National College of Ireland

Configuration Manual

MSc Research Project
Cyber Security

Sunaina Amin
Student ID: X23322683

School of Computing
National College of Ireland

Supervisor: Liam McCabe

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Sunaina Amin

Student ID: X23322683

Programme: Master of Science in Cyber Security **Year:** 2024_2025

Module: Research Project

Lecturer: Liam McCabe

Submission Due Date: 11TH August 2025

Project Title: Secure Smart Home Using ECC

Word Count: **3153** **Page Count: 46**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Sunaina Amin

Date: 11th August 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple	□

copies).	
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Table of Figures

Configuration Manual

Secure Smart Home Using ECC

1. Environment setup

1.1 Installing and checking Python 3 on Mac

To serve the needs of the cryptographic scripts and the MQTT-based simulation of the smart home, Python 3 was installed through the macOS terminal, where a more modern version of Python must be had. The Mac is normally equipped with the 2.x version of Python, so the new version needed to be manually installed and configured.

Step 1: Open Terminal

The terminal has been opened through either Spotlight or through the Utilities > Terminal in Applications.

Step 2: Python Version check

bash

python3 --version

Output Python 3.12.8

This establishes that Python 3 has already been installed or successfully added through Homebrew or an official Python package.

All project scripts were executed under Python 3.12.8 in the .py format for both operations of the script: the RSA and ECC cryptographic operations.

1.2 Installing Necessary Python Libraries

To add cryptographic functions and messaging with the MQTT protocol, the following Python packages are installed:

Command Use

bash

pip3 install paho-mqtt ecdsa

Libraries Installed:

paho-mqtt: It is used when the Publisher and the Subscriber are communicating through the MQTT protocol.

ecdsa: Provides Elliptic Curve Digital Signature Algorithm (ECDSA) operations to ECC authentication.

six: A support library accessed by ecdsa.

The terminal result proves that all the required libraries have been installed fine, including ecdsa==0.19.1 and paho-mqtt==2.1.0.

```
Last login: Wed Jun 18 11:34:30 on console
(base) sunainaamin@sunainas-Air ~ %
(base) sunainaamin@sunainas-Air ~ % python3 --version

Python 3.12.8
(base) sunainaamin@sunainas-Air ~ % pip3 install paho-mqtt ecdsa

Collecting paho-mqtt
  Downloading paho_mqtt-2.1.0-py3-none-any.whl.metadata (23 kB)
Collecting ecdsa
  Downloading ecdsa-0.19.1-py2.py3-none-any.whl.metadata (29 kB)
Collecting six>=1.9.0 (from ecdsa)
  Downloading six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Downloading paho_mqtt-2.1.0-py3-none-any.whl (67 kB)
Downloading ecdsa-0.19.1-py2.py3-none-any.whl (150 kB)
Downloading six-1.17.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, paho-mqtt, ecdsa
Successfully installed ecdsa-0.19.1 paho-mqtt-2.1.0 six-1.17.0
(base) sunainaamin@sunainas-Air ~ %
```

Figure 1 python setup on macOS terminal

4 MQTT Broker Configuration Mosquitto Installation (macOS)

Installed via Homebrew package manager the Mosquitto MQTT broker was installed to facilitate the message-based intercommunication between IoT devices (publisher and subscriber).

Mosquitto is the main component to handle the communication protocol (MQTT) which is utilized in this smart home security project.

Step-by-Step Setup:

Step 1: Open Terminal and Mosquitto Install

bash

brew install mosquitto

Output Summary:

Mosquitto has an installed and default configuration file.

The configuration file is here:

/opt/homebrew/etc/mosquitto/mosquitto.conf

Step 2: Initiating Mosquitto Service By Hand

You may run the MQTT broker with:

bash

mosquitto /opt/homebrew/sbin/mosquitto -c /opt/homebrew/etc/mosquitto/mosquitto.conf

This initiates the Mosquitto broker on local port 1883, on which the Python publisher and subscriber clients can exchange authenticated messages using MQTT.

Why Mosquitto?

Mosquitto was selected because it is lightweight, provides native support on macOS, and supports paho-mqtt, the library on which Python scripts implement ECC and RSA authentication.

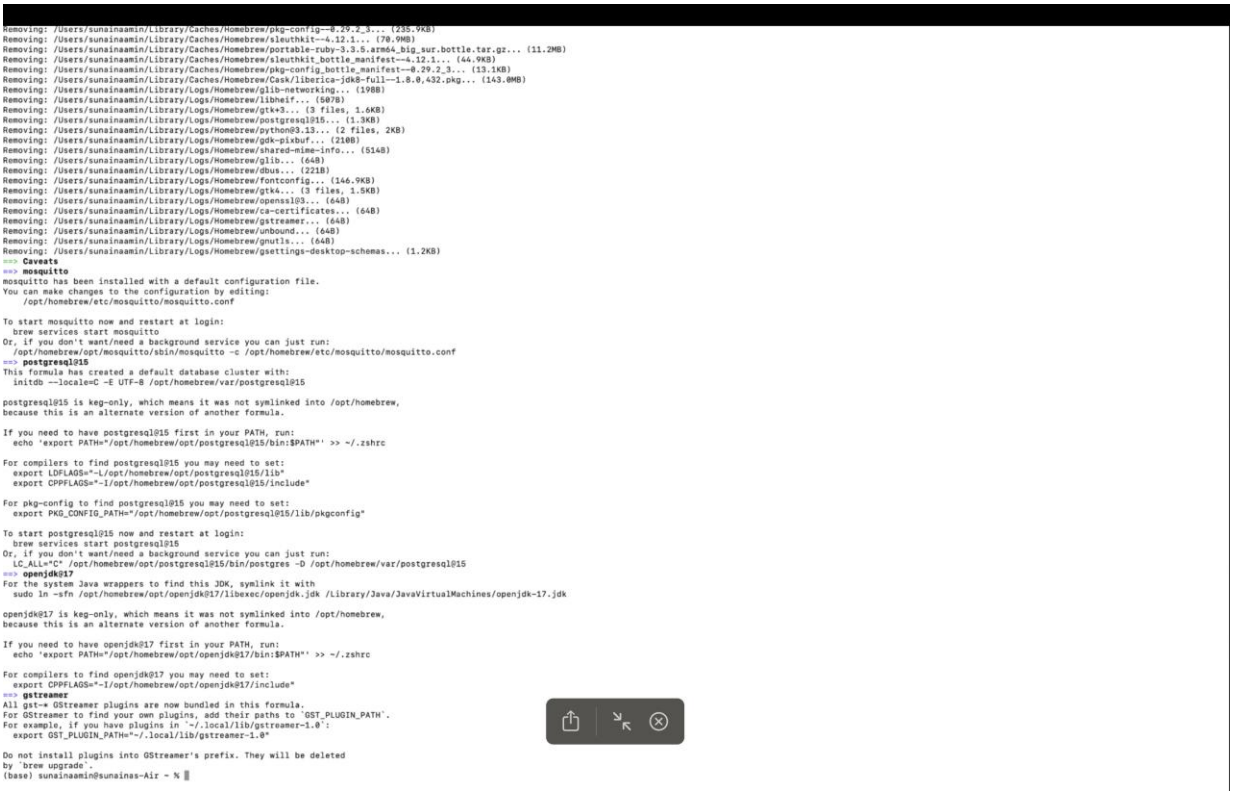

Figure 2 MQTT setup

Step 3: Installation of Visual Studio Code (VS Code) on macOS

3.1 Visual Studio Code installation

Fetched from the official site:

https://code.visualstudio.com

Drag and drop installation (Drag and drop the .app file into the Applications folder, typical Macintosh installation).

3.2 Startups of the VS Code

VS Code opened through Launchpad or Spotlight.

Setup successful, confirmed in the welcome screen.

A file folder with a project (e.g., SmartHomeProject).

Install the Python extension to highlight syntax and run debugging

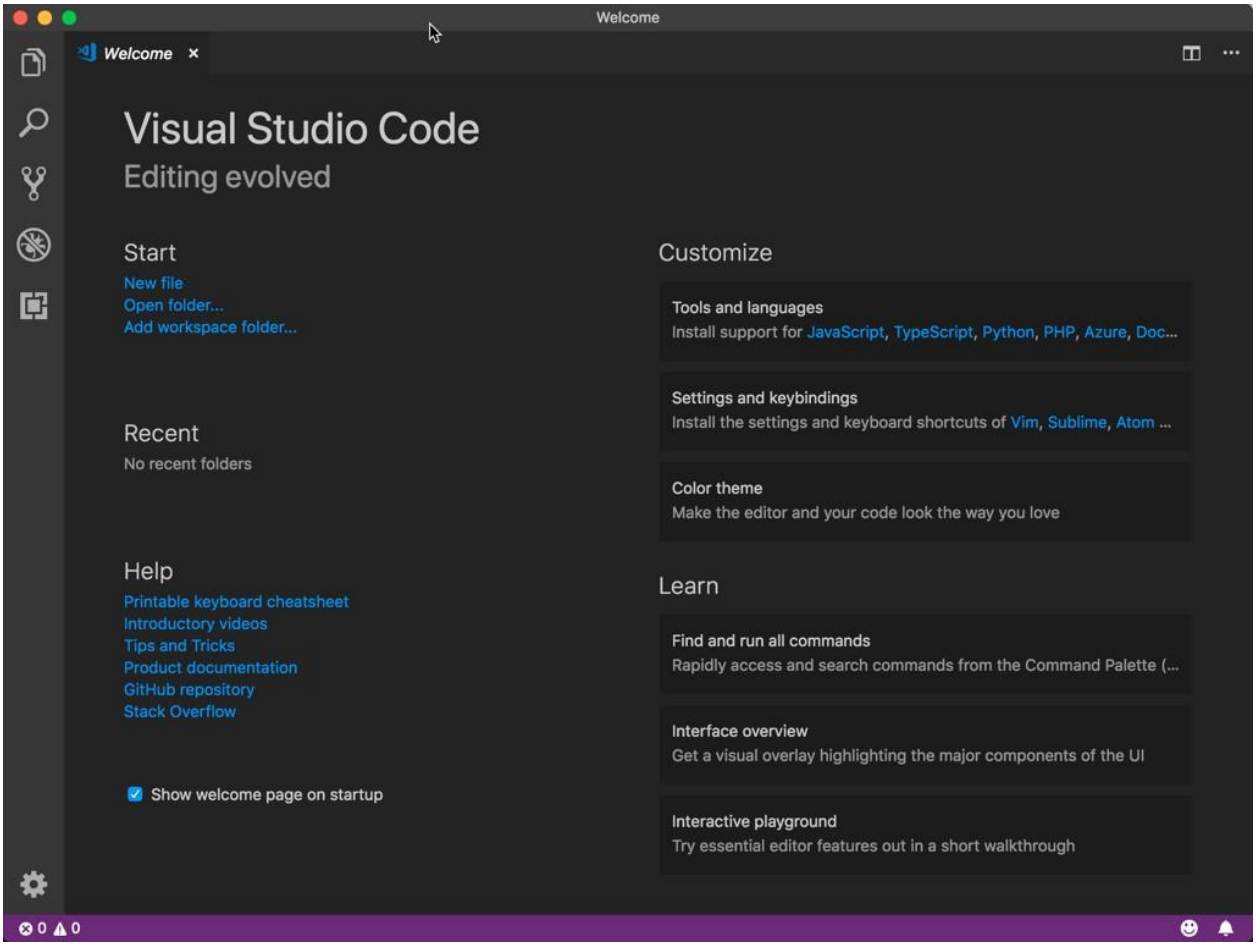

Figure 3 VS Code Installation

Step 4: Key Generation

Objective:

In the generation of key pairs in RSA and Elliptic Curve cryptography (ECC) algorithms, both are considered very basic to achieving message authentication in the Smart Home set-up.

Tools Used:

Python 3.12.8

VS Code

ECC key generation library (ECDSA)

Cryptography library (RSA key gen)

pem files (saving and referencing keys, generating)

Step-by-Step Process:

1. ECC Generation Key

The key pair generated was the ECC on the ecdsa Python library, with the SECP256k1 curve.

This was done by storing the private key in a file called privatekey.pem.pem.

I saved the corresponding public key in public_key.pem.pem.

Script Used: generate_keys.py

Output Files:

private_key.pem (ECC Private key)

public_key.pem (ECC public key)

2. RSA Key Generation

The Cryptography library was used to generate RSA key pairs (producing each key 2048 bits).

The special key was stored at rsa_private_key.pem.pem

The matching public key was stored in rsa_public_key.pem.pem.

Output Files:

rsa_private_key.pem (RSA Private Key)

rsa_public_key.pem (RSA public key)

Summary:

Successful generation of both ECC and RSA key pairs and saving in .pem format to be used subsequently in the authentication scripts was done. They are the keys needed in signing and authenticating messages between subscriber modules and publisher modules.

private_key.pem

```
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIAwHjHbEy4pS+umq5vddFgoH8GIuq8JUScdGk6dfUSBkoAcGBSuBBAAKoUQ
pEq89EroZOJnJGXdDRKyMW7rsZaCU0t8i/TTPT/UqqgiPSQaV2A8i8Jb28EK1+JpwI+
JM2PxQ==
-----END EC PRIVATE KEY-----
```

Figure 4 ECC private key

public_key.pem

```
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEDGpjpEq89EroZOJnJGXdDRKyMW7rsZaCU0t
qqgiPSQaV2A8i8Jb28EK1+JpwI+40snYr8GkJM2PxQ==
-----END PUBLIC KEY-----
```

Figure 5 ECC public Key

rsa_public_key.pem

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2OoQ8UhAJFZ4jR7xLX1/
qxdwnXUO0q11F4kQFARIhcwNu5EnPkgP8TgUcc3PO1r1/U1gvNTIZkEdICYlf3/e
VITG9sBuwPjEDJyqSAycOfN4whDVCS9oxS+gmCu0GsZQPJogDXRubiNEjR5DpAOZ
VBfhS53ayC5UuBRgRZDdSx/6KFD+UsctyY+3VvTholH28whDP+9af1cGPnYkZgeD
5k6ZMXdepZdG+zIXa0shKRFaC7XDbYMLfW1eVfmj3uAzwdThWX6E4P22wdwxgi2S
lsruqMk56vpyALexNpie6tA4bkrMXYVIXx4SZApMgJjLXaYyVTnl50Fv4Z0fcuLC
FQIDAQAB
-----END PUBLIC KEY-----
```

Figure 6: RSA public key

Figure 7: RSA private key

generate_keys.py > ...

```python
from ecdsa import SigningKey, SECP256k1

# Generate ECC private key
sk = SigningKey.generate(curve=SECP256k1)

# Get the corresponding public key
vk = sk.get_verifying_key()

# Save private key to file
with open("private_key.pem", "wb") as f:
    f.write(sk.to_pem())

# Save public key to file
with open("public_key.pem", "wb") as f:
    f.write(vk.to_pem())

print("ECC keys generated and saved to 'private_key.pem' and 'public_key.pem'")

```

Figure 8: Key Generation Setup

Smart Home MQTT setup

The script shows the authentication of the message that is ECC-based in a smart home MQTT environment. The commands (e.g., Turn on Smart Lock) signed with a secret ECC key via the publisher are then sent by the publisher to an MQTT topic, and the message is delivered along with a signature. This provides integrity and authenticity in front of the occurrence to a gadget.

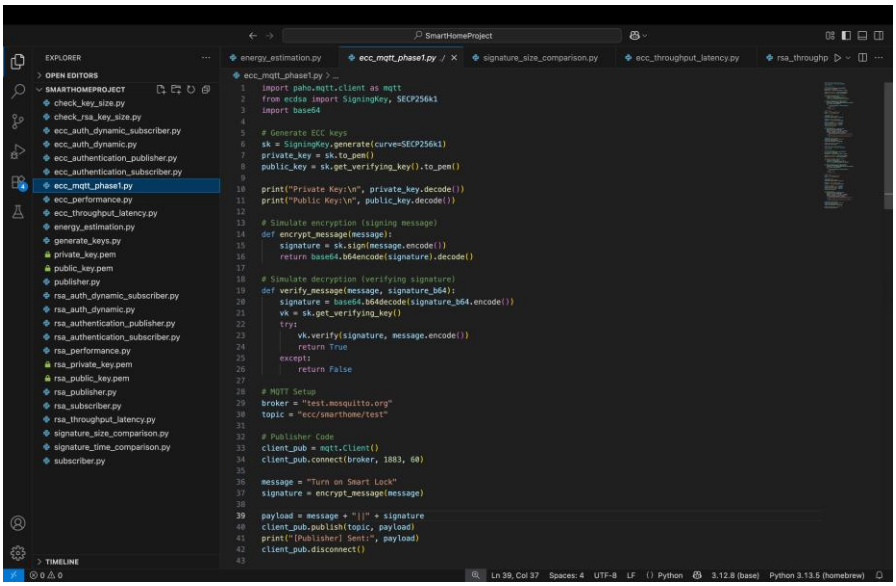

Figure 9: Dynamic Key Generation

To make the authentication process more secure, a dynamic authentication necessitates the use of ephemeral (perishable) key pairs generated on the fly. These keys are consumed after every session and replaced after a single use so that they cannot be reused to decrypt old messages, establishing forward secrecy and reducing the chances of a key leak.

In ecc_auth_dynamic_publisher.py and ecc_auth_dynamic_subscriber.py, each time around the message authentication loop, a fresh ECC key pair is created.

Equally, **in rsa_auth_dynamic_publisher.py and rsa_auth_dynamic_subscriber.py,** new RSA keys are created and used to sign and verify the messages only once.

This changing temporary key construct, even with the compromise of a single key, will never allow the recovery of previous or forthcoming sessions.

What are Dynamic Keys?

Ephemeral keys are short-lived cryptographic keys created on a session or message basis and are subsequently discarded. They strengthen security by avoiding the reuse of keys. To facilitate flawless forward secrecy and increase the system's resiliency against a key compromise attack or replay attack.

The Way i Used the Ephemeral Keys in my ECC System

In your implementation with the ECDSA Python library:

File: **ecc_auth_dynamic.py** (Publisher) In each run of the publisher script, A fresh pair of ECC keys is obtained:

python

