hex-encode it to get `mobius_data`.
4. Send `mobius_data` to the oracle. The oracle will do AES_Enc(IV $\oplus$ mobius_data) = AES_Enc($t_1 \oplus m_2$) = the MAC we need for $m_1||m_2$.

Now, the second part

```
DATA = b"user=cs409learner&password=V3ry$3cur3p455"
DATA = pad(DATA, AES.block_size)

creds_bytes = DATA + append_block
creds = creds_bytes.hex()
```

now this mac would be the mac for $m1||m2$

```
forged_mac = mobius_mac
```

What we are doing above–

1. Pad the original credentials to full AES blocks: `DATA` becomes m_1 (padded).
2. Append `append_block` (padded `"admin=true"`) to get `creds_bytes` = $m_1||m_2$.
3. Hex-encode `creds_bytes` as `creds` and send to the server as your credentials.
4. The tag returned by the oracle (`mobius_mac`) equals the MAC for $m_1||m_2$, so set `forged_mac = mobius_mac` and send it to the server.

We used the oracle to compute $t_2 = \text{func}(t_1 \oplus m_2)$ by sending $m_2 \oplus t_1 \oplus IV$. The oracle returned the tag for $m_1||m_2$. We then sent the combined credentials and that forged tag to the server and obtained admin.

```
the flag -->cs409{53cur1ty_f0r_4ll_t1m3_4lw4y5}
```

3 Challenge 3: Tick-Tock on the HMAC

3.1 Description

Even if your MAC-scheme is secure, it could be vulnerable to other side-channel attacks. In fact, you will implement a timing-based side channel attack on insecure digest comparisons.
when you are comparing a user-submitted MAC digest to a computed MAC digest, you should never use the standard string comparison operator. This is because they usually perform an early exit when they find the first mismatch in the two strings. This leaks timing information in the following sense: if the string comparison takes longer, that means the user-submitted MAC has a longer matching prefix with the correct MAC.
This timing leak can be used to recover the full MAC, as you will do in this challenge. An artificial timing delay has been introduced in this challenge where each successful byte com parison consumes 1 second. Use this to your advantage to guess the first 10 characters of the MAC's hexdigest

3.2 Approach

The server compares your HMAC guess byte-by-byte and exits early when a mismatch occurs. Each correct byte comparison adds an artificial delay of **1 second**. So: longer response time means a longer matching prefix. We use timing (and the server's "omniscient" reply when we guess exactly) to recover the first 10 hex characters of the MAC.

The code goes as follows......................

```
# The server told us the required message length in msg_len
msg = b"We are what we repeatedly do. Excellence, then, is not an act, but a habit, said A
msg = msg[:msg_len]
msg = msg.hex()
```

What we are doing above–

1. Prepare the plaintext message we want to send as bytes.
2. Truncate it to the exact length requested by the server (`msg_len`).
3. Hex-encode the truncated bytes and send that as the message.

second part–

```
HEX = "0123456789abcdef"
decoded = ""

# keep guessing until we have 10 hex chars
while len(decoded) < 10:
    found = False
    for x in HEX:
        candidate = decoded + x
        guess = candidate + "0" * (10 - len(candidate))
# build candidate: decoded + guess + padding to length 10
# do a few trials to reduce noise, take average time
        trials = 3
        times = []
        result_ = -1
        for i in range(trials):
            start = time.time()
            result = send_guess(guess)
            elapsed = time.time() - start
            times.append(elapsed)
            if result == 1:               # if server says omniscient -> done
                decoded = candidate
                result_ = 1
                break
        if result_ == 1:
            found = True
            break

        avg_t = sum(times) / len(times)
        # if avg time exceeds current matched-prefix length by ~0.5s(for latency),
        # we take it as matched
        if avg_t > (len(decoded)+0.5):
            decoded += x
            found = True
            break

    if not found:
        continue
```

if we reached full 10 chars, stop

```
    if len(decoded) == 10:
        break
```

What we are doing above–

1. Try each hex nibble (0–f) for the next position of the 10-char prefix.
2. For a candidate nibble, form a 10-char guess by padding the rest with zeros (server only checks the prefix).
3. Send the guess multiple times (here 3 trials) and record the elapsed time for each trial to reduce noise.
4. If the server replies with the special success message ("omniscient"), we got the exact prefix — set `decoded` and stop.
5. Otherwise compare the average elapsed time: if it's noticeably larger than the current matched-prefix length (we use a margin like +0.5s to tolerate latency), treat the guess("x") as correct and append it to `decoded`.
6. Repeat until we recover all 10 hex characters.

