

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

Figure 1 python setup on macOS terminal	3
Figure 2 MQTT setup	5
Figure 3 VS Code Installation	6
Figure 4 ECC private key	8
Figure 5 ECC public Key	8
Figure 6: RSA public key	8
Figure 7: RSA private key	9
Figure 8: Key Generation Setup	10
Figure 9: Dynamic Key Generation	10
Figure 10: ECC Dynamic subscriber.py	15
Figure 11: ECC dynamic Publisher.py	15
Figure 12: ECC Dynamic Publisher output	16
Figure 13: ECC dynamic subscriber output	17
Figure 14: Authentication Successful	17
Figure 15: RSA Dynamic Subscriber	20
Figure 16: RSA dynamic Publisher	21
Figure 17: RSA dynamic key authentication successful	21
Figure 18: ECC subscriber.py	23
Figure 19: ECC Successful authentication	24
Figure 20: ECC publisher.py	24
Figure 21: RSA subscriber.py	25
Figure 22: RSA publisher.py	26
Figure 23: RSA subscriber.py	26
Figure 24: RSA Successful Authentication	27
Figure 25: RSA publisher.py Output	28
Figure 26: RSA performance.py	29
Figure 27: RSA performance code	30
Figure 28: RSA performance output	31
Figure 29: ECC performance.py	33
Figure 30: ECC performance.py output	34
Figure 31: ECC key size check	35
Figure 32: ECC key size check output	36
Figure 33: RSA key size check	37
Figure 34: RSA key size check output	38
Figure 35: ECC throughput latency	39
Figure 36: ECC throughput latency results	40
Figure 37: RSA throughput latency	41
Figure 38: RSA throughput latency results	41
Figure 39: Signature size comparison for ECC and RSA	43
Figure 40: Signature time comparison	44
Figure 41: Energy estimation	45

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.

```

Removing: /Users/sunainaamin/Library/Caches/Homebrew/pkg-config--8.29.2_3... (235.9KB)
Removing: /Users/sunainaamin/Library/Caches/Homebrew/sleuthkit--4.12.1... (78.9MB)
Removing: /Users/sunainaamin/Library/Caches/Homebrew/portable-ruby-3.3.5.arm64_big_sur.bottle.tar.gz... (11.2MB)
Removing: /Users/sunainaamin/Library/Caches/Homebrew/sleuthkit_bottle_manifest--4.12.1... (44.9KB)
Removing: /Users/sunainaamin/Library/Caches/Homebrew/pkg-config.bottle.manifest--8.29.2_3... (13.3KB)
Removing: /Users/sunainaamin/Library/Caches/Homebrew/Cocoa/librca-jdk8-full--1.8.0_432.pkg... (143.0MB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/glib-networking... (190B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/libnetf... (487B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/gtk+3... (3 files, 1.6KB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/postgresql915... (1.3KB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/python@3.13... (2 files, 2KB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/gdk-pixbuf... (210B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/shred-mime-info... (514B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/glib... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/dbus... (221B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/Fontconfig... (146.9KB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/gtk4... (3 files, 1.5KB)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/openssl@3... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/ca-certificates... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/gstreamer... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/unbound... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/guile... (64B)
Removing: /Users/sunainaamin/Library/Logs/Homebrew/gsettings-desktop-schemas... (1.2KB)
==> Caveats
==> mosquitto
mosquitto has been installed with a default configuration file.
You can make changes to the configuration by editing:
/opt/homebrew/etc/mosquitto/mosquitto.conf

To start mosquitto now and restart at login:
brew services start mosquitto
Or, if you don't want/need a background service you can just run:
/opt/homebrew/opt/mosquitto/sbin/mosquitto -c /opt/homebrew/etc/mosquitto/mosquitto.conf
==> postgresql915
This formula has created a default database cluster with:
initdb --locale=C -E UTF-8 /opt/homebrew/var/postgresql915

postgresql915 is keg-only, which means it was not symlinked into /opt/homebrew,
because this is an alternate version of another formula.

If you need to have postgresql915 first in your PATH, run:
echo 'export PATH="/opt/homebrew/opt/postgresql915/bin:$PATH"' >> ~/.zshrc

For compilers to find postgresql915 you may need to set:
export LD_LIBRARY_PATH="/opt/homebrew/opt/postgresql915/lib"
export CPPFLAGS="-I/opt/homebrew/opt/postgresql915/include"

For pkg-config to find postgresql915 you may need to set:
export PKG_CONFIG_PATH="/opt/homebrew/opt/postgresql915/lib/pkgconfig"

To start postgresql915 now and restart at login:
brew services start postgresql915
Or, if you don't want/need a background service you can just run:
LC_ALL=C /opt/homebrew/opt/postgresql915/bin/postgres -D /opt/homebrew/var/postgresql915
==> openjdk817
For the system Java wrappers to find this JDK, symlink it with
sudo ln -sfn /opt/homebrew/opt/openjdk817/libexec/openjdk8.jdk /Library/Java/JavaVirtualMachines/openjdk-17.jdk

openjdk817 is keg-only, which means it was not symlinked into /opt/homebrew,
because this is an alternate version of another formula.

If you need to have openjdk817 first in your PATH, run:
echo 'export PATH="/opt/homebrew/opt/openjdk817/bin:$PATH"' >> ~/.zshrc

For compilers to find openjdk817 you may need to set:
export CPPFLAGS="-I/opt/homebrew/opt/openjdk817/include"
==> gstreamer
All gstreamer plugins are now bundled in this formula.
For GStreamer to find your own plugins, add their paths to 'GST_PLUGIN_PATH'.
For example, if you have plugins in ~/.local/lib/gstreamer-1.0:
export GST_PLUGIN_PATH=~/.local/lib/gstreamer-1.0"

Do not install plugins into GStreamer's prefix. They will be deleted
by 'brew upgrade'.
(base) sunainaamin@sunaina-Air: ~ %