```
private_key = SigningKey.generate(curve=SECP256k1)
public_key = private_key.verifying_key
```

These two pairs are utilized just once to encrypt the delivered message (eg, "Turn on Lights").

The message payload is encoded, and together with the public key, sent to the subscriber:

python

```
payload = {
    "message": message,
    "signature": signature_b64,
    "public_key": public_key_pem
```

}

File: ecc_auth_dynamic_subscriber.py (Subscriber)

The subscriber:

Accepts the payload that the publisher sends, which is the public key.

Checks the received message with the help of this key:

python

public_key = VerifyingKey.import_pem(public_key_pem)

public_key.verify(signature, message_str.encode, hashfunc=hashlib.sha256)

And gives up the key, then that once it is proved, it cannot be reused.

Ephemeral Key-Based Authentication Set up (ECC)

Script Files:

ecc_auth_dynamic.py (Publisher script)

Subscriber script - ecc_auth_dynamic_subscriber.py

Description:

Such an arrangement employs ephemeral keys, i.e., a different set of ECC key pairs is dynamically created each time the message is to be sent. This improves security to the extent that every session is individually secured, and replay and impersonation attacks are harder.

Step 1: On the publisher side, ephemeral Key Generation

In the ecc auth dynamic.py script:

There is a new ECC key pair with:

python

bob = SigningKey.generate(curve=SECP256k1)

publickey = privkey.get_verifying_key()

Within the message payload, there is:

Device ID

Timestamp

Command (e.g., Turn on Lights)

A payload is signed with the newly generated ephemeral private key in Base64:

python

signature = private_key.sign(message_str.encode(), hashfunc=hashlib.sha256)

Public key is also coded and carried together with the payload so that the subscriber may ensure the status of the message.

To execute the publisher:

bash

python3 ecc_auth_dynamic. py

Terminal Output:

bash

ECC Demo: The new key pair is sent with the message.

Step 2: Listening & Verification of Subscribers

At the ecc_auth_dynamic_subscriber.py script:

The subscriber can hear the topic ecc/auth/dynamic:

python

client.subscribe("ecc/auth/dynamic")

Upon getting the message:

Derives the message, signature, and the public key information in the payload.

Checks that a replay attack is prevented with a timestamp condition:

python

and abs(time.time() - message["timestamp"]) > 5:

print(" Rejected: The message is too old.")