We exploit the byte-by-byte comparison timing leak: each matched byte adds 1s. By measuring response time (and confirming with the server's exact-match reply), we recover the first 10 hex chars of the MAC.

```
the flag --> cs409{k3$h4_0r_t4yl0r_5w1ft?}
```

4 Challenge 4: Commitment Issues (Merkle's Version)

4.1 Description

The server constructs a Merkle tree on top of the flag. Each character of the flag forms a lead node of the Merkle tree. The length of the flag is n, which is assured to be a perfect power of two in this challenge.
You are allowed to make n/4 Merkle proof queries. Your mission, should you choose to accept it, is to recover the entire flag.

4.2 Approach

We are allowed $n/4$ Merkle-proof queries. So we query one index every 4 bytes (indices 0,4,8,...). For each query we get the leaf value and a proof (list of hex hashes). Use the proof to recover nearby bytes: the last proof element is the immediate sibling leaf hash (one character), the second-last (if present) is a hash of a 2-byte subtree. Brute-force printable characters / printable pairs to match those hashes and fill the flag. Unknown bytes become '¿ and are sent as-is.

The code goes as follows.......................

Main recovery loop (one query per 4-byte block)

```
recovered = [None] * DATA_LEN

# one query per 4-byte block (allowed = DATA_LEN/4)
for idx in range(0, DATA_LEN, 4):
    val, proof = get_proof(idx)
    recovered[idx] = val if isinstance(val, int) else val[0]

    proof_bytes = [bytes.fromhex(p) for p in proof]
# immediate sibling leaf (last proof element)
# last proof hash = sibling leaf
    if proof_bytes:
        sib_hash = proof_bytes[-1]
        sib_idx = idx ^ 1
# brute-force single printable char whose hash matches
        if recovered[sib_idx] is None:
            for ch in string.printable:
                if sha256(ch.encode()).digest() == sib_hash:
                    recovered[sib_idx] = ord(ch)
                    break

# sibling subtree of size 2 (second-last proof element) -> brute-force printable pairs
# second-last proof hash = sibling subtree (2-byte pair)
    if len(proof_bytes) >= 2:
        pair_hash = proof_bytes[-2]
        base = (idx // 4) * 4
        a_pos, b_pos = base + 2, base + 3
# brute-force printable pairs to match subtree hash
        if (0 <= a_pos < DATA_LEN) and (0 <= b_pos < DATA_LEN) and (recovered[a_pos] is No
            found = False
            for a in string.printable:
                ha = sha256(a.encode()).digest()
                for b in string.printable:
                    if sha256(ha + sha256(b.encode()).digest()).digest() == pair_hash:
                        recovered[a_pos], recovered[b_pos] = ord(a), ord(b)
                        found = True
                        break
                if found:
                    break

# fill unknowns with '?' and produce final bytes
data = bytes([c if c is not None else ord('?') for c in recovered])
```

What we are doing above–

1. Create a list `recovered` of length `DATA_LEN` to store byte values (or `None`).
2. For every index `idx` in steps of 4: call `get_proof(idx)`. The server returns the value at `idx` and the Merkle proof.
3. Convert each proof element from hex to bytes (`proof_bytes`).

4. The last proof element is the sibling leaf hash. Brute-force all printable characters and match sha256(ch) to that hash. If matched, we recovered the sibling character (set `recovered[sib_idx]`).

5. If the proof has at least 2 elements, the second-last element is the hash of a 2-byte sibling subtree. We brute-force printable pairs (a, b). For each pair we compute sha256(sha256(a) $\|$ sha256(b)) and compare with that proof hash. If matched, we recovered two bytes at positions `a_pos` and `b_pos`.