```



Figure 2 MQTT setup

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

3.1 Visual Studio Code installation

Fetches 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

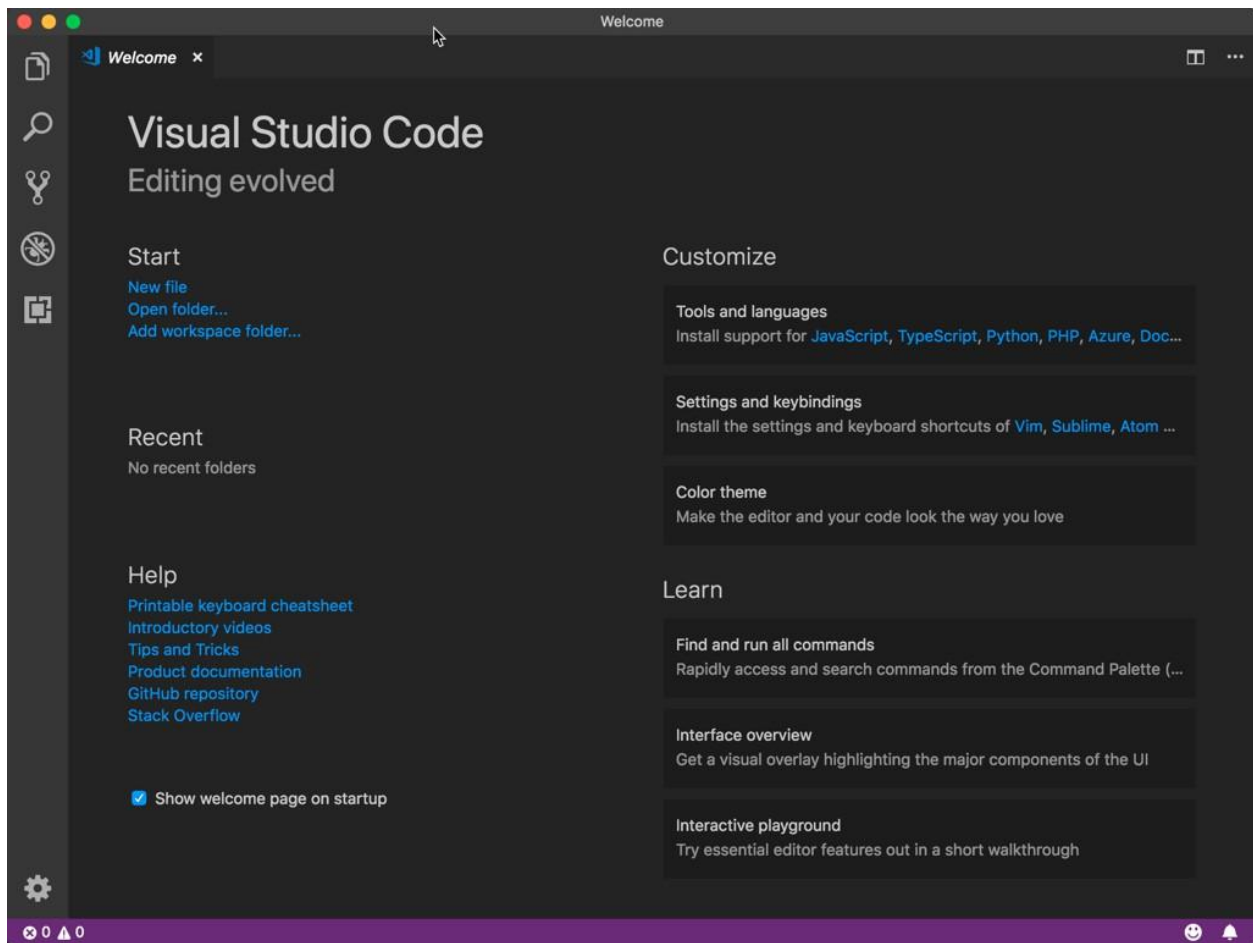


Figure 3 VS Code Installation

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
1  -----BEGIN EC PRIVATE KEY-----
2  MHQCAQEEIAwHjHbEy4pS+umq5vddFgoH8GIuq8JUScdGk6dfUSBkoAcGBSuBBAKoUQ
3  pEq89EroZ0JnJGXdDRKyMW7rsZaCU0t8i/TTPT/UqqgiPSQaV2A8i8Jb28EK1+JpwI+
4  JM2PxQ==
5  -----END EC PRIVATE KEY-----
6

```

Figure 4 ECC private key

```

🔒 public_key.pem
1  -----BEGIN PUBLIC KEY-----
2  MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEDGpjpEq89EroZ0JnJGXdDRKyMW7rsZaCU0t