Reconstructs the message and uses the ephemeral public key to authenticate the signature:

python

public_key.verfiy(signature, message_str.encode(), hashfunc=hashlib.sha256)

To execute the subscriber:

Write in Terminal

python3 ecc_auth_dynamic_subscriber.py

Output

ECC Dynamic Subscriber is listening...

Auth Success in 0.005839s

Command: Turn on Lights

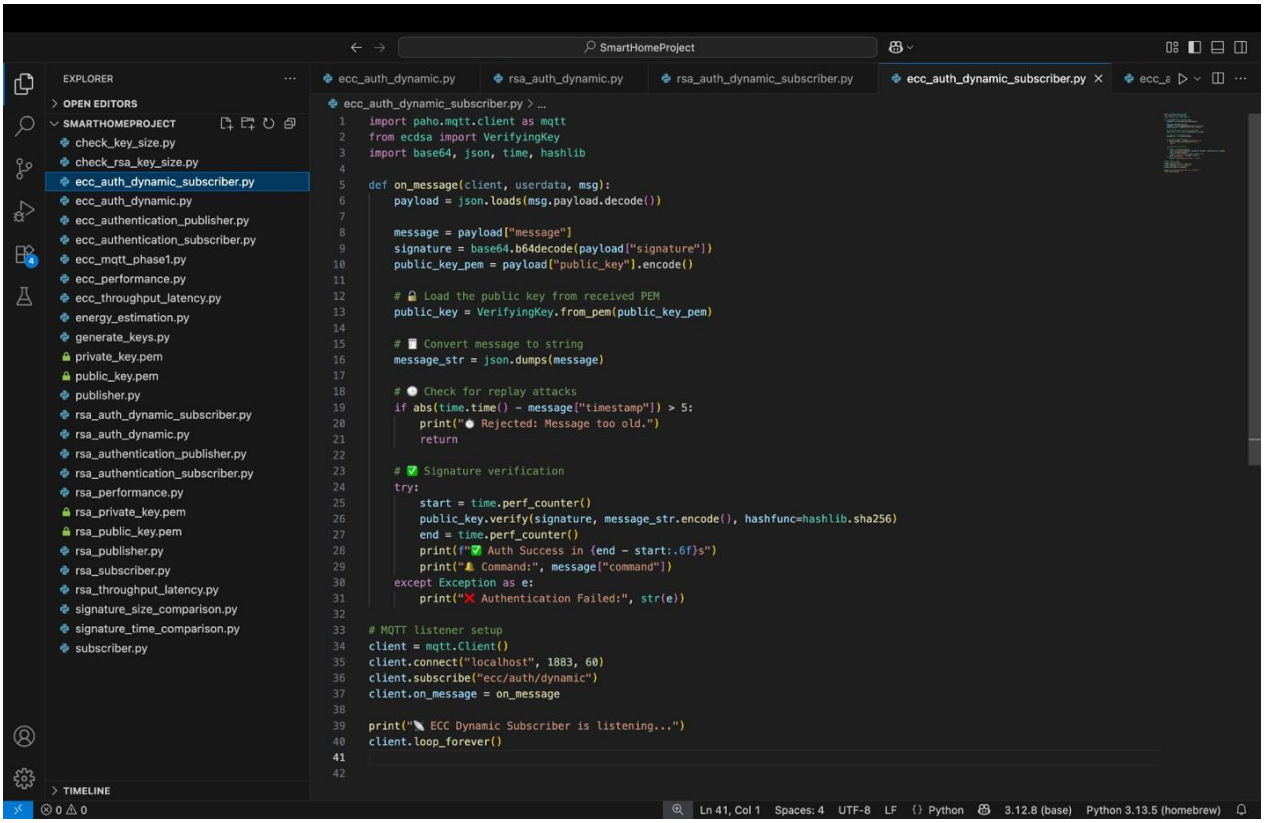

Figure 10: ECC Dynamic subscriber.py

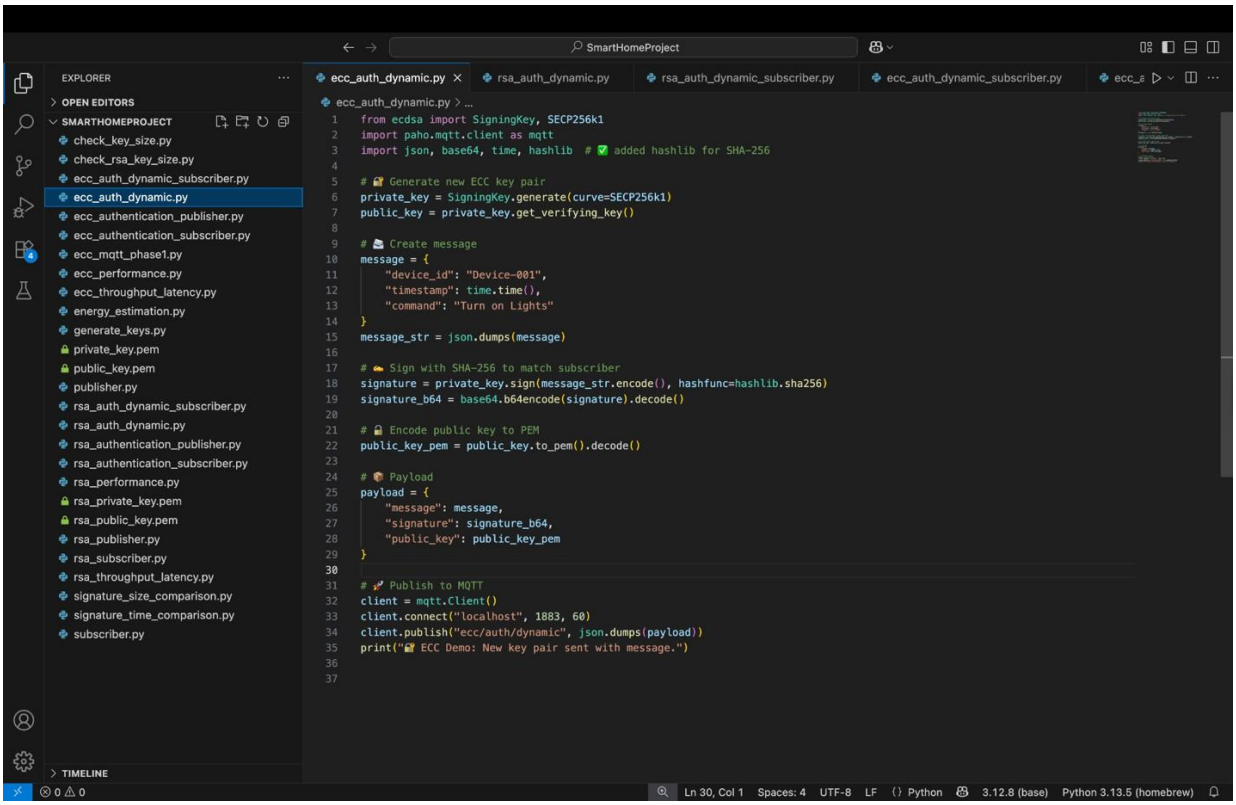

Figure 11: ECC dynamic Publisher.py

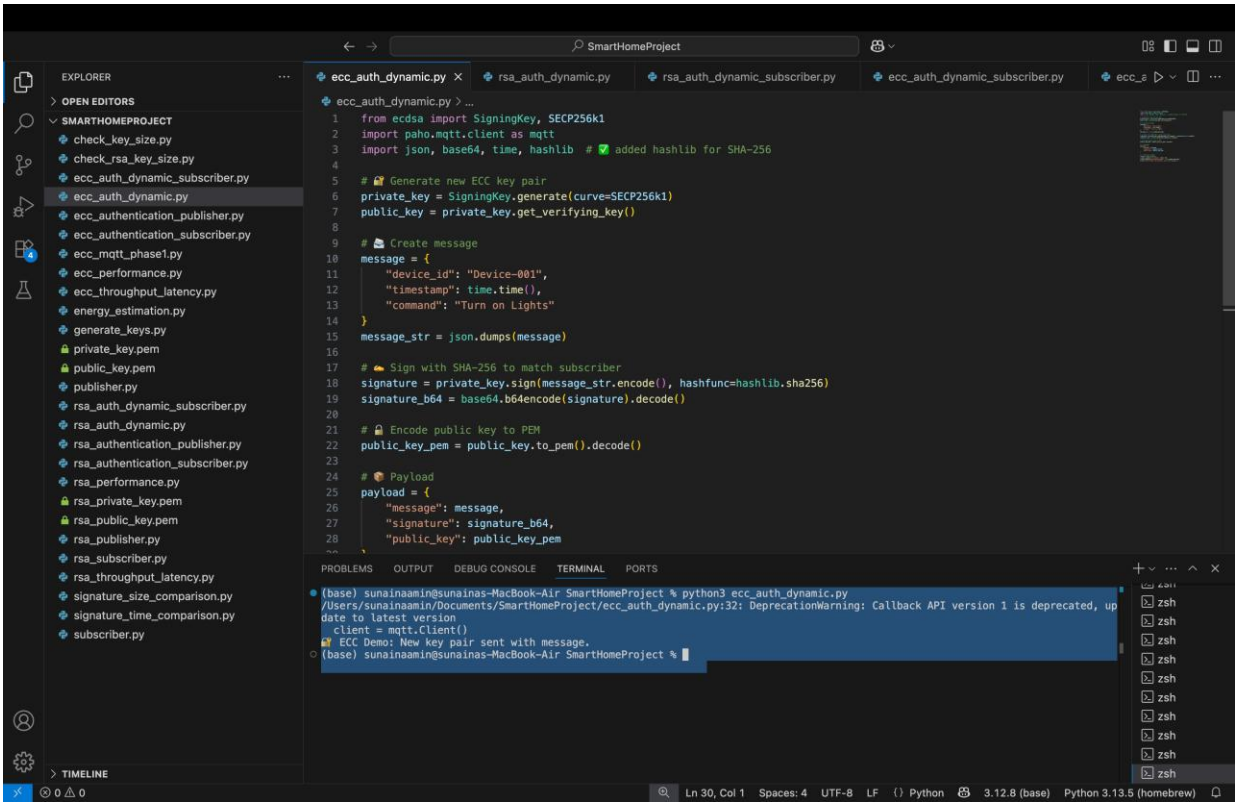

Figure 12: ECC Dynamic Publisher output

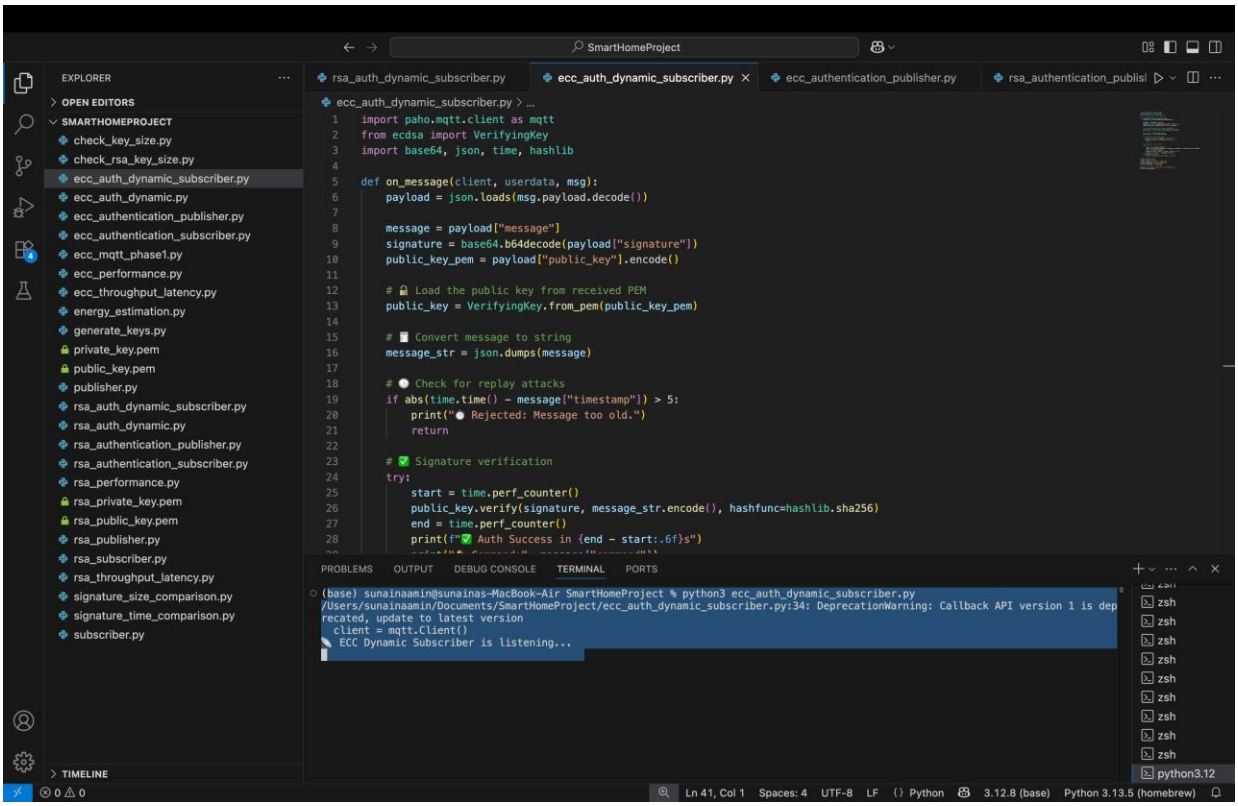

Figure 13: ECC dynamic subscriber output

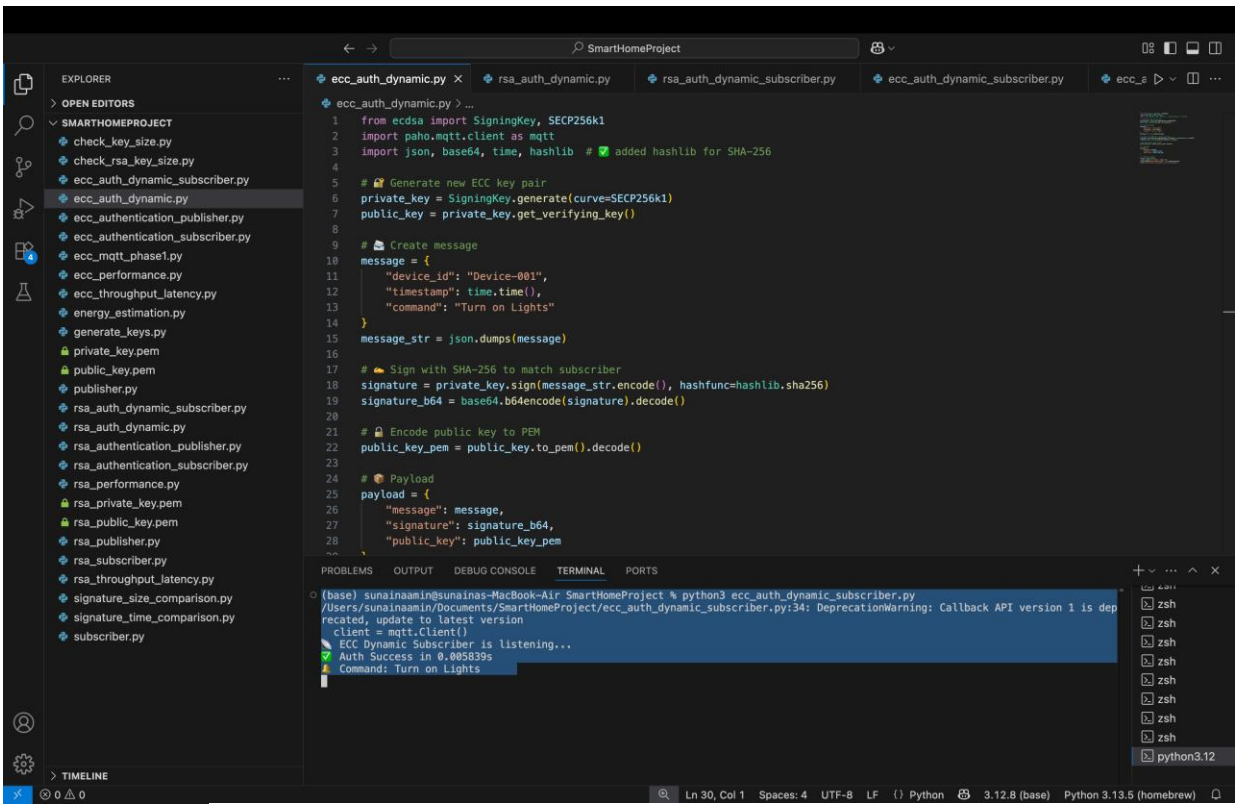

Figure 14: Authentication Successful

Dynamic RSA Key Set Up

In this section, details on how the dynamic RSA key-based authentication between Internet of Things devices is configured are explained using the publisher-subscriber MQTT communication model**.**

The following script files are used:

rsa_auth_dynamic.py RSA Dynamic Publisher

rsa_auth_dynamic_subscriber.py RSA Dynamic Subscriber

Objective:

To dynamically create RSA key pairs per session and sign and verify the authentication message using them to improve security.

Step-by-Step Configuration:

1. Start the RSA Dynamic Publisher

Start the publisher using the terminal command:

Write in Terminal

python3 rsa_auth_dynamic.py

This script on every execution, it dynamically creates a new RSA key pair using

python

private_key = rsa.gen_private_key(public_exponent=65537, key_size=2048)

Infrastructures: a message with:

Device ID

Timestamp

Action command (e.g., Turn on Lights)

Sign the message using an RSA newly generated key that is its private key.

User can send the signature and a public key in PEM format in the form of a JSON payload using MQTT.

Console output:

RSA Demo: Released with new key.

Start the RSA Dynamic Subscriber

With the use of the command:

Write in Terminal

python3 rsa_auth_subscriber_subscriber.py

This script subscribes to an MQTT topic that it listens to.

Receives signed payload.

Decrypts the base64 coded public key and signature.

Would check the signature through

python

public_key.verify(signature, message_str, padding.PSS(...), hashes.SHA256())

Does a replay attack check by checking whether the timestamp difference is less than 5 seconds.

Has the verification result and the command been printed

Console Output:

RSA Dynamic Subscriber listening...

Auth Success in 0.001585s

Command: Turn on Lights