6. After looping all blocks, replace any `None` with '¿ and form the final `data` bytes to send (hex-encoded).

Query one index every 4 bytes, use the proof hashes to brute-force sibling leaves and 2-byte sibling subtrees, fill recovered array, replace unknowns with "?" and submit the hex.

```
the flag --> cs409{maybe_you_don't_know_what's_lost_'til_you_find_it_merkle!}
```

Lab4 Report

Rishi Vardhan Majji
24B0969

September 2025

1 Challenge 1: Public Encryption

You are given the RSA-OAEP private and public keys of a user in pub.pem and priv.pem respectively. The flag encrypted using the user's public key is provided to you in cipher.bin. Use openssl to decrypt the ciphertext to recover the flag!

1.1 Approach

Using openssl, this qsn can be solved using the terminal.

```
openssl pkeyutl -decrypt -inkey priv.pem \
-in cipher.bin -pkeyopt \
rsa_padding_mode:oaep
```

we get the flag **cs409{r54_043p_3t_0p3n_55l}**

2 Challenge 2: Is This Certified?

2.1 Description

Who digitally certified https://www.cse.iitb.ac.in?
The flag for this challenge is the name of the issuer of the digital certificate of https://www.cse.iitb.ac.in

2.2 Approach

Open the browser, go to the required website (cse.iitb.ac.in)
And then click on the icon left to the search bar
(the one that shows site information)

Then click on the option ”Connection is secure”, then on ”certificate is valid” And then at the section issued by, you can see its common name as RapidSSL TLS RSA CA G1

hence the flag is **cs409{RapidSSL_TLS_RSA_CA_G1}**

3 Challenge 3: ECDSA Nonce Reuse

3.1 Description

A nonce should only be used once. Repeating nonce across signatures can lead to disastrous consequences. In this challenge, you'll be given two signatures generated using the same nonce.
You need to recover the nonce and in fact, the private key of the signer!

3.2 Approach

We are given two messages, m_1 and m_2, and their respective signatures, (r_1, s_1) and (r_2, s_2). We observe that $r_1 = r_2$, confirming nonce reuse. We can just use $r = r_1$.

Let $z_1 = H(m_1)$ and $z_2 = H(m_2)$. We have:

1. $s_1 = k^{-1}(z_1 + r \cdot d) \pmod n$
2. $s_2 = k^{-1}(z_2 + r \cdot d) \pmod n$

By solving for k:

$$k \cdot s_1 = z_1 + r \cdot d \pmod n$$
$$k \cdot s_2 = z_2 + r \cdot d \pmod n$$
$$\Rightarrow k(s_1 - s_2) = z_1 - z_2 \pmod n$$

From this:

$$k = (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod n$$

Once k is found, we can substitute it back into the first equation to solve for the private key d:

$$d = (k \cdot s_1 - z_1) \cdot r^{-1} \pmod n$$

We will implement these two formulas to recover both the nonce and the private key.
The code is as follows.................

First, we get the curve order n from `ecdsa.SECP256k1`.

```
n = ecdsa.SECP256k1.order
```

Next, we hash the two messages using SHA-256 and convert their hex digests to integers. These are our z_1 and z_2 values (here named `h1` and `h2`). We also set $r = r_1$.

```
h1 = int(hashlib.sha256(msg1.encode()).hexdigest(), base=16)
h2 = int(hashlib.sha256(msg2.encode()).hexdigest(), base=16)
r = r_1
```

We implement the formula for k. We compute the difference of s values and the difference of h values, modulo n.

```
# k = (h1- h2) * (s1- s2)^-1 mod n
s_diff = (s_1 - s_2) % n
h_diff = (h1 - h2) % n
```

We find the modular inverse of `s_diff` and multiply by `h_diff` to get the nonce.

```
# (s1- s2)^-1 mod n
inv_s_diff = inverse(s_diff, n)
nonce_rec = (h_diff * inv_s_diff) % n # Got the nonce
```

Now we implement the formula for d. We first find the modular inverse of r.

```
# d = (k*s1- h1) * r^-1 mod n
# r^-1 mod n
inv_r = inverse(r, n)
```

Finally, we compute the numerator $(k \cdot s_1 - h_1) \pmod{n}$ and multiply it by `inv_r` to recover the private key.

```
k_s1 = (nonce_rec * s_1) % n
k_s1_h1 = (k_s1 - h1) % n
privkey_rec = (k_s1_h1 * inv_r) % n # Got the private key
```

These two values, `nonce_rec` and `privkey_rec`, are sent to the server, which then provides the flag.

```
# flag-->cs409{n0nc3_5h0uld_b3_u53d_0nc3}
```

4 Challenge 4: EdDSA Variants

4.1 Description

One of the several digital signature schemes is EdDSA
In this challenge, you are provided with two insecure custom variants of EdDSA. Figure out how you can exploit these variants to forge signatures on arbitrary message

4.2 Approach

We will attack each variant separately to recover its private key, then use that key to sign the server's challenge. All calculations are modulo q, the order of the curve.

