MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

# Information Security
# Lab Assigment I

Aarón Cela Riveiro

In this lab assignment, I developed an application that securely encrypts, sends and receives messages using nested hybrid encryption and onion routing over an MQTT broker. Messages are routed through multiple hops, each decrypting only one layer before forwarding it, preserving privacy and sender anonymity. The project is divided into three modules: `tor.py` for encryption logic, `mqtt-listener.py` for message relaying, and `mqtt-sender.py` for sending routed messages. An additional file, `pubkeys.py`, stores public keys of classmates for testing multi-hop communication.
Full implementation available at https://github.com/ronezz/SI/tree/main/LAB1 .

## File: `tor.py`

This file contains the core cryptographic functions of the project. Below, each group of functions is explained and shown in code blocks.

### 1. Key Management

Open my private and public keys (read from `private_key.pem` and `public_key.pub`):

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import  pubkeys


# Open private key
with open("private_key.pem", "rb") as key_file:
    private_key = serialization.load_ssh_private_key(
    key_file.read(),
    password=None,
    backend=default_backend()
    )

# Open public key
with open("public_key.pub", "rb") as key_file:
    public_key = serialization.load_ssh_public_key(
    key_file.read(),
    backend=default_backend()
)
```

Listing 1: Read RSA private and public keys

Normalize user identifiers and convert back from bytes to string:

```
# Format userId to 5b
def adjust_userId(user_id):
    b = user_id.encode('utf-8')
    b = b[:5]
    b = b.ljust(5, b'\x00')
    return b

def _bytes5_to_str(b5):
    return b5.rstrip(b'\x00').decode('utf-8', errors='ignore')
```

Listing 2: User-id helpers

Look up a recipient's public key from the shared dictionary:

```
# Find user public key
def find_pubKey(user_id):
    public_key= pubkeys.pubkey_dict.get(user_id)

    if public_key is not None:
        return serialization.load_ssh_public_key(
                ('ssh-rsa ' + public_key).encode('ascii'),
                backend=default_backend())
    else:
        print("Public key not found")

    return public_key
```

Listing 3: Find recipient public key

## 2. Hybrid Encryption

AES-GCM functions for Encrypt/Decrypt

```
# AESCG Functions
def aesgcm_encryption(key, data):
    aesgcm = AESGCM(key)
    nonce = key
    ciphertext = aesgcm.encrypt(nonce, data, None)
    return ciphertext


def aescgm_descryption(key, ciphertext):
    aesgcm = AESGCM(key)
    nonce = key
    return aesgcm.decrypt(nonce, ciphertext, None)
```

Listing 4: AES-GCM helpers

RSA-OAEP functions for encrypt/decrypt

```
# RSA Functions
def rsa_encryption(public_key, message):
    encrypted = public_key.encrypt(
        message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None)
    )
    return encrypted


```

```python
def rsa_decryption(encrypted):
    original_message = private_key.decrypt(
    encrypted,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None)
    )
    return original_message
```

Listing 5: RSA-OAEP helpers

Hybrid wrappers

```python
# Hybrid Encryption/Decryption
def hybrid_encryption(public_key, data):
    key = AESGCM.generate_key(bit_length=128)
    aescgm_cipher = aesgcm_encryption(key, data)
    rsa_cipher = rsa_encryption(public_key, key)
    return rsa_cipher + aescgm_cipher

def hybrid_decryption(ciphertext):
    key_length = private_key.key_size // 8

    cipher_key = ciphertext[:key_length]
    cipher_data = ciphertext[key_length:]

    key = rsa_decryption(cipher_key)
    data = aescgm_descryption(key, cipher_data)

    return data
```

Listing 6: Hybrid encryption/decryption

## 3. Onion Layer Construction

Build the onion

```
# Build
def nest_hybrid_encryption(path, data, anonymous):
    sender = path[0]
    recipient = path[-1]

    # Check if we want to send the message anonymously
    sender_field = "none" if anonymous else sender

    final_content = adjust_userId("end") + adjust_userId(sender_field) + data
    ciphertext = hybrid_encryption(find_pubKey(recipient), final_content)
    for index in range(len(path) - 2, -1, -1):
        current_hop = path[index]
        next_hop = path[index + 1]
        payload_for_hop = adjust_userId(next_hop) + ciphertext
        ciphertext = hybrid_encryption(find_pubKey(current_hop),
            payload_for_hop)

    return ciphertext
```

Listing 7: Nested onion construction

## Decryption and Relay

**8-** Peel the onion

```
# Peel
def decode_and_relay(ciphertext):

    payload = hybrid_decryption(ciphertext)
    next_hop = payload[:5]
    rest = payload[5:]

    nh = _bytes5_to_str(next_hop).lower()
    if nh != "end":
        return ("forward", _bytes5_to_str(next_hop), rest)
    else:
        sender = _bytes5_to_str(rest[:5])
        message = rest[5:]
        return ("deliver", sender, message)
```

Listing 8: Decode one layer and relay/deliver

## File: mqtt_listener.py

This file is responsible for receiving and forwarding encrypted messages through the MQTT broker.

### 1. MQTT Configuration

The MQTT server parameters are defined here, as well as the identity of the current node, which is used to subscribe to inbound messages in its topic.