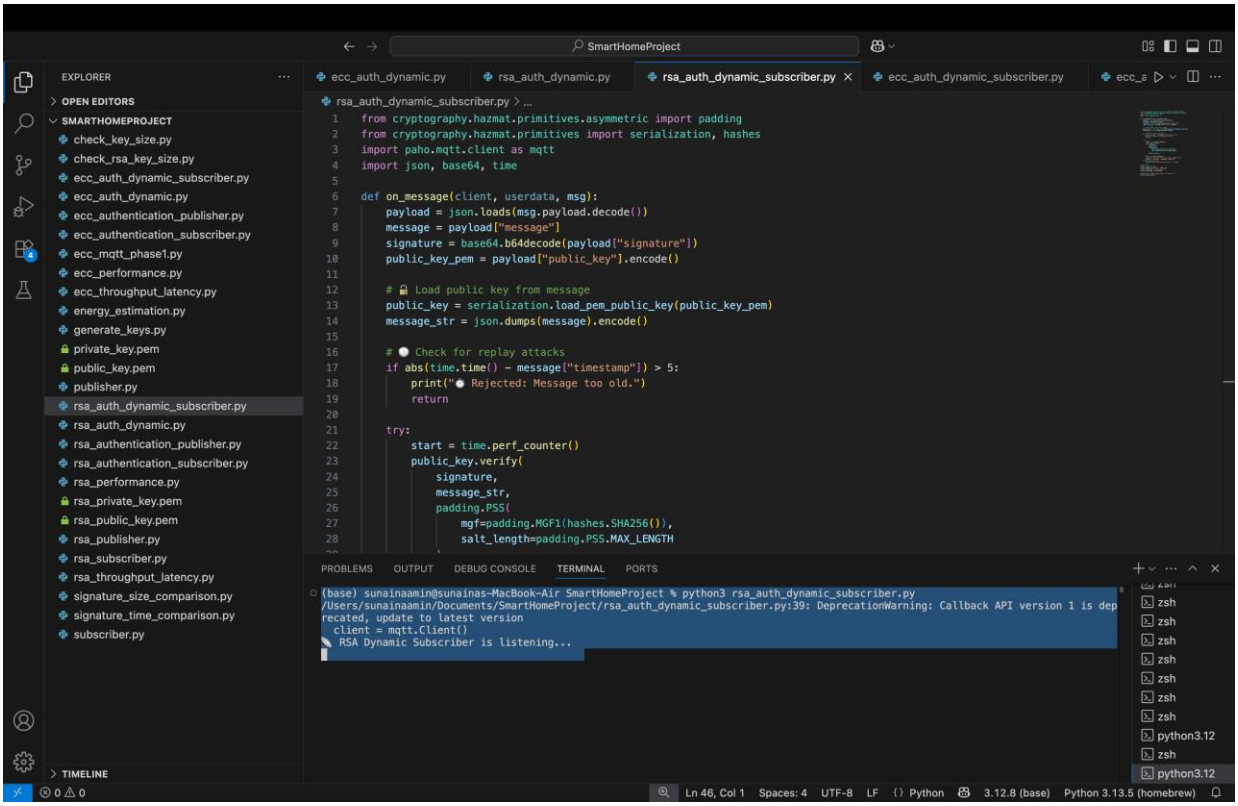

Figure 15: RSA Dynamic Subscriber

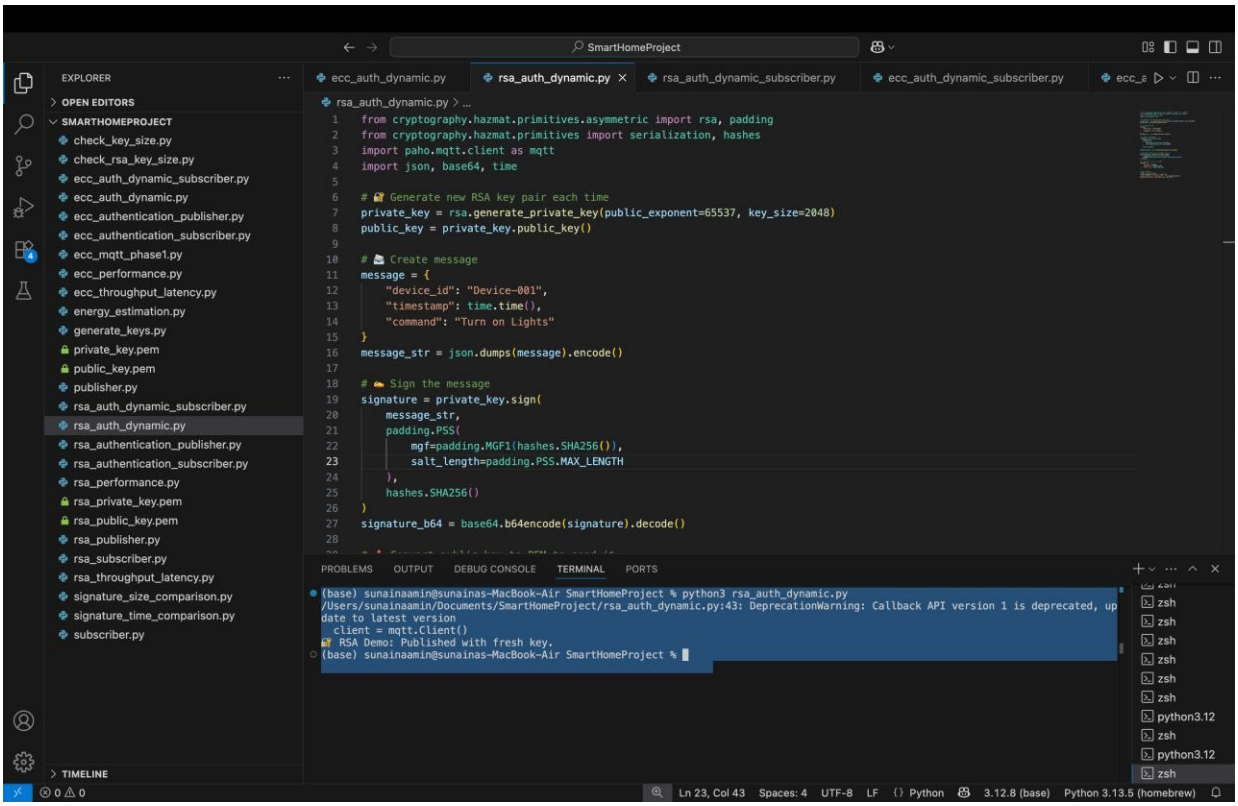

Figure 16: RSA dynamic Publisher

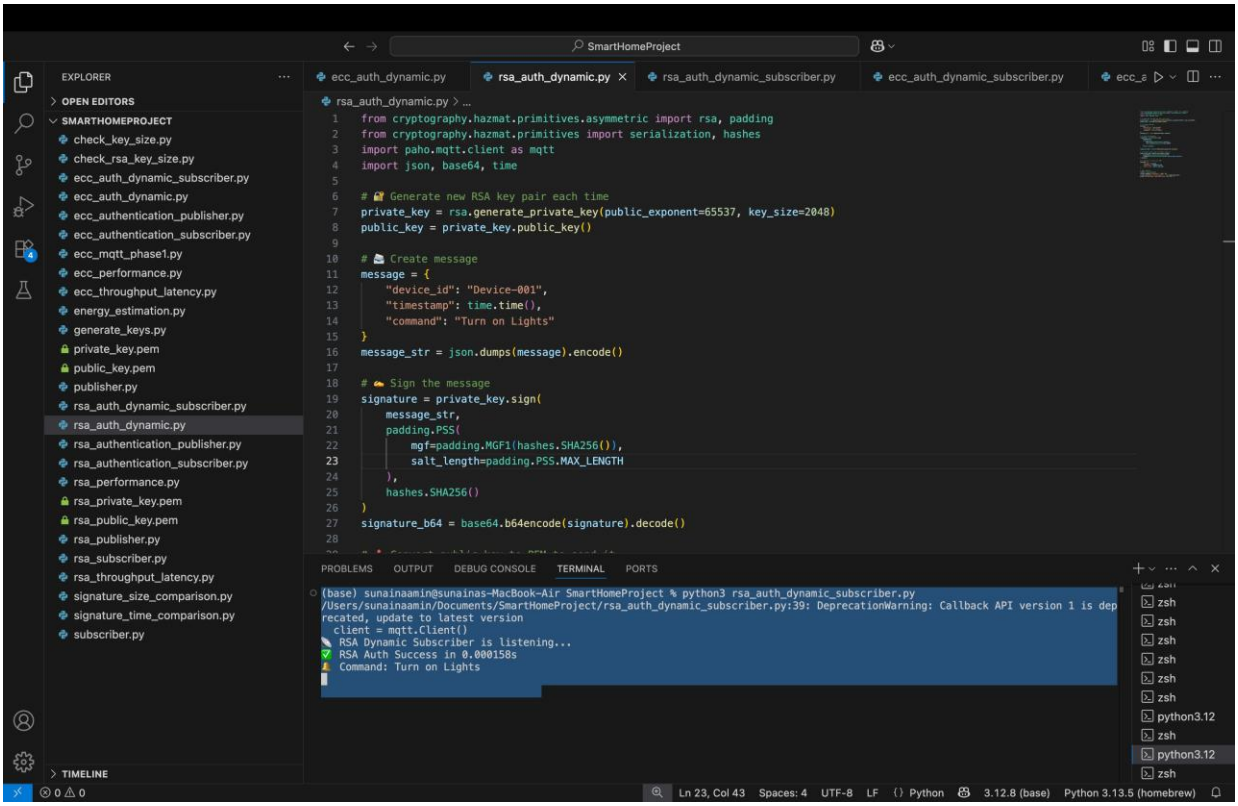

Figure 17: RSA dynamic key authentication successful

Figure 14: RSA Dynamic Key Authentication Success

- **ECC and RSA Authentication with the use of saved keys**

2. Publisher Setup (publisher.py)

Executed using:

Write in Terminal

python3 publisher.py

The script it uses the priv_key.pem for the private key.

Generates a secure message including a timestamp, device_id, and command.

Derive an ECC-based signature and base64 encode the message signature**.**

Transfers the signed message via MQTT.

Terminal Output:

Public Key, Loaded Private Key Loaded

[ECC Publisher] Sent secure message...

3. Subscriber config (subscriber.py)

Executed using:

Write in Terminal

python3 subscriber.py

Does read in public_key.pem.

ecc/smarthome/auth Listens to the file. Authenticates the signature with the ECC public key.

Terminal Output:

[ECC Subscriber] Listening…

[ECC Subscriber] Verification of a signature took in 0.008588 seconds

[ECC Subscriber] Successful authentication

Command: Turn on Lights

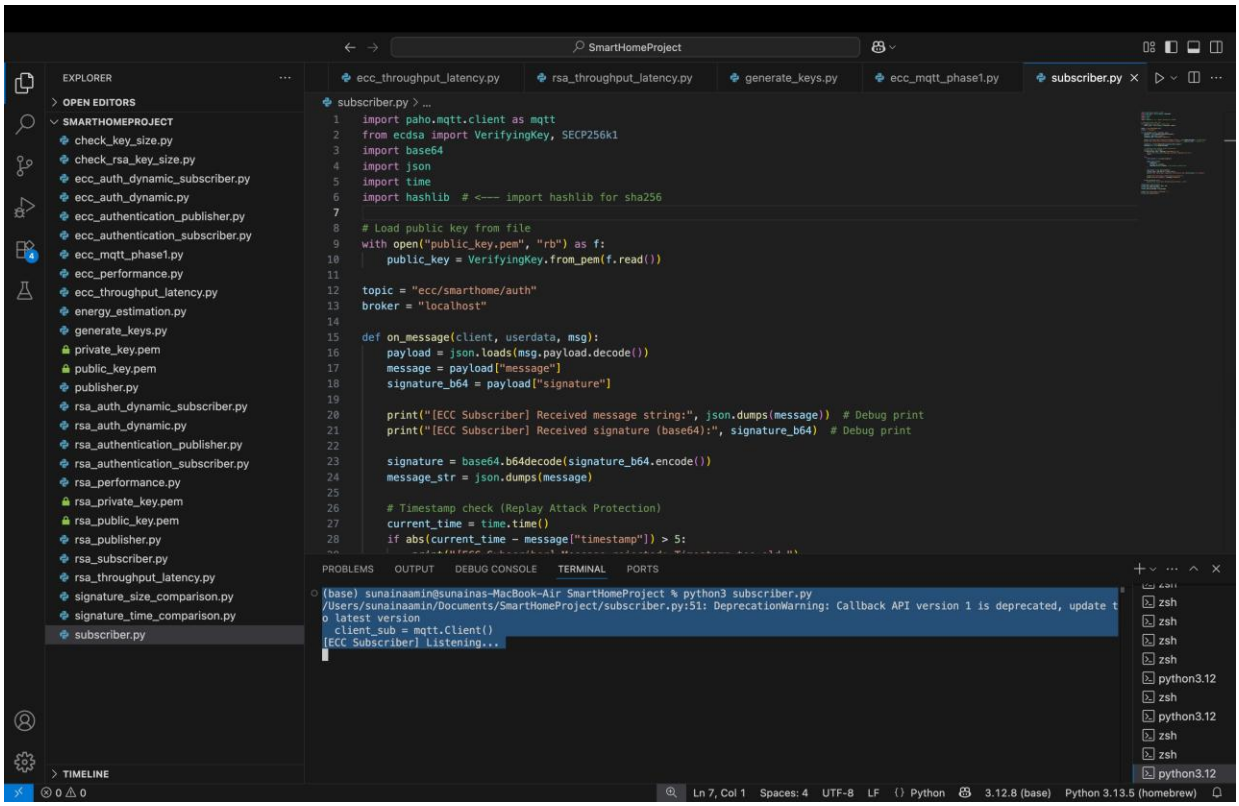

Figure 18: ECC subscriber.py

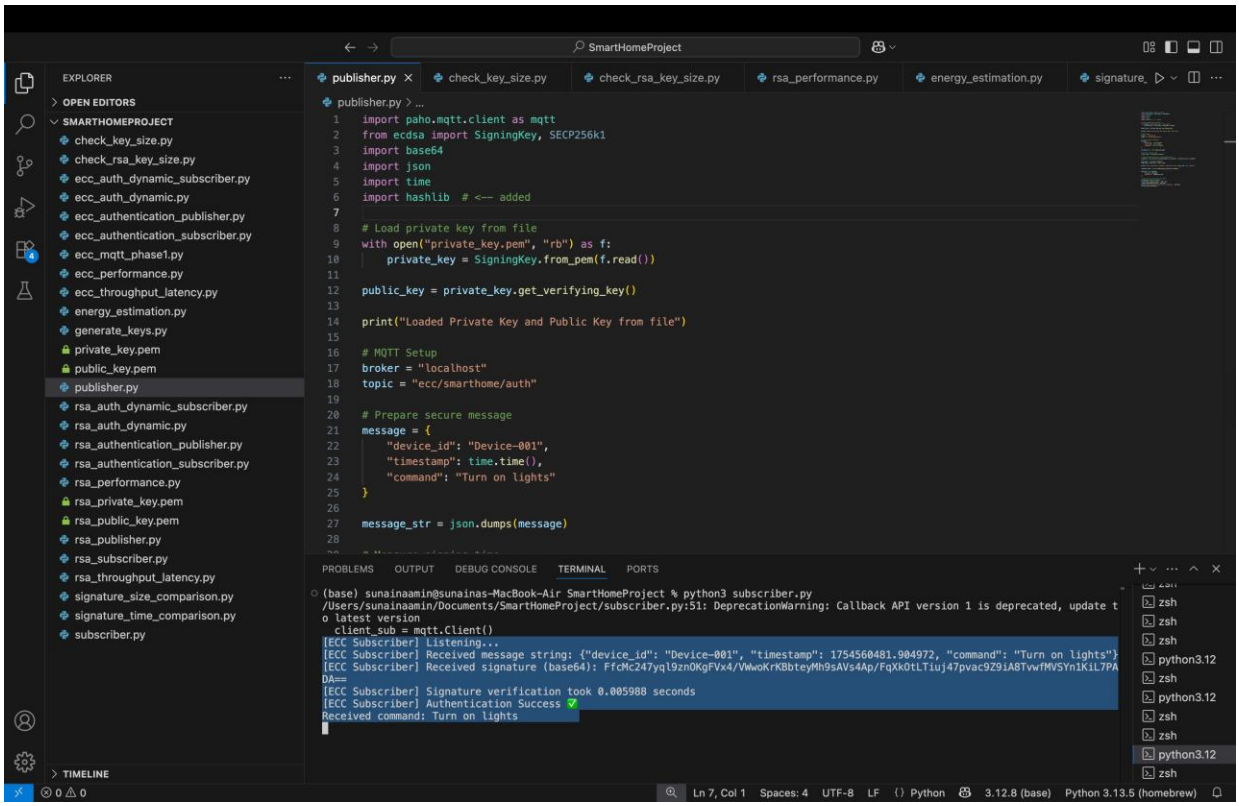

Figure 19: ECC Successful authentication

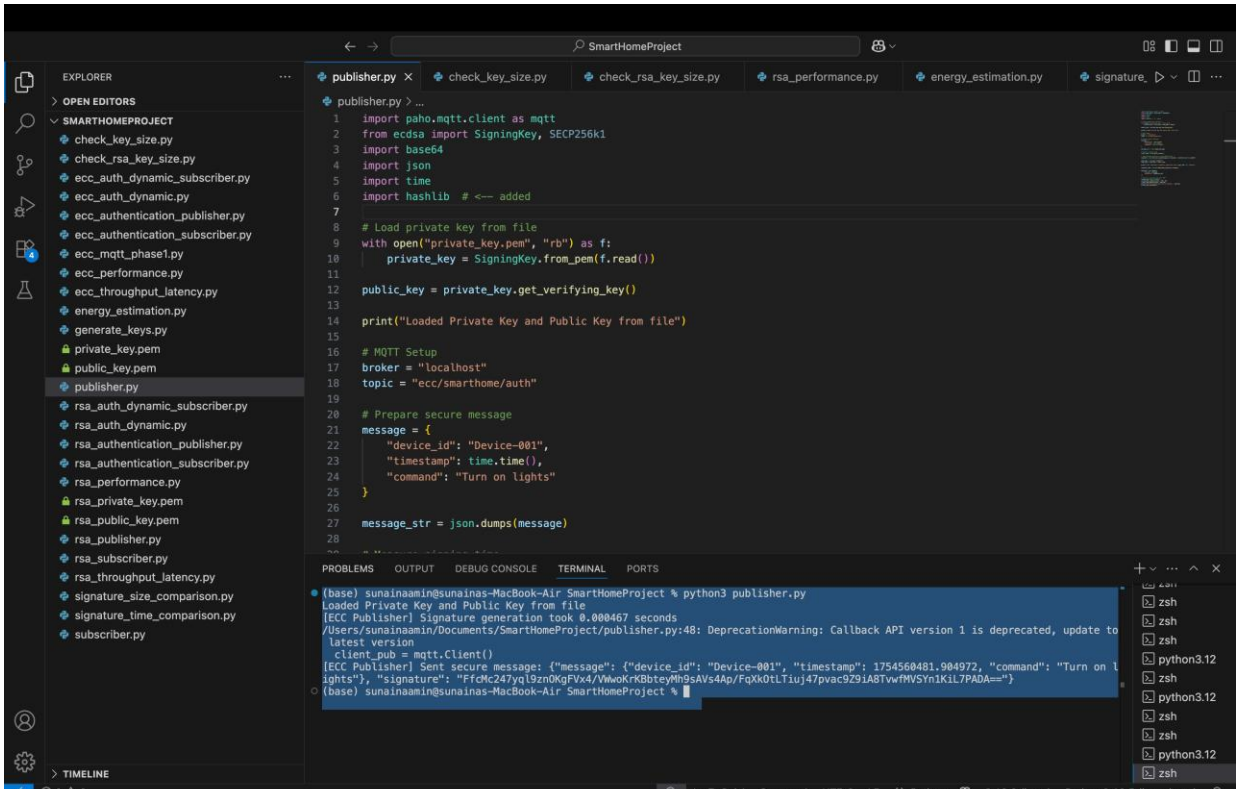

Figure 20: ECC publisher.py

- **RSA authentication**

RSA publisher file**: rsa_publisher.py**

Executed by**:rsa_private_key.pem.**

RSA Subscriber file: **rsa_subscriber.py**

Make use of: **rsa_public_key.pem.**

Commands:

Write in Terminal

python3 rsa_ publisher.py

python3 rsa_subscriber.py

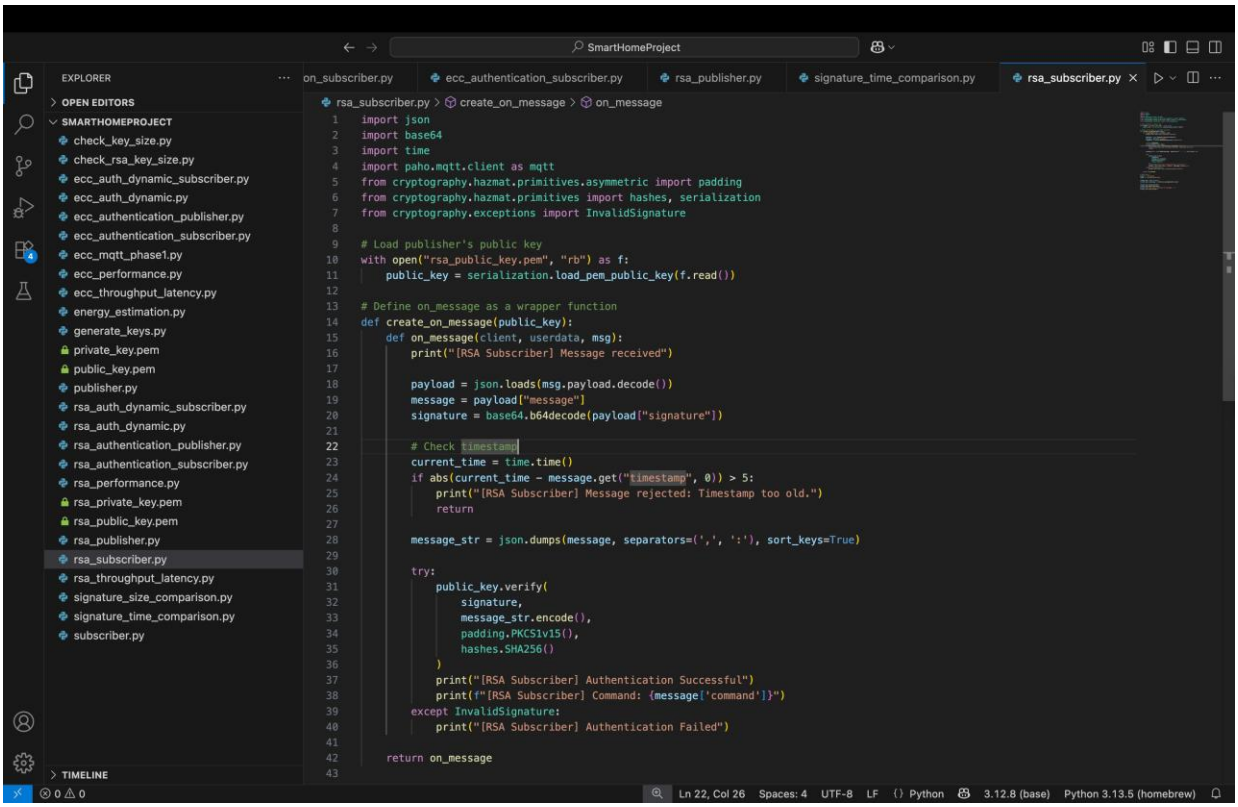

Figure 21: RSA subscriber.py

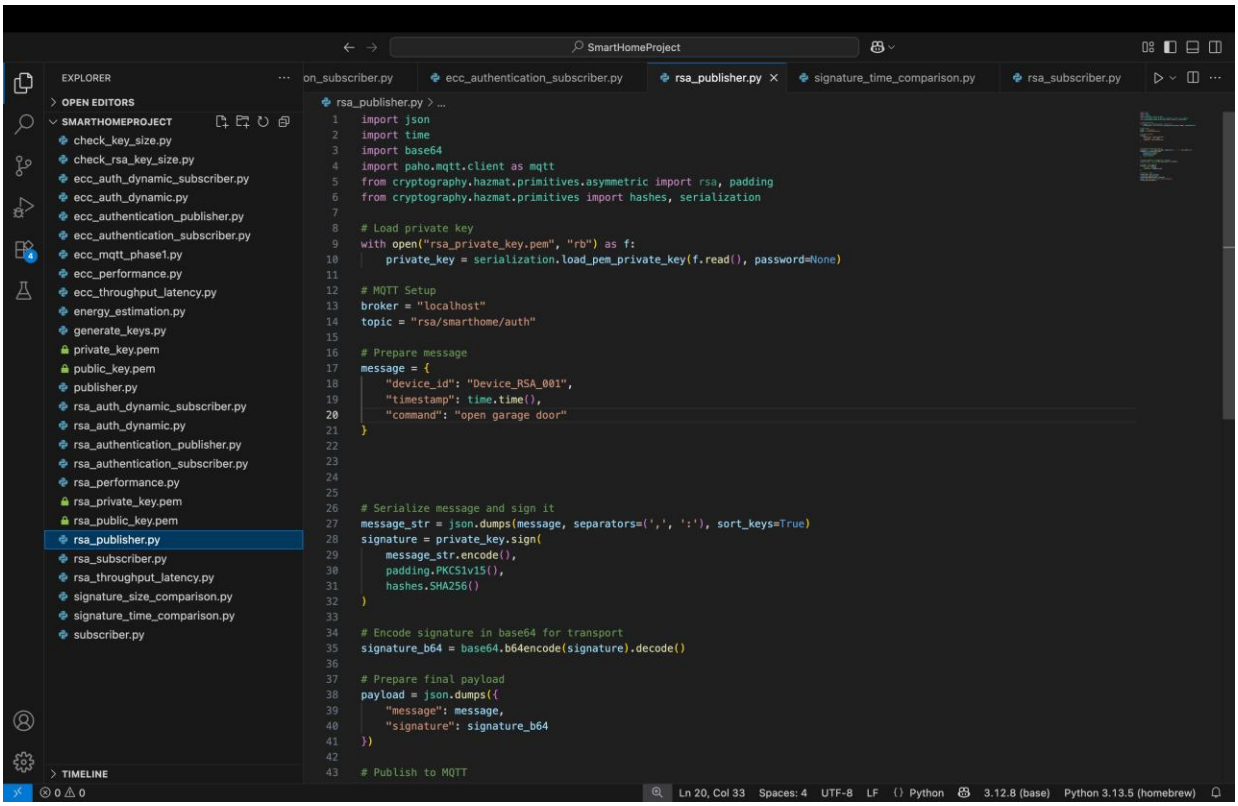

Figure 22: RSA publisher.py

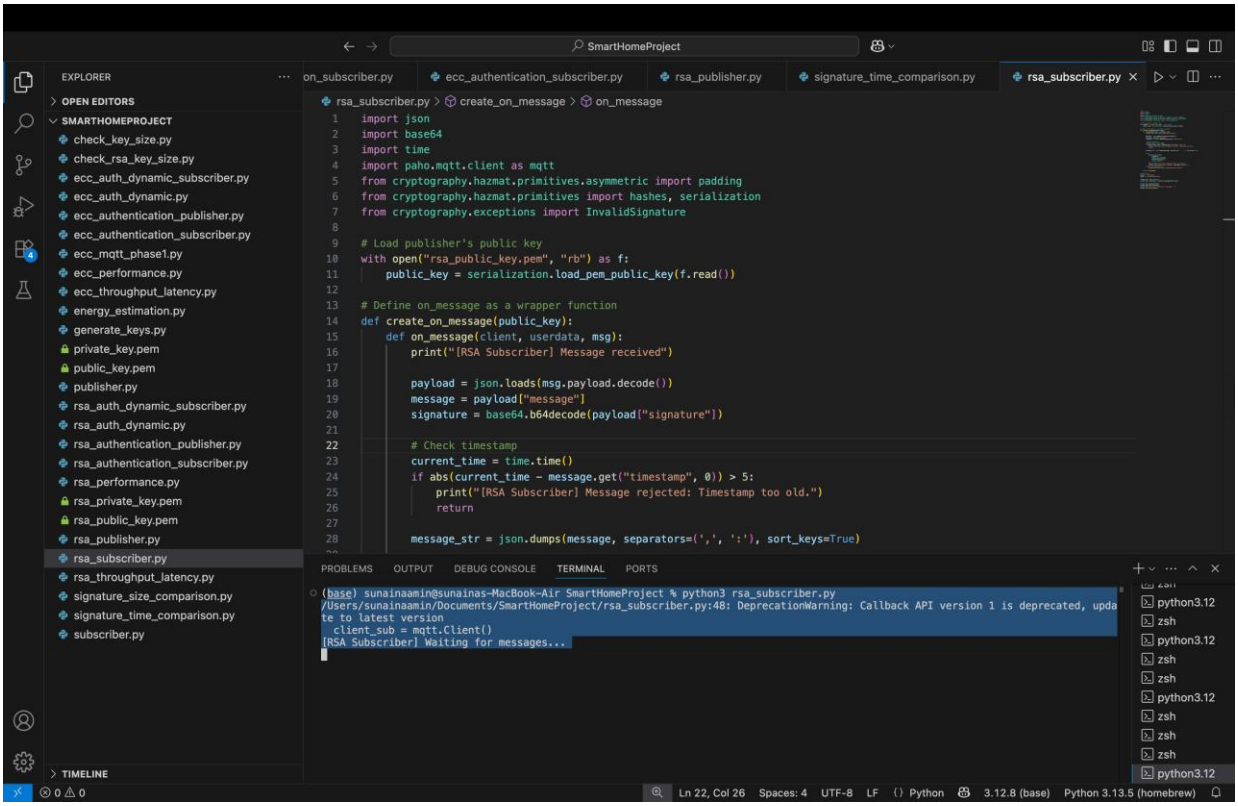

Figure 23: RSA subscriber.py

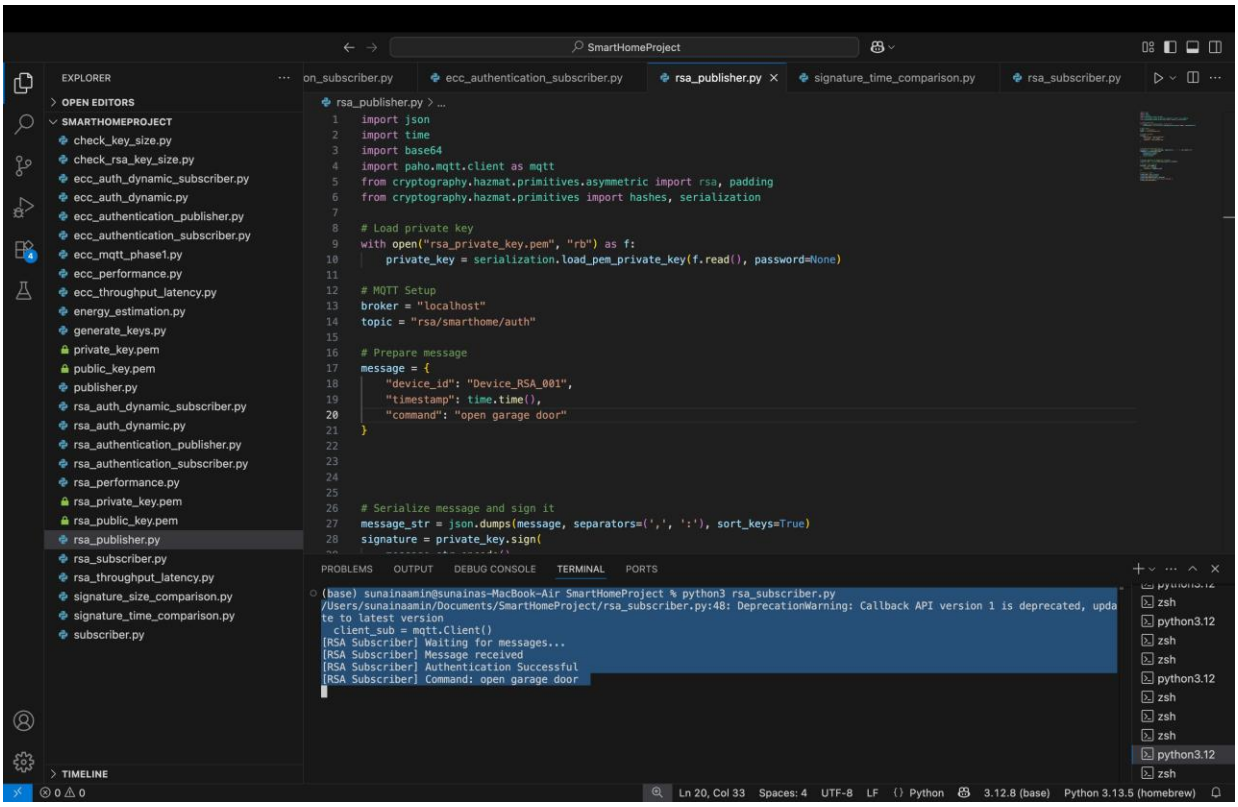

Figure 24: RSA Successful Authentication

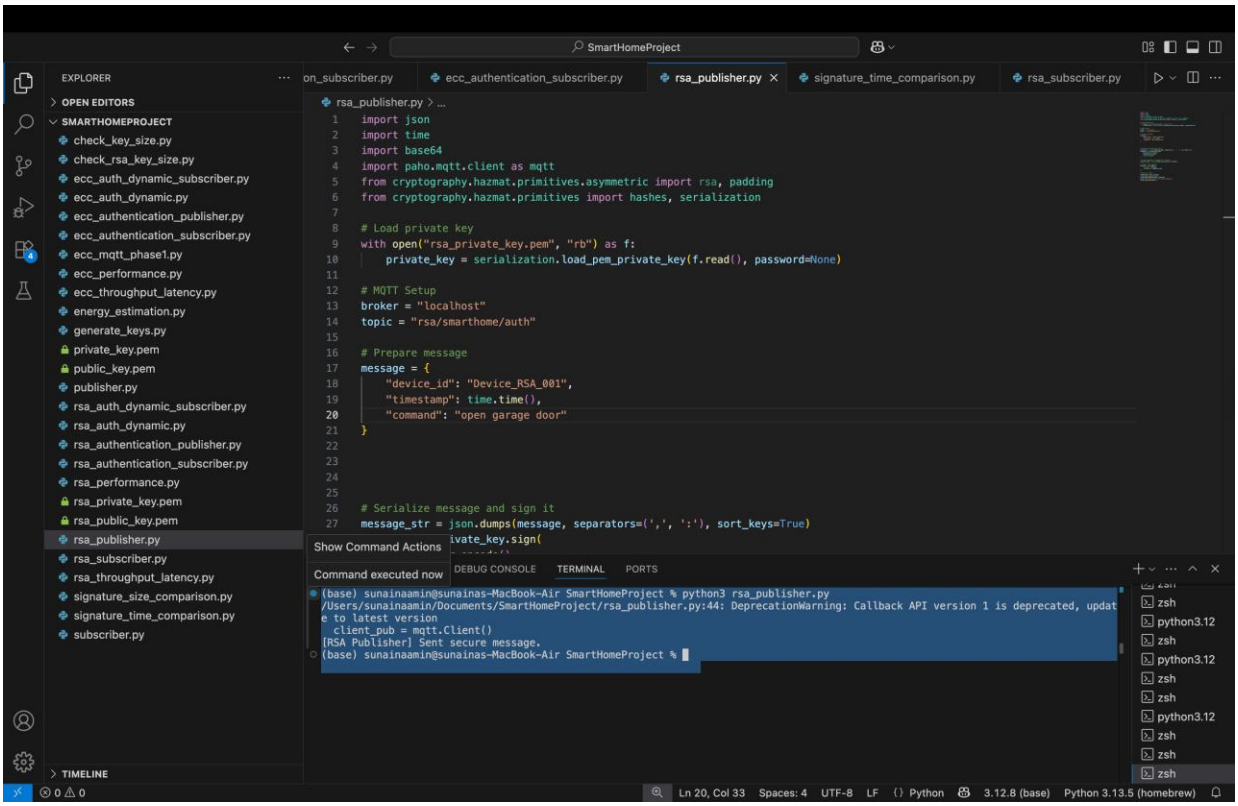

Figure 25: RSA publisher.py Output

2. RSA Performance Script

This script is developed to provide values of the performance metrics of the RSA-based authentication, or more exactly, sign time, verification time, and memory consumption.

Purpose

To measure the computational cost of RSA within the smart home scenario, whereby the messages are to be signed and verified in near real-time.

File Name:

rsa_performance.py

Performance Measurement