3  qqqgiPSQaV2A8i8Jb28EK1+JpwI+40snYr8GkJM2PxQ==
4  -----END PUBLIC KEY-----
5

```

Figure 5 ECC public Key

```

🔒 rsa_public_key.pem
1  -----BEGIN PUBLIC KEY-----
2  MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA20oQ8UhAJFZ4jR7xLX1/
3  qxdwnXU00q11F4kQFARIhcwNu5EnPkgP8TgUcc3P01r1/U1gvNTIZkEdICYlf3/e
4  VITG9sBuwPjEDJyqSAyc0fN4whDVCS9oxS+gmCu0GsZQPJogDXRubINEjR5DpA0Z
5  VBfhS53ayC5UuBRgRZDdSx/6KFD+UsctyY+3VvTholH28whDP+9af1cGPnYkZgeD
6  5k6ZMXdepZdG+zIXa0shKRFaC7XDbYMLfW1eVfmj3uAzwdThWX6E4P22wdwxgi2S
7  lsruqMk56vpyALexNpie6tA4bkrMXYVIXx4SZApMgJjLXaYyVTnl50Fv4Z0fcuLC
8  FQIDAQAB
9  -----END PUBLIC KEY-----
10

```

Figure 6: RSA public key

🔒 rsa_private_key.pem