```
from paho.mqtt import client as mqtt_client
import tor

MQTT_SERVER = "18.101.140.151"
MQTT_USER = "sinf"
MQTT_PASSWORD = "sinf2025"
MQTT_PORT = 1883
MQTT_KEEPALIVE = 60

MY_ID = "ancr"
```

Listing 9: MQTT connection parameters

### 2. Message Callback Handler

This function is triggered automatically when a message is received on the MQTT topic associated with this node. It decodes one onion layer using `decode_and_relay()` from `tor.py`.

```
# While Listening...
def on_message(client, userdata, msg):
    try:
        action, who, payload = tor.decode_and_relay(msg.payload)
        if action == "forward":
            print(f"[{MY_ID}] Forwarding next hop '{who}'...")
            client.publish(who, payload)
        else:
            try:
                texto = payload.decode("utf-8")
            except:
                texto = repr(payload)
            print(f"[{MY_ID}] Message from '{who}': {texto}")
    except Exception as e:
        print(f"[{MY_ID}] Error processing message: {e}")
```

Listing 10: Handle incoming message

### 3. MQTT Loop Initialization

This function connects to the MQTT broker, subscribes to the node's topic (its user-id), and waits indefinitely for incoming messages.

```
def main():
    c = mqtt_client.Client()
    c.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    c.on_message = on_message
    c.connect(MQTT_SERVER, MQTT_PORT, MQTT_KEEPALIVE)
    c.subscribe(MY_ID)
    print(f"[{MY_ID}] Listening on topic '{MY_ID}'...")
    c.loop_forever()

if __name__ == "__main__":
    main()
```

Listing 11: Main MQTT loop

## File: mqtt_sender.py

This file is responsible for sending encrypted messages using nested hybrid encryption over MQTT. It uses the function `nest_hybrid_encryption()` from `tor.py` to wrap the message with multiple encryption layers before publishing it to the MQTT broker.

### 1. MQTT Configuration and Client Creation

The MQTT server credentials are defined here. A helper function is used to initialize and connect the MQTT client.

```python
from paho.mqtt import client as paho_mqtt
import tor

MQTT_SERVER = "18.101.140.151"
MQTT_USER = "sinf"
MQTT_PASSWORD = "sinf2025"
MQTT_PORT = 1883
MQTT_TOPIC = "ancr"
MQTT_KEEPALIVE = 60

def mqtt_client(server, port, topic, user, password, keepalive):
    client = paho_mqtt.Client()
    client.username_pw_set(user, password)
    client.connect(server, port, keepalive)
    client.subscribe(topic)
    return client


# Create MQTT Client
client = mqtt_client(MQTT_SERVER, MQTT_PORT, MQTT_TOPIC, MQTT_USER,
    MQTT_PASSWORD, MQTT_KEEPALIVE)
```

Listing 12: MQTT setup and client creation

### 2. Encrypt and Send a Message

A simple test message is encrypted using nested onion encryption. The ciphertext is then published to the first hop in the route.

```python
m = b"Testing the network..."

path=["ancr","ancr","ancr","ancr","ancr"]