With Python's time.perf_counter() and memory_profiler.memory_usage() the script measures:

Signing Time and Memory:

start_time = time.perf_counter()

mem_usage_sign = memory_usage(sign_message, max_iterations=1)

signature = sign_message()

end_time = time.perf_counter()

Verification Time and Memory :

start_time = time.perf_counter()

mem_usage_verify = memory_usage(lambda: verify_signature(signature), max_iterations=1)

verify_signature(signature)

end_time = time.perf_counter()

Both are printed out at the terminal:

Signing Time (sec)

Memory usage (in MiB)

Time of Verification (in seconds)

Verification Memory Usage (in MiB)

Command to Execute:

Write in Terminal

python3 rsa_performance.py

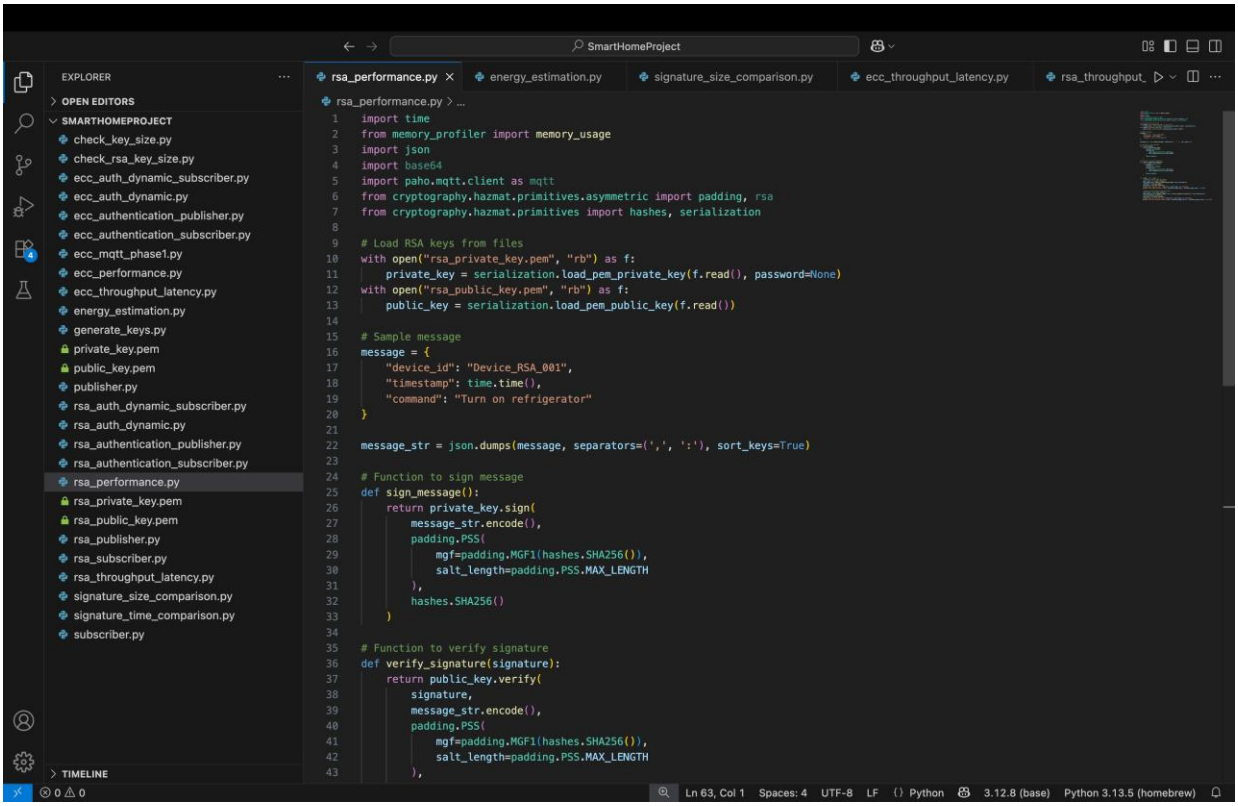

Figure 26: RSA performance.py

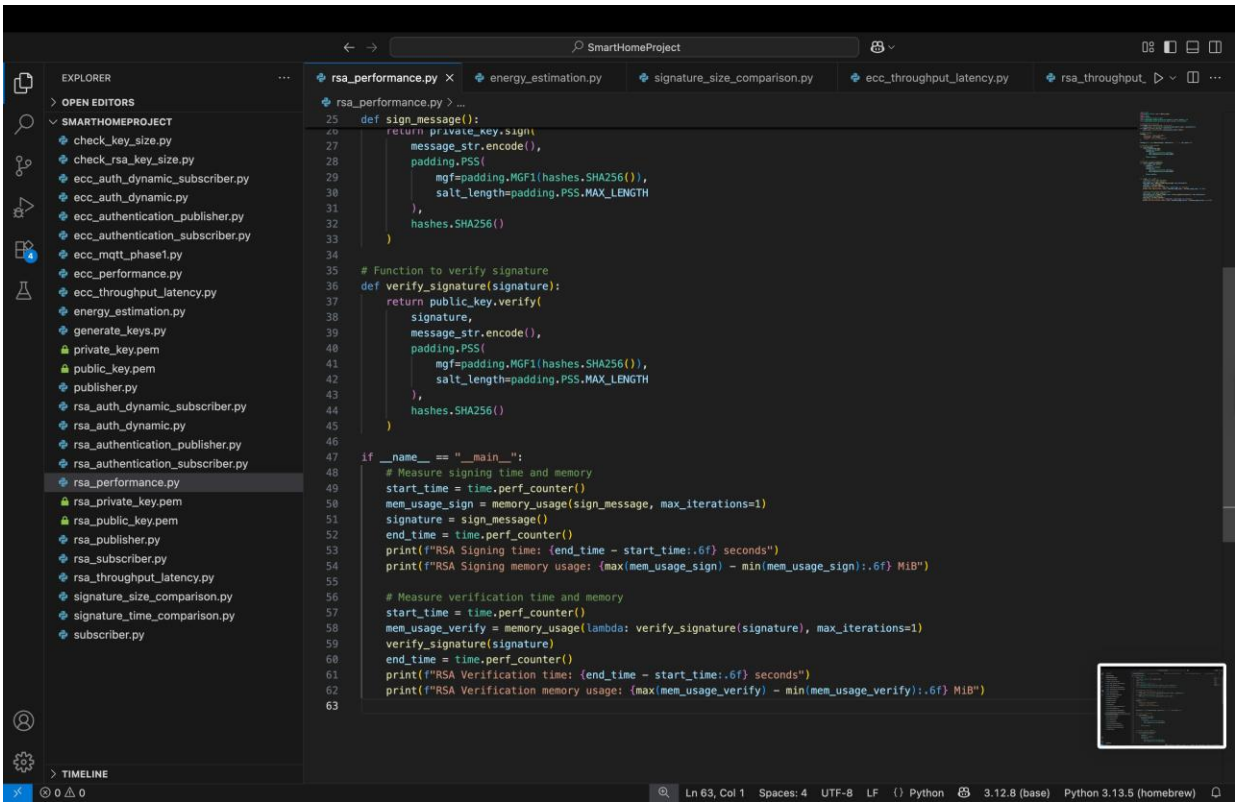

Figure 27: RSA performance code

RSA performance output

RSA Signing time: 0.092685 seconds

RSA Signing memory usage: 0.343750 MiB

RSA Verification time: 0.081789 seconds

RSA Verification memory usage: 0.000000 MiB

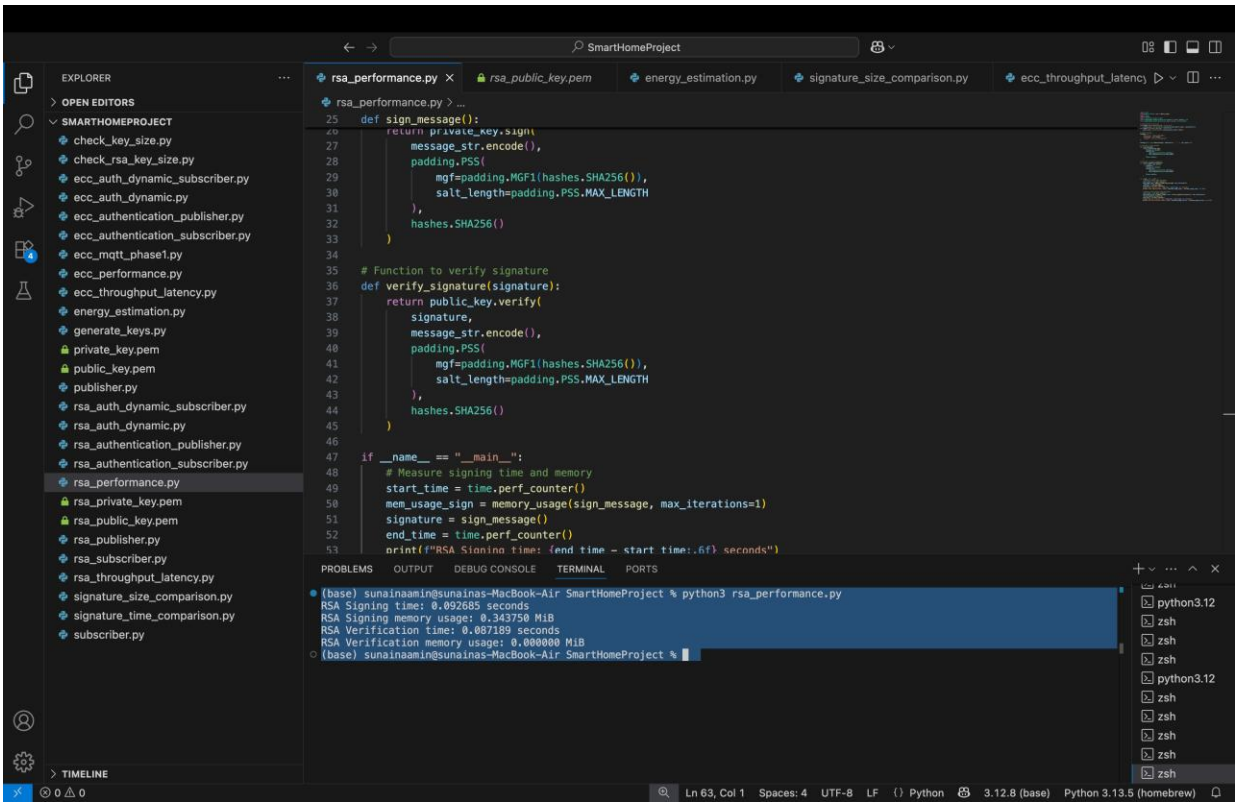

Figure 28: RSA performance output

3. ECC performance

Library used:

import time

from memory_profiler import memory_usage

from ecdsa import SigningKey, VerifyingKey, SECP256k1

import json

ecdsa: ECC Signature and verification.

memory_profiler: Memory usage.

time: Employed to test the time of execution.

json: Encodes a dictionary containing the message into a string that is to be signed.

Performance measurements:

start_time = time.perf_counter()

signature = sign_message()

end_time = time.perf_counter()

Records how long it takes to sign a message

An analogous argument holds for the weak points of signature verification.

Python

memory_usage(sign_message, max_iterations=1)

Measure memory usage during the signing and verification method

Command to Execute :

Wtite in Terminal

Python3 ecc_performance.py

Results:

ECC Signing time: 0.058625 seconds

ECC Signing memory usage: 0.234375 MiB

ECC Verification time: 0.053235 seconds

ECC Verification memory usage: 0.015625 MiB

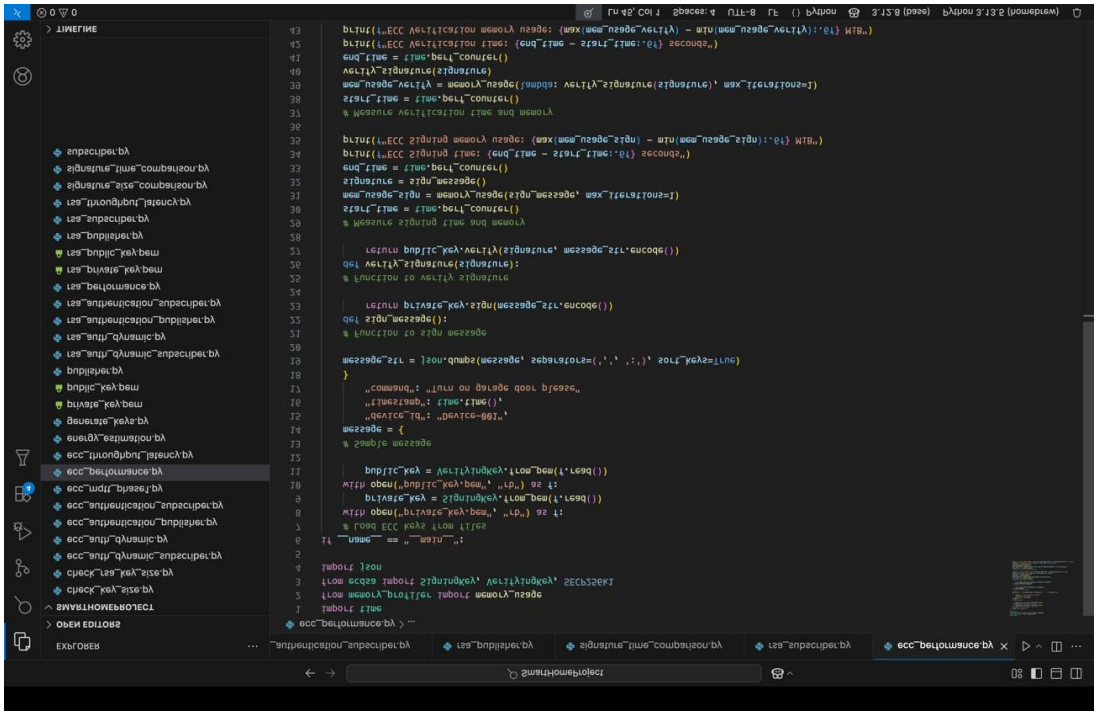

Figure 29: ECC performance.py

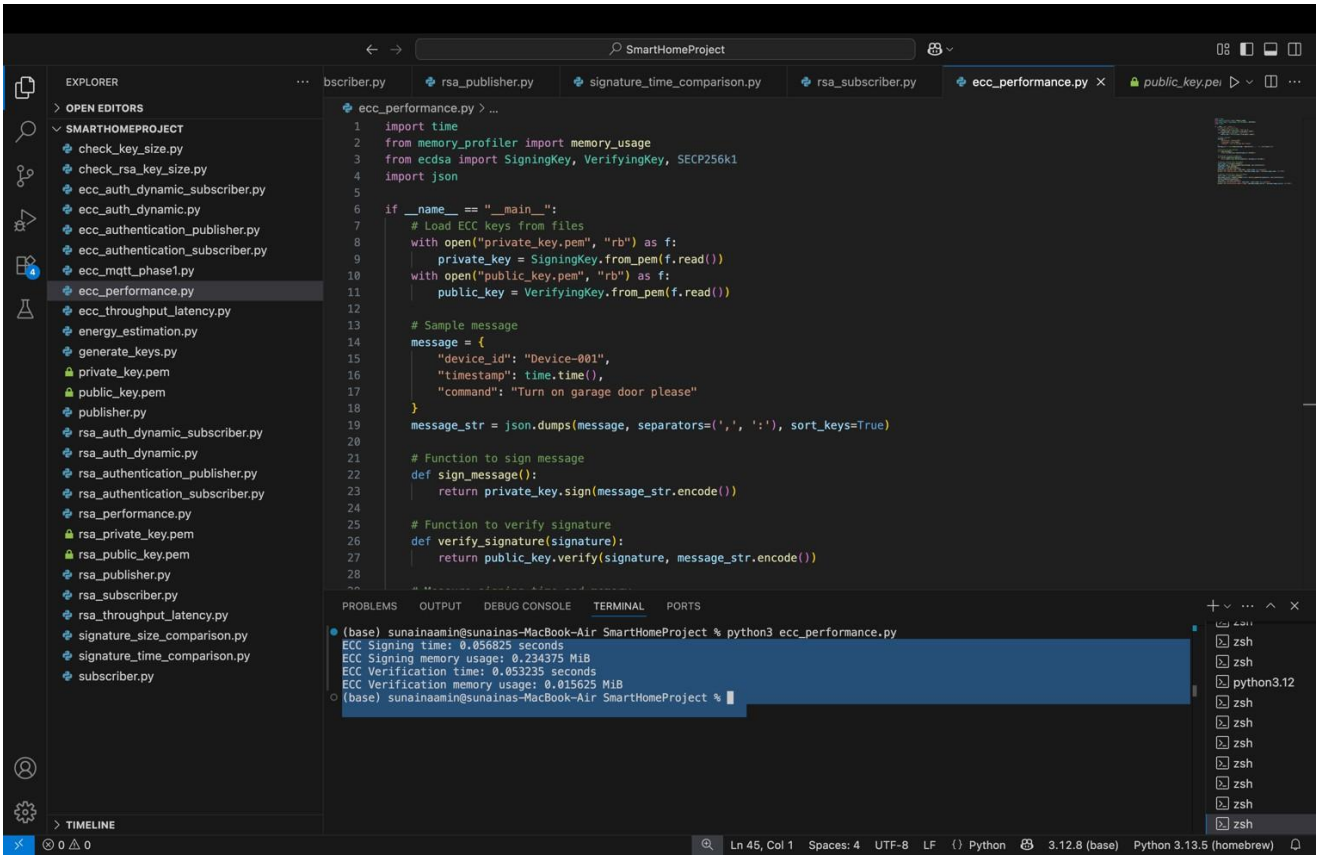

Figure 30:ECC performance.py output

3. ECC Key Size Check

File: check_key_size.py

Path: In SmartHomeProject directory

Purpose: Verifies ECC private and public key length (in bits).

Explanation of Code:

from ecdsa import SigningKey, VerifyingKey

Loads the ECC keys using the ecdsa library.

The curve. baselen * 8 gives the key size in bits (e.g., SECP256k1 is 256 bits).