```
1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQEA20oQ8UhAJFZ4jR7xLX1/qxdwnXU00q11F4kQFARIhcnu5En
3 PkgP8TgUcc3P01r1/U1gvNTIZKEdICYlf3/eVITG9sBuwPjEDJyqSAyc0fN4whDV
4 CS9oxS+gmCu0GsZQPJogDXRubINEjR5DpA0ZVBfhS53ayC5UuBRgRZDdSx/6KFD+
5 UsctyY+3VvTholH28whDP+9af1cGPnYkZgeD5k6ZMXdepZdG+zIXa0shKRFaC7XD
6 bYMLfW1eVfmj3uAzwdThWX6E4P22wdwxgi2SlsruqMk56vpyALexNpie6tA4bkrM
7 XYVIXx4SZApMgJjLXaYyVTnl50Fv4Z0fcuLCFQIDAQABAoIBAD0zzugEBa3jooZY
8 mrnyE00yABi33xmewKuMcvWic7rYgWl01rrEqHAhotXhbYTDUKDYwbccgrMeau5t/
9 HC8Axni00UvdYEjEws4SxY+42cACZYU0R+MRltJxYMEEcBEBrn7LqqmVvAPnfZZ
10 AVxlszVWH9FLReuzB2R9DtImBykxfLNfvlLzZKAnvUcWnXg6+qM/l9dLS9rVz0g3
11 6k45LIHipCPsg2t0IEH8FbnhfculQlDMTbtAt9dL846M7QfvMTWP37dH4DnwhTa
12 00WQptaAge2DtJw7sfcXgHIigE0Sc6CCpft8j1GvzQPLchiAjWq9VMK3zXwT+CGc
13 rQBcIqECgYEA7M96F7nX/LMFUYN/pNnlunN0NZY80fuaDceLc+4nC7cN4z0T65P+
14 MpC0wN3jeeqs2BR//Csb9PAT/Q1YoMgJuxYbh6USFlw9EjRljpf4mKt3gyA2KTPm
15 8idZsAeLFctnvWkVdID2kQVHdkEUq30FBM6pfGzZAK1d0eX12uZAX0CgYEA6n3a
16 aBqGJulmy6YQfXQIWjkwLZKSR2LBHvn532Kn70DqtIpTGLzQGYLJpXIZPbfHcaAt
17 uRt0+9bSIXGGpntXycvgVVfP2JgirVwD3i/aFsKL3s7FaGSbE2ENxo2D6txV/cx7
18 /BGt22JwriMubqdmQCNFjE03kBrw6IXbszUipnkCgYEA2iTJ8J820sbKqGs9I0vC
19 Lr5XFk/+W0Sv5e+ii7mfaFBJT7lMkt9yc3wPtfbwvHcsn6RKva2shDbAAwTPvtT0
20 +f0EirDJ96Uic0mpf+qLr/+MCV0b8Nqp7PnIybo5KmuEddwecoL2H2NAkd0lac+x
21 1trVEBRo15ESxuoKT4VEDTUCgYEAzPY2NwU3L0zt3XjeikgRm7tKcLvy16VvUuUU
22 eP0n1CczH//Fmc0u6urI0F0nXTJuKfLeLXhVJhJqakBkSoPcIdLWPxnv6XiPBIB0
23 gA9UajzXkwJfL0UdooMc6dJ/aAjaUZE3oinJK8CcW5x19cvRE501XZLYE/78j8M
24 AeRMNLEcGYBeFq0wFdMShEbXt8yDRt7dPAUgfp9ddLjU2EaJbAepYokGzq99CmoP
25 6jIQjHA4k1GjbVN7MwRp1TZXPeyHQbo/4wvYbzEWkP/DRYHvV3UgBjukU9L25qnc
26 fUppehWLRMIPEQ1oaaSmAyYcnvW7ow9Jf7j0WHodVPc4RuYhXDpcjw==
27 -----END RSA PRIVATE KEY-----
28
```

Figure 7: RSA private key