encrypted_to_send = tor.nest_hybrid_encryption(path, m, False)
result = client.publish("ancr", encrypted_to_send)
```

Listing 13: Encrypt and publish nested ciphertext

# Tests and Validation

This section presents a series of tests carried out to verify the correct operation of the implemented system, including nested hybrid encryption, onion-routing behaviour, and MQTT message delivery across multiple hops. The objective is to demonstrate that ciphertexts are successfully routed through one or more intermediate nodes and correctly decrypted only at the final destination.

## Test 1: Self-routing with multiple hops

In this first test, a message was sent from my user `ancr` to myself using a route with multiple self-intermediate hops.

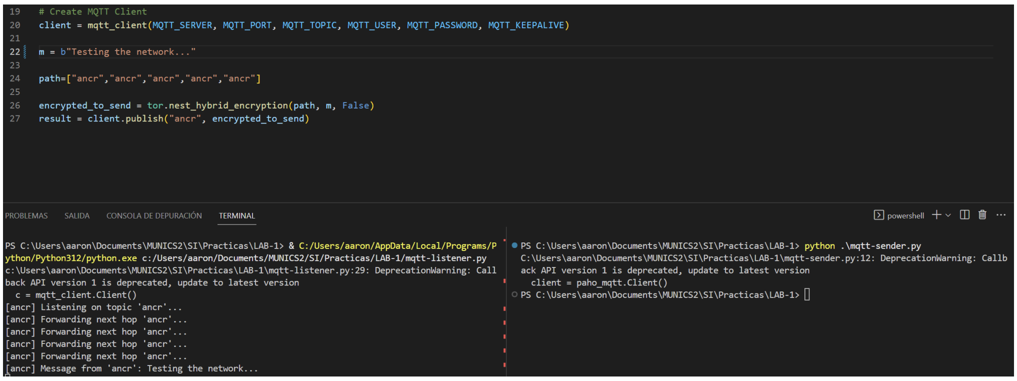


Figure 1: Self-message routing through multiple nested hops

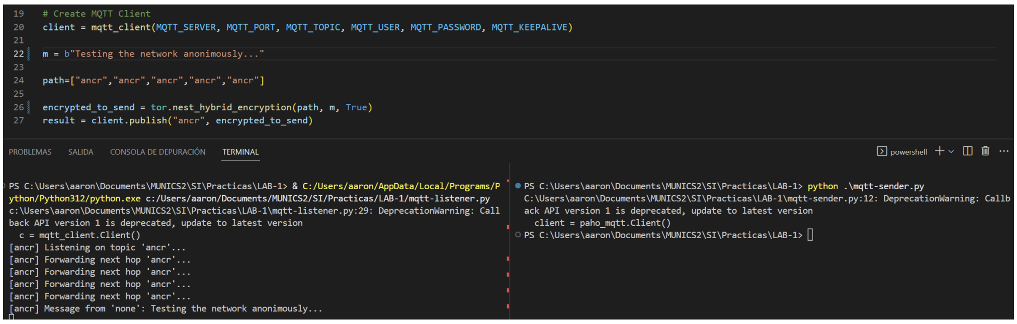