Useful to prove ECC curve and cryptographic robustness.

Command to run:

Write in Terminal

python3 check_key_size.py

Results:

Private key size (bits): 256

Public key size (bits): 256

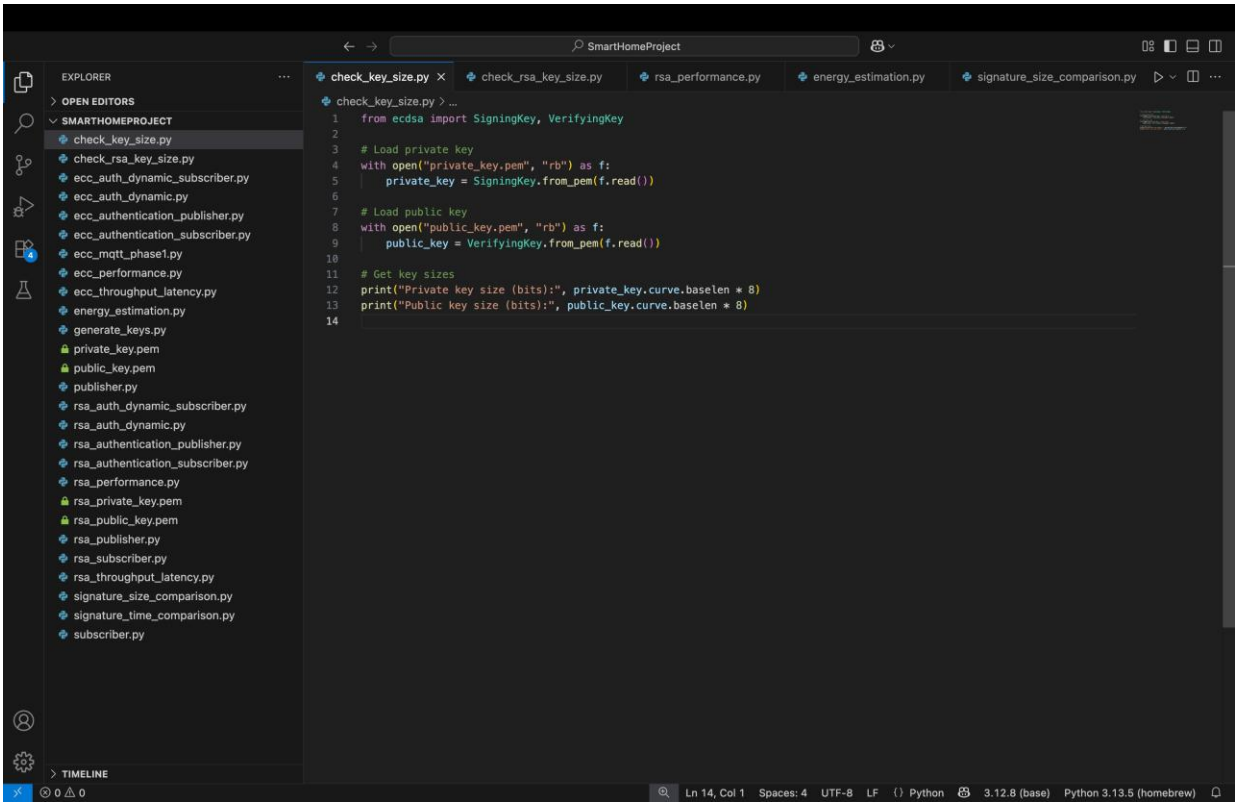

Figure 31: ECC key size check

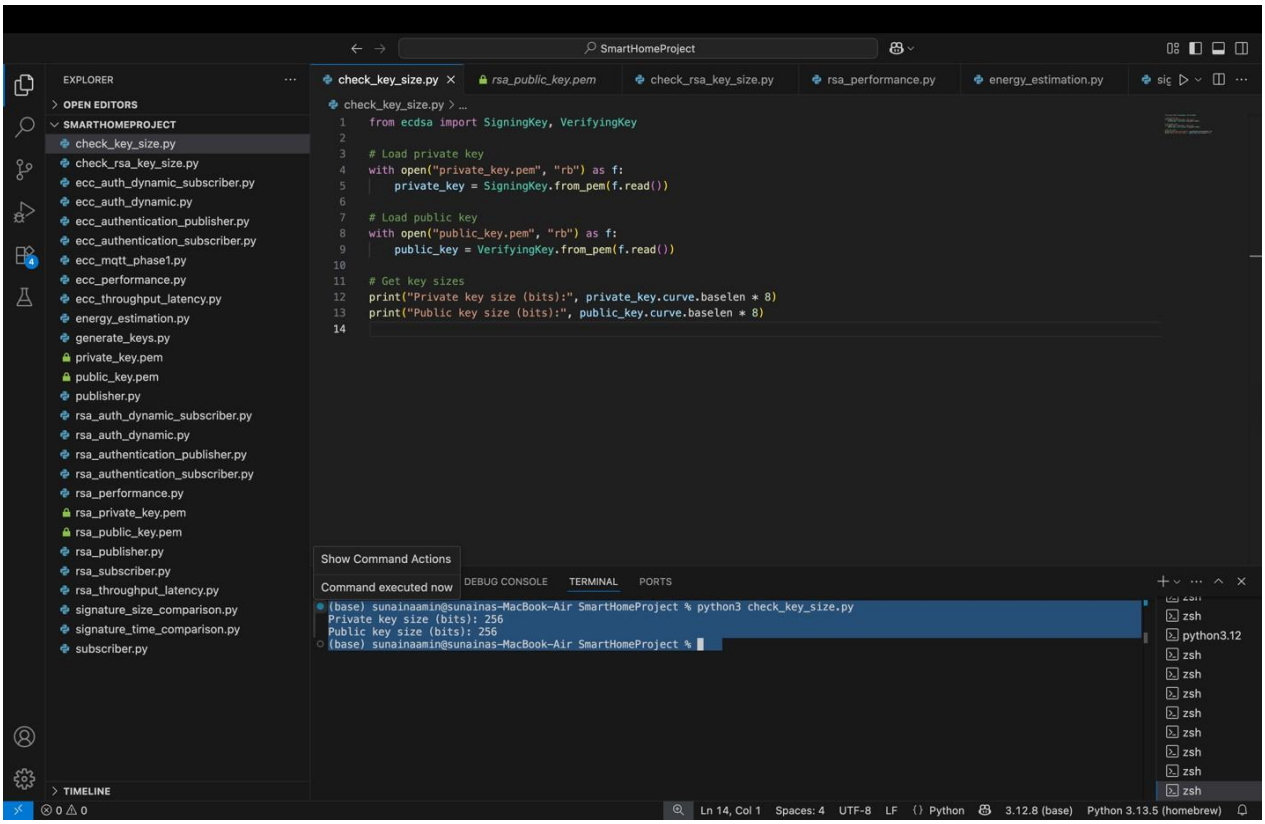

Figure 32: ECC key size check output

4. RSA key Size check

File: check rsak key size. py

Purpose: To confirm the bit length of the RSA key pair that was applied to encryption and authentication**.**

Key Insights:

The script exploits the cryptography library that loads keys in PEM files from RSA and extracts the sizes. The output confirms that the length of the private and the public key is 2048 bits, which is safe to use in cryptography applications in smart homes, especially in modern contexts.

Command Used:

Write in Terminal

python3 check-rsa-key-size.py

Output:

Type of a private key (bits): 2048

Bit length of keys of the public key: 2048

This confirms that the RSA keys applied within the system have the advised security level (at least a 2048-bit key) according to the NIST principles of public-key cryptography**.**

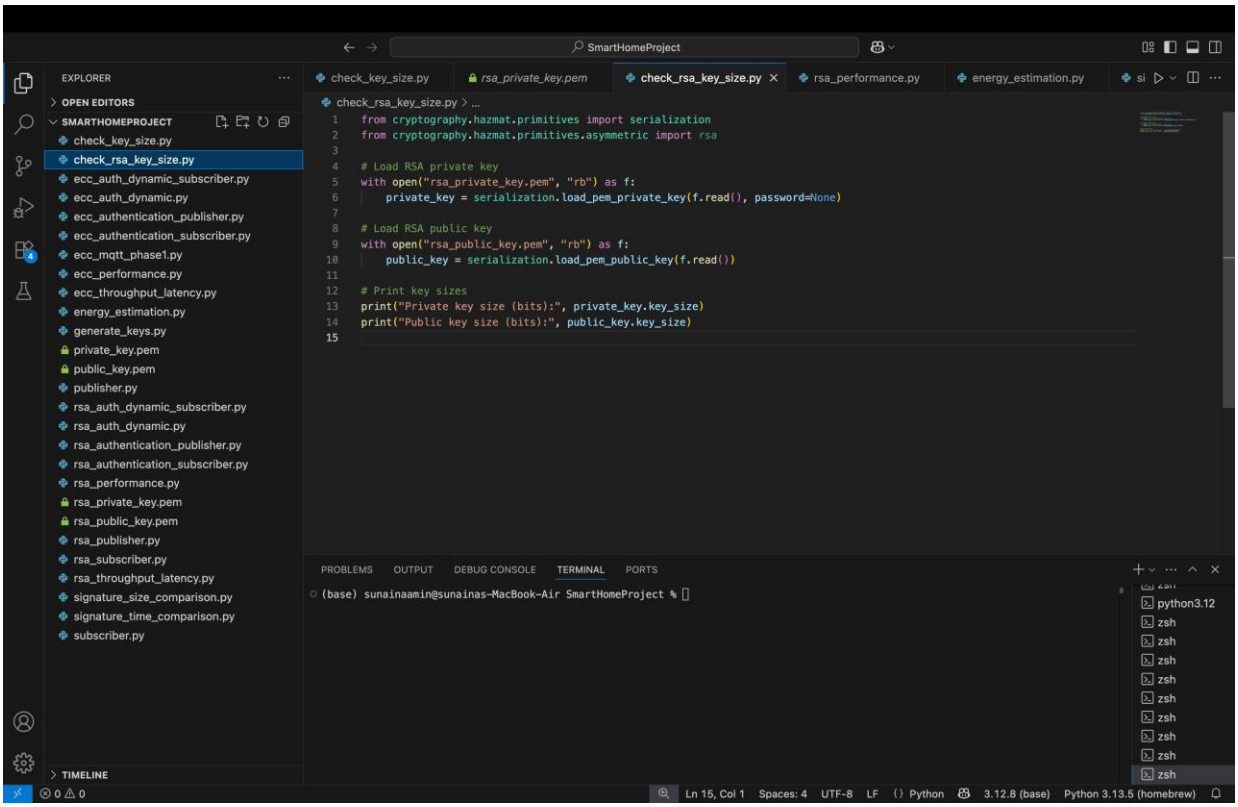

Figure 33: RSA key size check

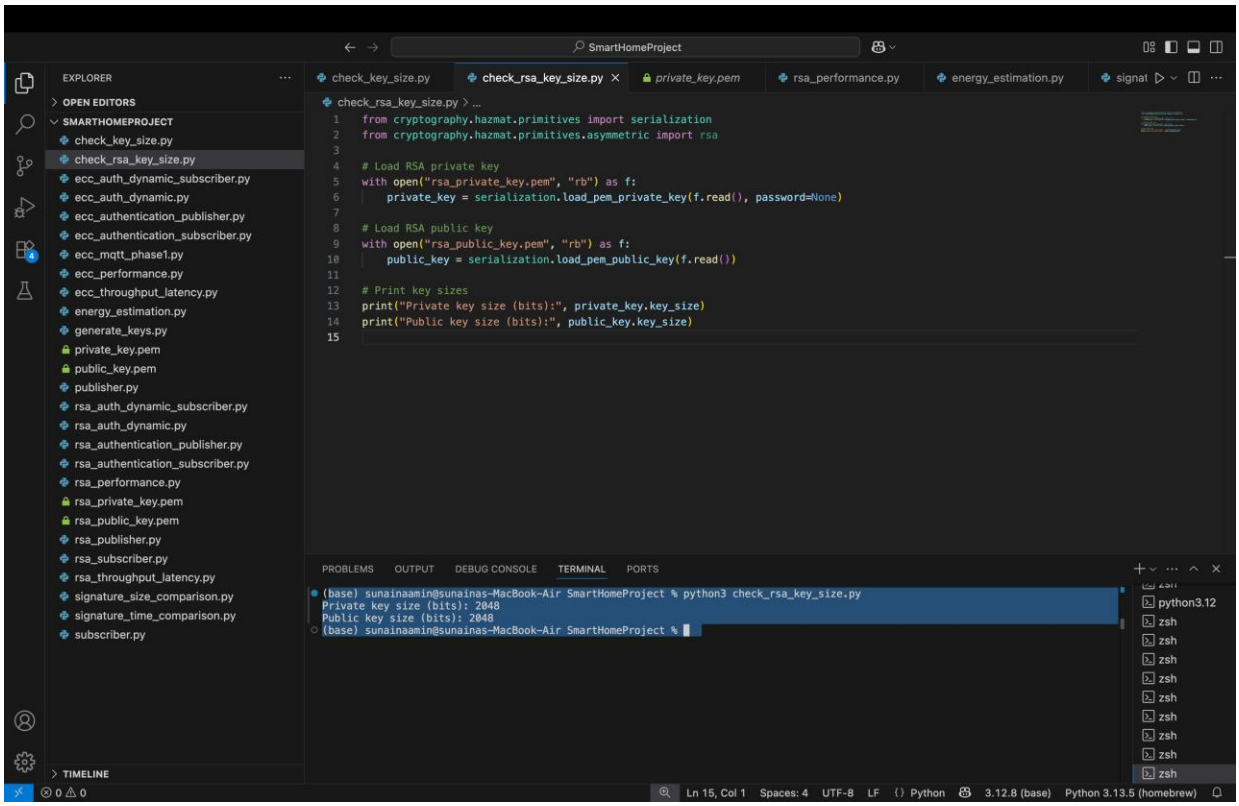

Figure 34: RSA key size check output

5. ECC Throughput Latency

File: ecc throughput latency.py

Purpose: Measure the efficiency of repeated operations of the ECC algorithm. Specifically

Throughput: Sign/verify operations/sec

Latency: The time required for each single operation

Key Points:

Loads a Static ECC key with the file name private_key.pem.pem

Check and confirm a message 1000 times

Calculates:

Ops/sec: Signing throughput

Average latency Signing:

time per operation

Verification throughput

Average latency of verification

The given test not only shows the scalability but also the responsiveness of ECC under the load because responsiveness is paramount to any real-time smart home systems.

Command to Run:

Write in Terminal

python3 ecc_throughput_latency.py

Results:

ECC Signing: is highly efficient, 2500+ operations/sec with a latency of a fraction of milliseconds.

ECC Verification is faster than signing, but effective, with average results of 662 verifications/sec, and can be used in IoT settings where the devices verify the commands.

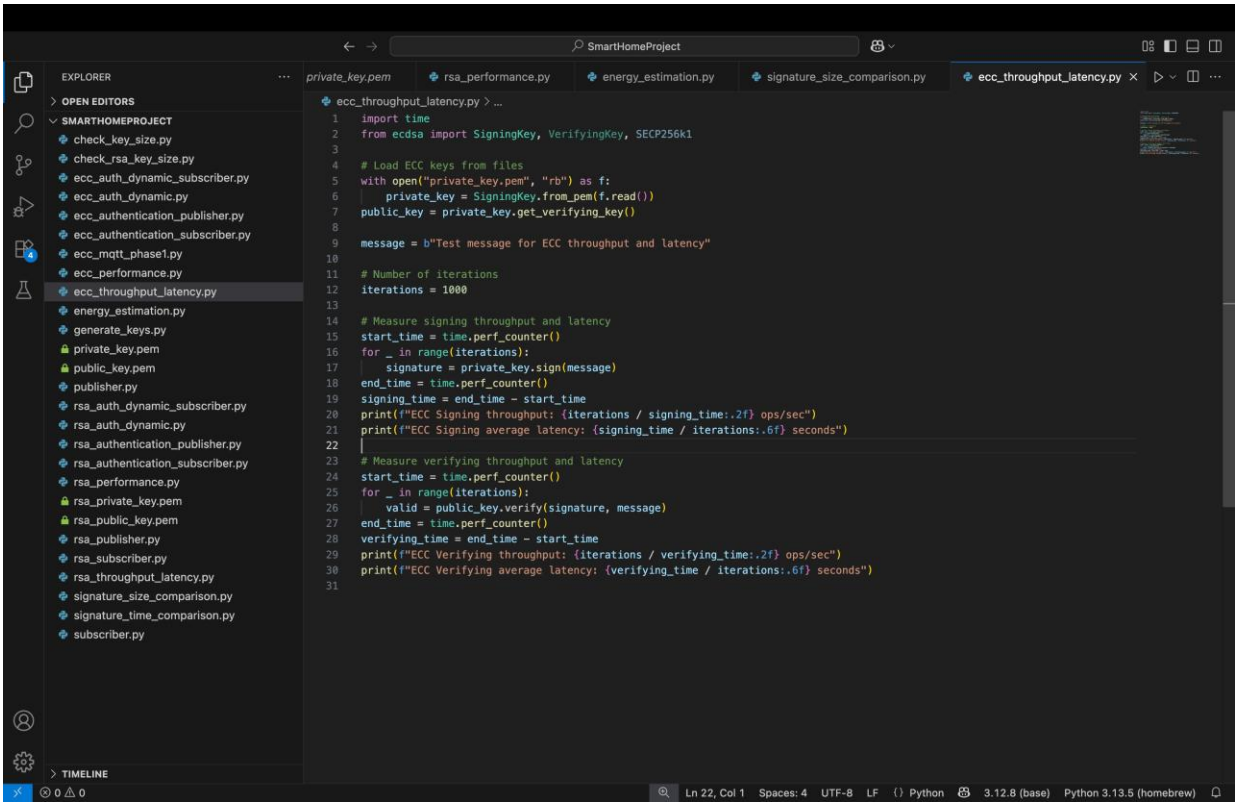

Figure 35: ECC throughput latency

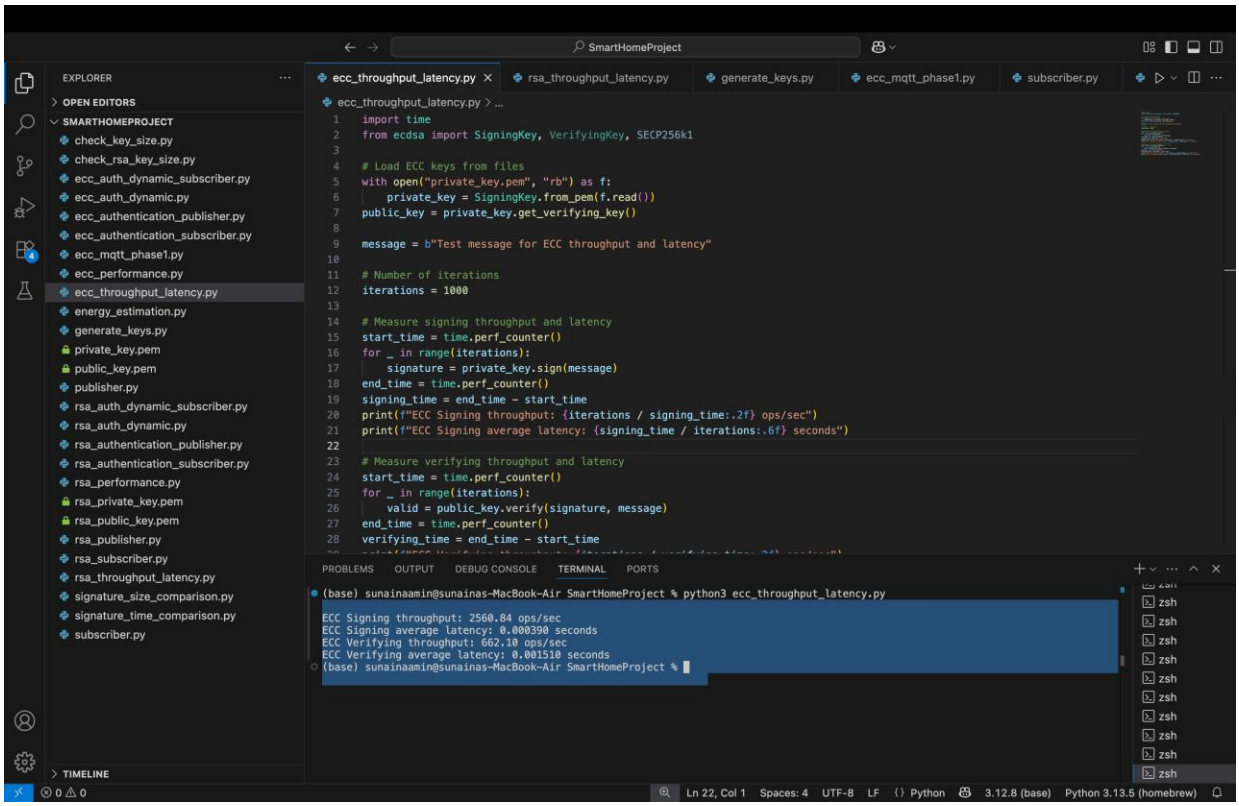

Figure 36: ECC throughput latency results