4.2.1 Variant 1: Deterministic Nonce

The vulnerability is that the nonce k (which we'll call `rKnown`) is not random, but is computed as $k = H(m \,||\, P_K.x)$. The signature equation is $s = k + e \cdot d \pmod{q}$, where h (which we'll call `hKnown`) is $h = H(R \,||\, P_K.x \,||\, m)$.

Since we can calculate k and h for any known signature, we can rearrange the equation to solve for the private key d (which we'll call `privkeyRec1`):

$$d = (s - k) \cdot h^{-1} \pmod{q}$$

Once we have d, we can forge a signature for the challenge message m_c by following the same rules.
The code goes as follows..........
First, we get the curve order q and take the first message and signature from the server.

```
q = ecdsa.NIST256p.generator.order()
G = ecdsa.NIST256p.generator
msgKnown = msgs[0].encode()
Rknown, Sknown = sigs[0]
```

We calculate the deterministic nonce `rKnown` (our k) for the known message.

```
rKnownHashInp = msgKnown + str(VARIANT1_PUBKEY.x()).encode()
rKnown = int(hashlib.sha256(rKnownHashInp).hexdigest(), base=16) % q
```

We calculate the challenge hash `hKnown` (h) for the known signature.

```
hKnownHashInp = str(Rknown.x()).encode() + str(VARIANT1_PUBKEY.x()).
 encode() + msgKnown
hKnown = int(hashlib.sha256(hKnownHashInp).hexdigest(), base=16) % q
```

Now we solve for the private key `privkeyRec1`.

```
hKnownInv = inverse(hKnown, q)
privkeyRec1 = ((Sknown-rKnown)*hKnownInv) % q
```

We forge the challenge signature by following the same flawed process:

```
# Calculate challenge nonce kc, point Rc, hash ec
kc_hash_inp = challenge_msg_1 + str(VARIANT1_PUBKEY.x()).encode()
kc = int(hashlib.sha256(kc_hash_inp).hexdigest(), base=16) % q
Rc = kc * G
ec_hash_inp = str(Rc.x()).encode() + str(VARIANT1_PUBKEY.x()).encode() + challenge_msg_1
ec = int(hashlib.sha256(ec_hash_inp).hexdigest(), base=16) % q

# Calculate final signature s = (kc + ec * d) mod q
s = (kc + ec * privkeyRec1) % q
R = Rc
```

4.2.2 Variant 2: Colliding Nonce via Message Prefix

The vulnerability here is that the nonce k is computed based on the private key and only the first half of the message: $k = H(\text{msg}[: \text{len}/2] \,||\, d)$. If we send two messages with the *same first half*, the server computes the same k for both, causing a nonce reuse.

This gives us $R_a = R_b$ and lets us solve for d:

$$s_a - s_b = (h_a - h_b) \cdot d \implies d = (s_a - s_b) \cdot (h_a - h_b)^{-1} \pmod q$$

With d, we can forge a signature by following the server's flawed logic.
The code is as follows...........
We get two signatures from same-prefix messages and find their hashes.

```
msgAbytes = msgs[0].encode()
msgBbytes = msgs[1].encode()
R_a , s_a = sigs[0]
R_b, s_b = sigs[1]
Rknown = R_a # R is the same

h_aHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
 encode() + msgAbytes
h_a = int(hashlib.sha256(h_aHashInp).hexdigest(), base=16) % q

h_bHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
 encode() + msgBbytes
h_b = int(hashlib.sha256(h_bHashInp).hexdigest(), base=16) % q
```

We implement the formula to recover the private key `privkeyRec2`.

```
sDiff = (s_a- s_b) % q
hDiff = (h_a- h_b) % q
hDiffInv = inverse(hDiff, q)
privkey = (sDiff * hDiffInv) % q
```