Figure 2: Anonymous self-message routing through multiple nested hops

### Test 2: Long message delivery

This second test evaluates the system's ability to handle large messages exceeding a single RSA block. Thanks to the implemented hybrid encryption (RSA + AES-GCM), messages of arbitrary length can be securely transmitted.

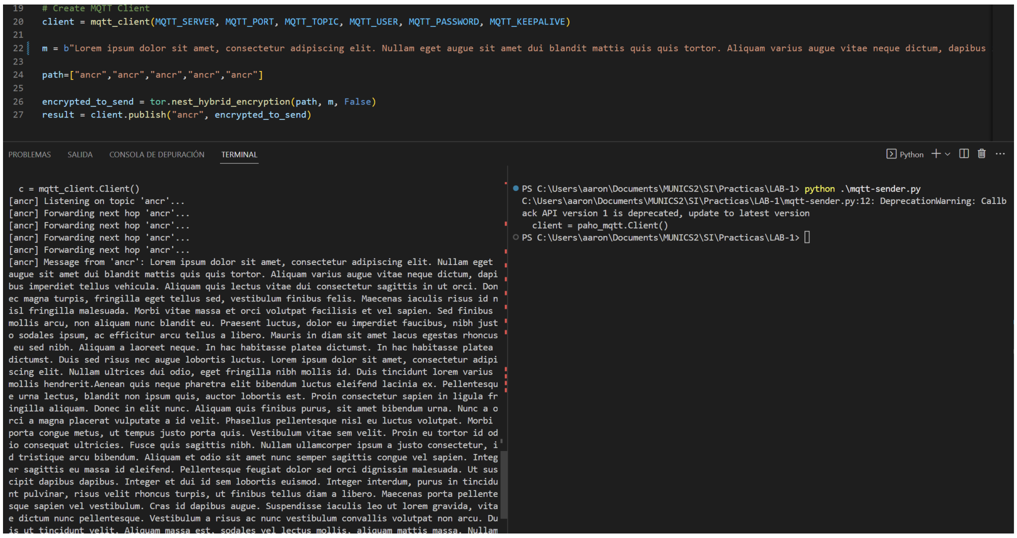

Figure 3: Successful routing of a large self-message through multiple nodes

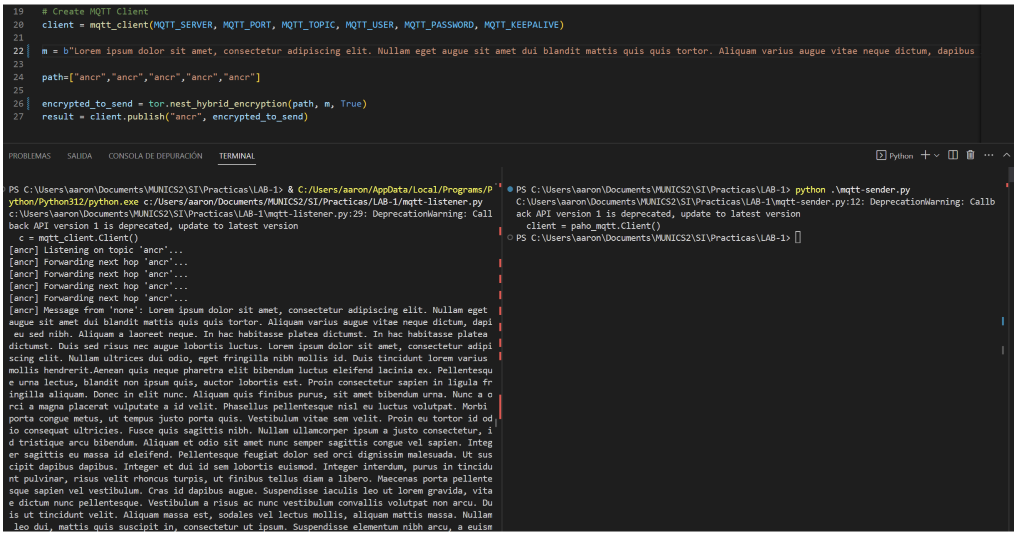

Figure 4: Anonymous routing of a large self-message through multiple nodes

### Test 3: Routing through classmates

In the final test, a route was configured including two classmates, `svr` and `bge`, as intermediate relays.

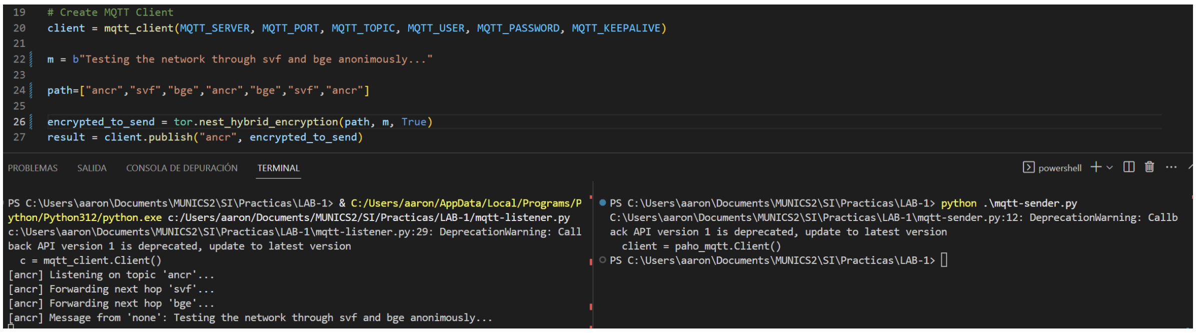


Figure 5: Anonymous message routed through classmates `svr` and `bge`

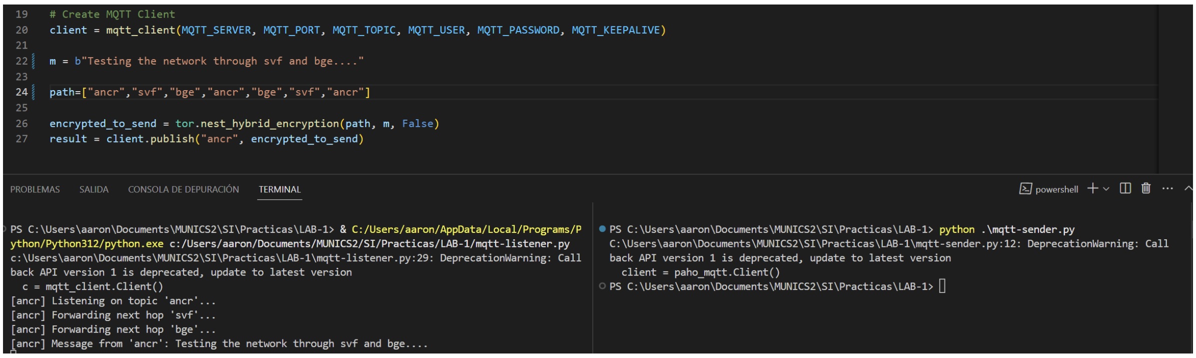


Figure 6: Standard message routed through classmates `svr` and `bge`