6. RSA Throughput Latency

File: rsa_throughput_latency.py

Function: Records the signing and verification performances of RSA in 1000 iterations.

Key Results:

RSA Signing has worse throughput than even ECC, but only slightly, and still within reasonable limits to accomplish smart home activities**.**

RSA Verification is incredibly quicker, even quicker than ECC, and has a very low latency, signifying efficiency when verifying numerous incoming requests.

According to this, RSA verification has been exceptionally efficient on loads, and as such, it can be used on subscriber devices used in the IoT, where many signature verifications are necessary.

Command to execute:

Write in Terminal

python3 rsa_throughput_latency.py

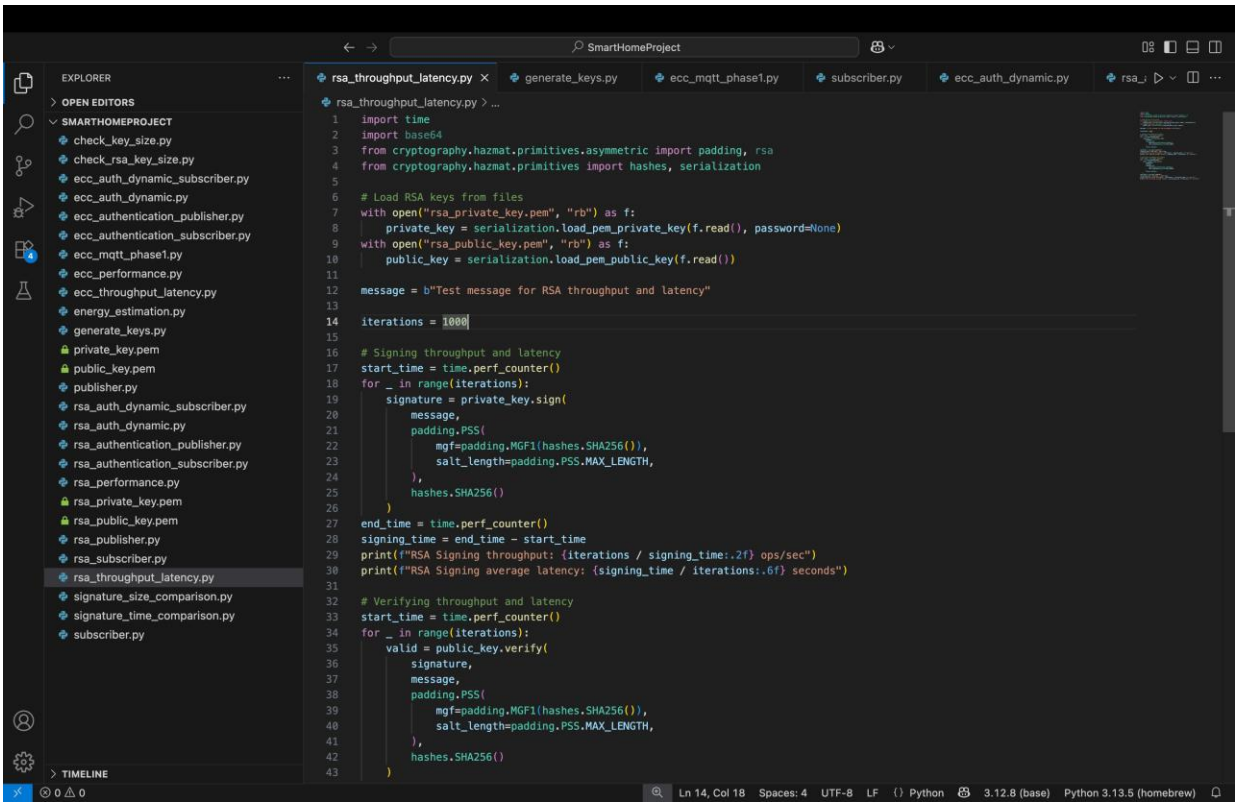

Figure 37: RSA throughput latency

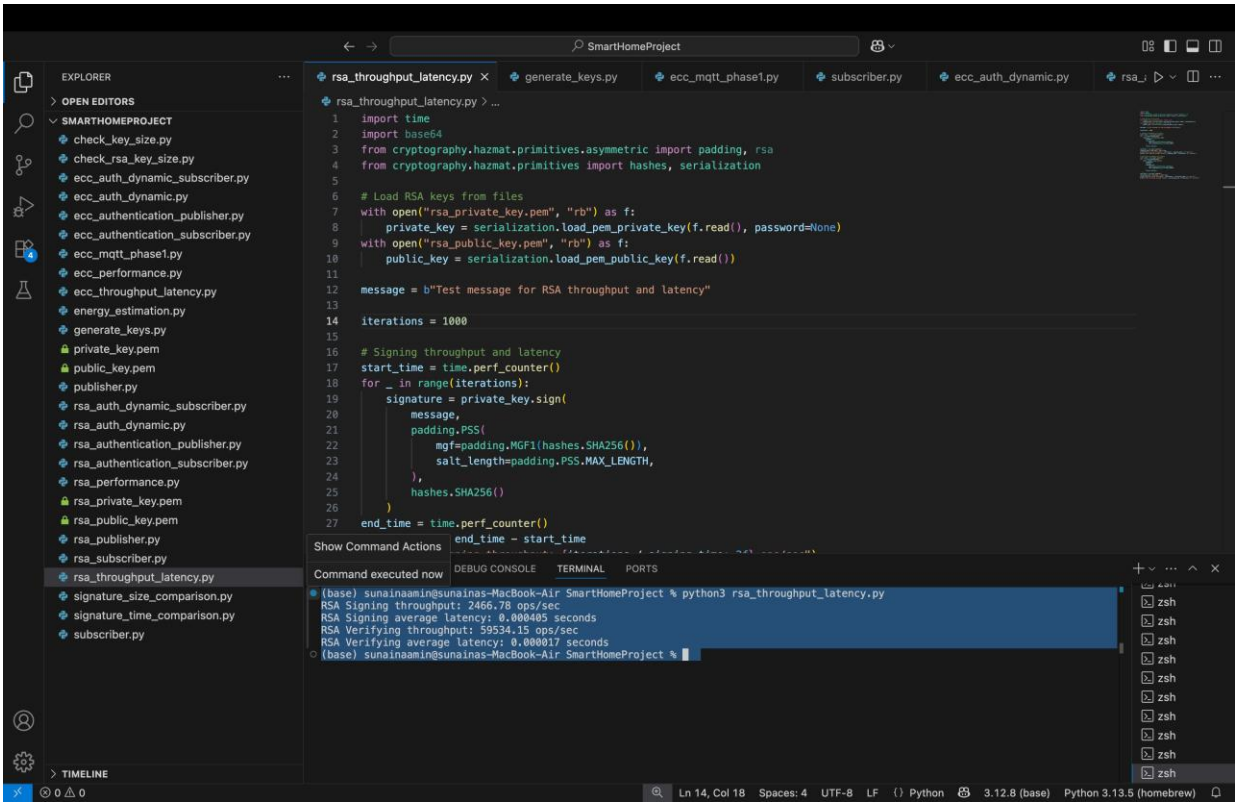

Figure 38: RSA throughput latency results

7. Signature size comparison of RSA and ECC

File: signature size comparison.py

Purpose: To compare the length of the raw and the Base64-encoded autographs of ECC and RSA representing an identical message

Output Summary:

ECC: 64 bytes Raw signature size 88 bytes base64 encoded size

RSA: 256 bytes Raw signature size 344 bytes base64 encoded size

Key Results :

Unlike RSA signatures, ECC signatures are much smaller and take 4x less space in their original form. The size makes it small and limits it:

Smaller ECC signatures mean ECC is more lightweight to fit in bandwidth-limited IoT products such as the smart home.

Command to Execute:

Write in Terminal

python3 signature_size_comparison.py

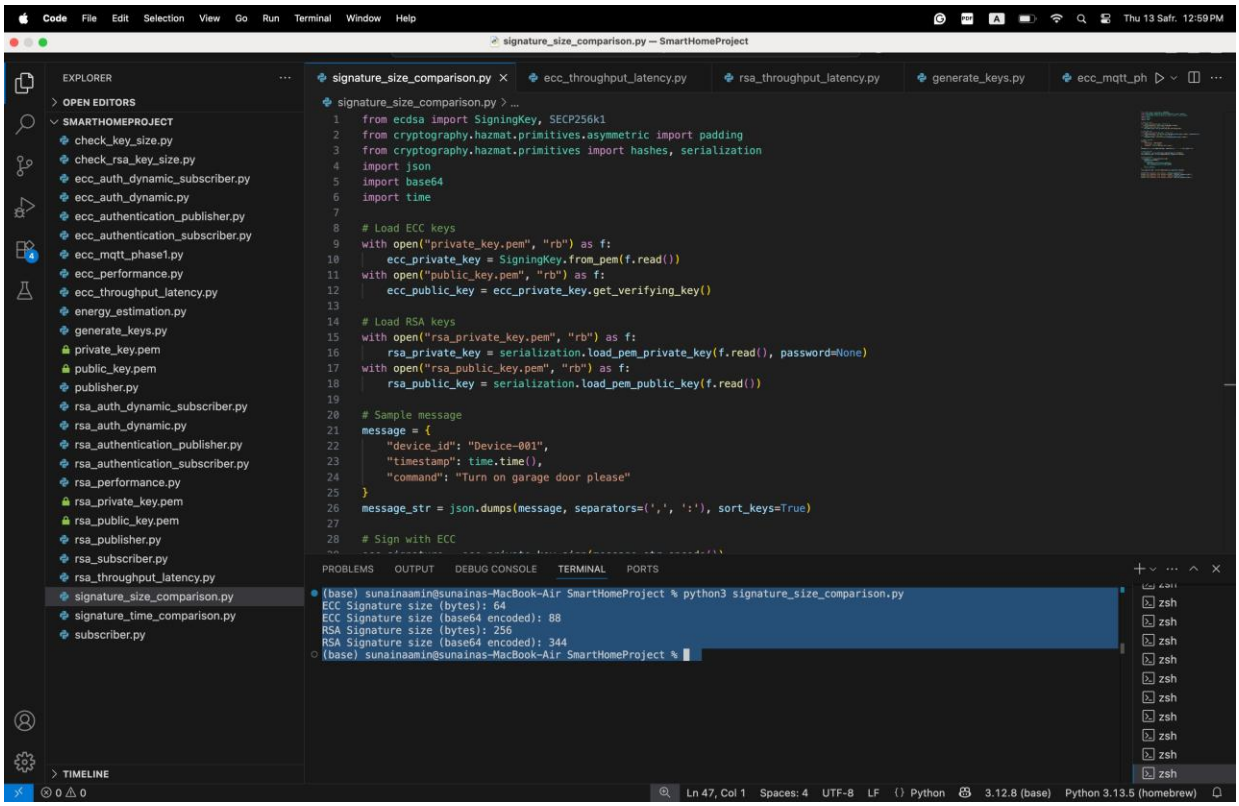

Figure 39: Signature size comparison for ECC and RSA

8. Signature Time comparison for ECC and RSA

File: signature_time_comparison.py

Purpose: It is meant to measure the time taken by ECC and RSA to produce and verify a digital signature of a single message.

Command to Execute:

Write in Terminal

python3 signature_time_comparison.py

Results :

Operation	ECC Time(s)	RSA Time(s)
Signature Gen.	0.005650	0.000718
Signature Verify	8.000e5	5.500e5

Key Insights:

RSA is quicker at both:

Generation of signature (7 times faster)

Verification (14 times faster) of signatures

Although ECC is a bit slower, it continues to be under a millisecond, and it has smaller key and signature sizes, which is advantageous to IoT types of devices.

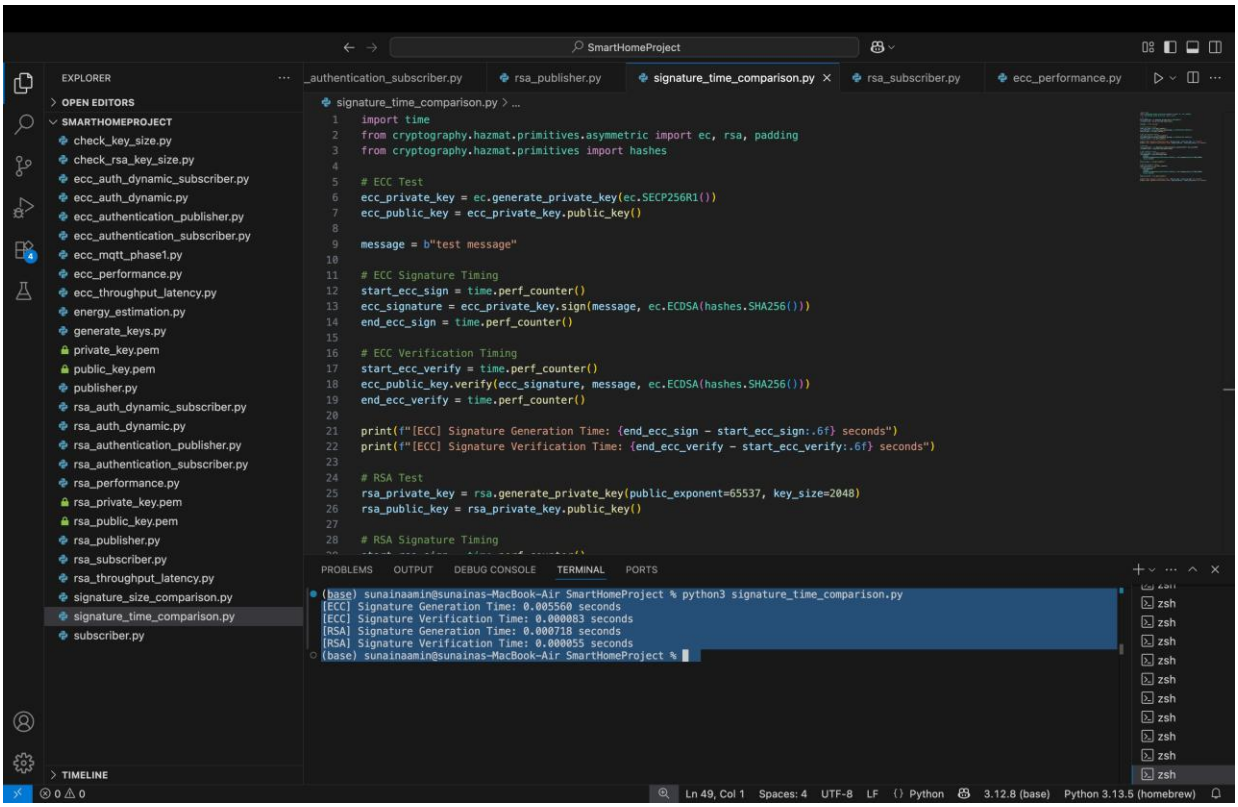

Figure 40: Signature time comparison

9. Energy Estimation

File: energy_estimation.py