Now, we forge the signature for `challenge_msg_2` using the key.

```
# Calculate the challenge nonce 'kc' based on its prefix
msg_c = challenge_msg_2
msg_c_prefix = msg_c[:len(msg_c)//2]
kc_hash_inp = msg_c_prefix + str(privkey).encode()
kc = int(hashlib.sha256(kc_hash_inp).hexdigest(), base=16) % q

# Calculate the challenge point 'Rc'
Rforge = kc * G

# Calculate the challenge hash 'hc'
hc_hash_inp = str(Rforge.x()).encode() + str(VARIANT2_PUBKEY.x()).encode() + msg_c
hc = int(hashlib.sha256(hc_hash_inp).hexdigest(), base=16) % q
```

```
# Calculate the final signature 'sc'
sForge = (kc + hc * privkey) % q
R = Rforge
s= sForge

    # flag-->cs409{3dd54_g0t_m3_t4lk1ng_n0nc353nc3}
```

We get the flag **cs409{3dd54_g0t_m3_t4lk1ng_n0nc353nc3}**

5 Bonus Challenge: Grover's Cipher

The server's challenge is to find a 32-bit (4-byte) message. The normal way to solve this would be to **brute-force**, or guess all 2^{32} possible messages. This is way too slow and we'd run out of time.

So we use an attack called meet-in-the-middle attack that's much faster. Instead of solving one huge 32-bit problem, we'll split it into two easy 16-bit problems.

This is how it goes:

1. **Split the List:** The server gives us a list v of 32 numbers. We split this list into two halves: a `vLeft` (the first 16 numbers) and a `vRight` (the last 16 numbers).
2. **Make a dictionary for the Left Half:** We pre-calculate all 2^{16} possible answers for the `vLeft` list. We store these in our dictionary where the key is the product c_L, and the value is the 16-bit mask m_L that made it.
3. **Search the Right Half:** Now, we loop 2^{16} times, checking every possible mask m_R for the `vRight` list. For each one, we calculate its product c_R.
4. **Find the Match:** We know that $c_L \times c_R = c$ (the final answer). We can rearrange this to ask: "What c_L do I need to find in the dictionary made?" The answer is $c_L = c \times (c_R)^{-1} \pmod{pub}$. We look up this target c_L.
5. The moment we find a target c_L that is in our phone book, we've found our match! We get the `m_L` from the phone book and have the `m_R` from our loop. We just stick them together to make the final 32-bit message: $m = (\texttt{m\_L} \ll 16) | \texttt{m\_R}$.

The code goes as follows...........
First, we split the 32-element vector `v` into two 16-element halves.

```
    n = len(v)
    half_n = n // 2

    vLeft = v[:half_n]
    vRight = v[half_n:]
```

We create the lookup table `leftTable` by iterating through all 2^{16} possible masks for the left half (`m_L`) and storing their products.

```
leftTable = {}
for m_L in range(1 << half_n):
    c_L = 1
    for i in range(half_n):
        # Check the (15-i)-th bit of m_L
        if (m_L >> (half_n - 1 - i)) & 1:
            c_L = (c_L * vLeft[i]) % pub
    leftTable[c_L] = m_L
```

Now we search the right half. We iterate through all 2^{16} masks for the right half (`m_R`) and compute their products `c_R`.

```
for m_R in range(1 << half_n):
    c_R = 1
    for i in range(half_n):
        # Check the (15-i)-th bit of m_R
        if (m_R >> (half_n - 1 - i)) & 1:
            c_R = (c_R * vRight[i]) % pub
```

We check if `c_R` is invertible. If it is, we find its inverse and calculate the target `c_L_target` we need to find in our table.

```
    if gcd(c_R, pub) != 1:
        continue

    c_Rinv = pow(c_R, -1, pub)
    c_L_target = (c * c_Rinv) % pub
```

If the target exists in `leftTable`, we've found our match. We reconstruct the full 32-bit solution and break the loop.

```
    if c_L_target in leftTable:
        m_L_found = leftTable[c_L_target]

        solution_mask = (m_L_found << half_n) | m_R
        break
```

Finally, we convert the 32-bit integer mask into the 4-byte hex string the server expects.

```
n_bytes = (n + 7) // 8
message_bytes = long_to_bytes(solution_mask, n_bytes)
message = message_bytes.hex()

# flag-->cs409{4174_br34k5_unbr34k34bl3_c1ph3r5}
```