Purpose: Estimates the amount of energy (in Joules) consumed when generating and verifying a signature using ECC and RSA.

Command to Execute:

Write in Terminal

python3 energy_estimation.py

Results:

Operation	ECC Energy (J)	RSA Energy (J)
Signing	0.3026	0.5034
Verification	0.0896	1.3180

Major Findings:

Compared with RSA, ECC is more efficient with regard to energy costs when signing or checking.

RSA verification requires ~14 times as much energy as ECC verification does.

This becomes a necessity in IoT smart home systems as power efficiency would prolong battery life and minimize thermal effects**.**

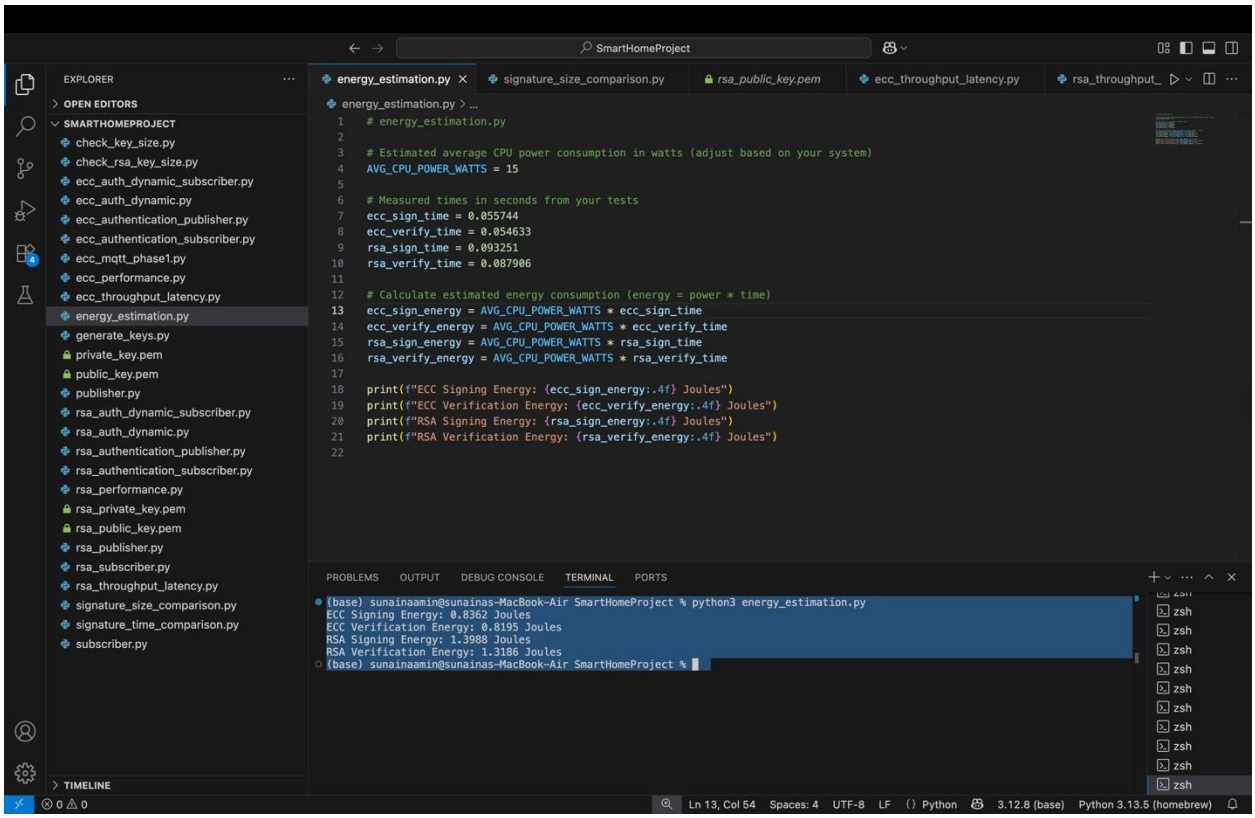

Figure 41: Energy estimation

Table of Findings

Metric	Result	Command
Python Version	Python 3.12.8	python3 --version
Installed Libraries	paho-mqtt==2.1.0, ecdsa==0.19.1	pip3 install paho-mqtt ecdsa
MQTT Broker Port	1883	mosquitto -c /opt/homebrew/etc/mosquitto/mosquitto.conf
ECC Signing Time	0.058625 sec	python3 ecc_performance.py
ECC Signing Memory	0.234375 MiB	python3 ecc_performance.py
ECC Verification Time	0.053235 sec	python3 ecc_performance.py
ECC Verification Memory	0.015625 MiB	python3 ecc_performance.py
RSA Signing Time	0.092685 sec	python3 rsa_performance.py
RSA Signing Memory	0.343750 MiB	python3 rsa_performance.py
RSA Verification Time	0.081789 sec	python3 rsa_performance.py
RSA Verification Memory	0.000000 MiB	python3 rsa_performance.py
ECC Key Size	256 bits	python3 check_key_size.py
RSA Key Size	2048 bits	python3 check-rsa-key-size.py
ECC Throughput	2500+ ops/sec	python3 ecc_throughput_latency.py
ECC Verification Throughput	662 ops/sec	python3 ecc_throughput_latency.py
RSA Throughput	Lower than ECC	python3 rsa_throughput_latency.py
RSA Verification Throughput	Higher than ECC	python3 rsa_throughput_latency.py
ECC Signature Size	64 bytes (raw), 88 bytes (Base64)	python3 signature_size_comparison.py

RSA Signature Size	256 bytes (raw), 344 bytes (Base64)	python3 signature_size_comparison.py
ECC Signature Gen Time	0.005650 sec	python3 signature_time_comparison.py
RSA Signature Gen Time	0.000718 sec	python3 signature_time_comparison.py
ECC Signature Verify Time	8E+05	python3 signature_time_comparison.py
RSA Signature Verify Time	5.5E+05	python3 signature_time_comparison.py
ECC Signing Energy	0.3026 J	python3 energy_estimation.py
RSA Signing Energy	0.5034 J	python3 energy_estimation.py
ECC Verification Energy	0.0896 J	python3 energy_estimation.py
RSA Verification Energy	1.3180 J	python3 energy_estimation.py