```

generate_keys.py > ...
1  from ecdsa import SigningKey, SECP256k1
2
3  # Generate ECC private key
4  sk = SigningKey.generate(curve=SECP256k1)
5
6  # Get the corresponding public key
7  vk = sk.get_verifying_key()
8
9  # Save private key to file
10 with open("private_key.pem", "wb") as f:
11     f.write(sk.to_pem())
12
13 # Save public key to file
14 with open("public_key.pem", "wb") as f:
15     f.write(vk.to_pem())
16
17 print("ECC keys generated and saved to 'private_key.pem' and 'public_key.pem'")
18

```

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.

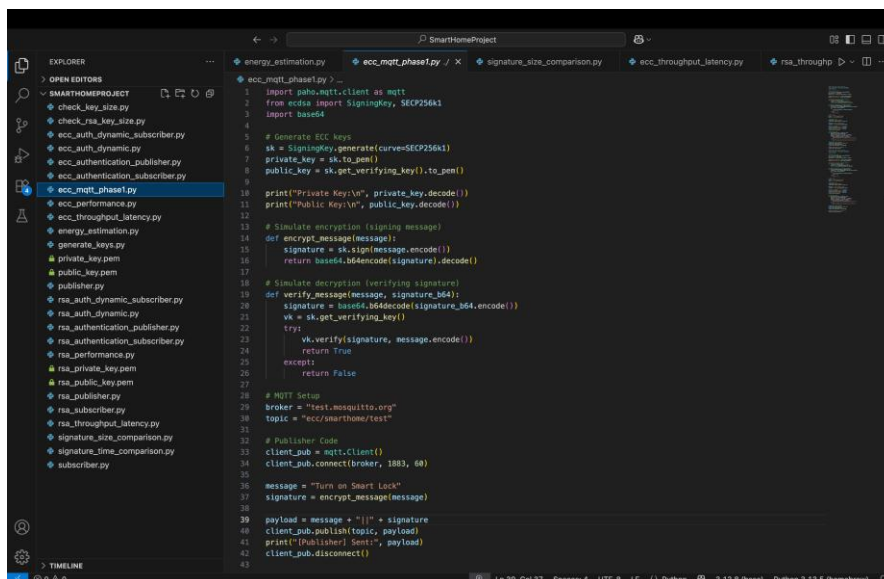


Figure 9: Dynamic Key Generation

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.verify(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
```

```

1 import paho.mqtt.client as mqtt
2 from ecdsa import VerifyingKey
3 import base64, json, time, hashlib
4
5 def on_message(client, userdata, msg):
6     payload = json.loads(msg.payload.decode())
7
8     message = payload["message"]
9     signature = base64.b64decode(payload["signature"])
10    public_key_pem = payload["public_key"].encode()
11
12    # Load the public key from received PEM
13    public_key = VerifyingKey.from_pem(public_key_pem)
14
15    # Convert message to string
16    message_str = json.dumps(message)
17
18    # Check for replay attacks
19    if abs(time.time() - message["timestamp"]) > 5:
20        print("Rejected: Message too old.")
21        return
22
23    # Signature verification
24    try:
25        start = time.perf_counter()
26        public_key.verify(signature, message_str.encode(), hashfunc=hashlib.sha256)
27        end = time.perf_counter()
28        print(f"Auth Success in {end - start:.6f}s")
29        print(f"Command:", message["command"])
30    except Exception as e:
31        print("Authentication Failed:", str(e))
32
33    # MQTT Listener setup
34    client = mqtt.Client()
35    client.connect("localhost", 1883, 60)
36    client.subscribe("ecc/auth/dynamic")
37    client.on_message = on_message
38
39    print("\n ECC Dynamic Subscriber is listening...")
40    client.loop_forever()
41
42

```

Figure 10: ECC Dynamic subscriber.py

```

1 from ecdsa import SigningKey, SECP256k1
2 import paho.mqtt.client as mqtt
3 import json, base64, time, hashlib # added hashlib for SHA-256
4
5 # Generate new ECC key pair
6 private_key = SigningKey.generate(curve=SECP256k1)
7 public_key = private_key.get_verifying_key()
8
9 # Create message
10 message = {
11     "device_id": "Device-001",
12     "timestamp": time.time(),
13     "command": "Turn on Lights"
14 }
15 message_str = json.dumps(message)
16
17 # Sign with SHA-256 to match subscriber
18 signature = private_key.sign(message_str.encode(), hashfunc=hashlib.sha256)
19 signature_b64 = base64.b64encode(signature).decode()
20
21 # Encode public key to PEM
22 public_key_pem = public_key.to_pem().decode()
23
24 # Payload
25 payload = {
26     "message": message,
27     "signature": signature_b64,
28     "public_key": public_key_pem
29 }
30
31 # Publish to MQTT
32 client = mqtt.Client()
33 client.connect("localhost", 1883, 60)
34 client.publish("ecc/auth/dynamic", json.dumps(payload))
35 print("ECC Demo: New key pair sent with message.")
36
37

```

Figure 11: ECC dynamic Publisher.py

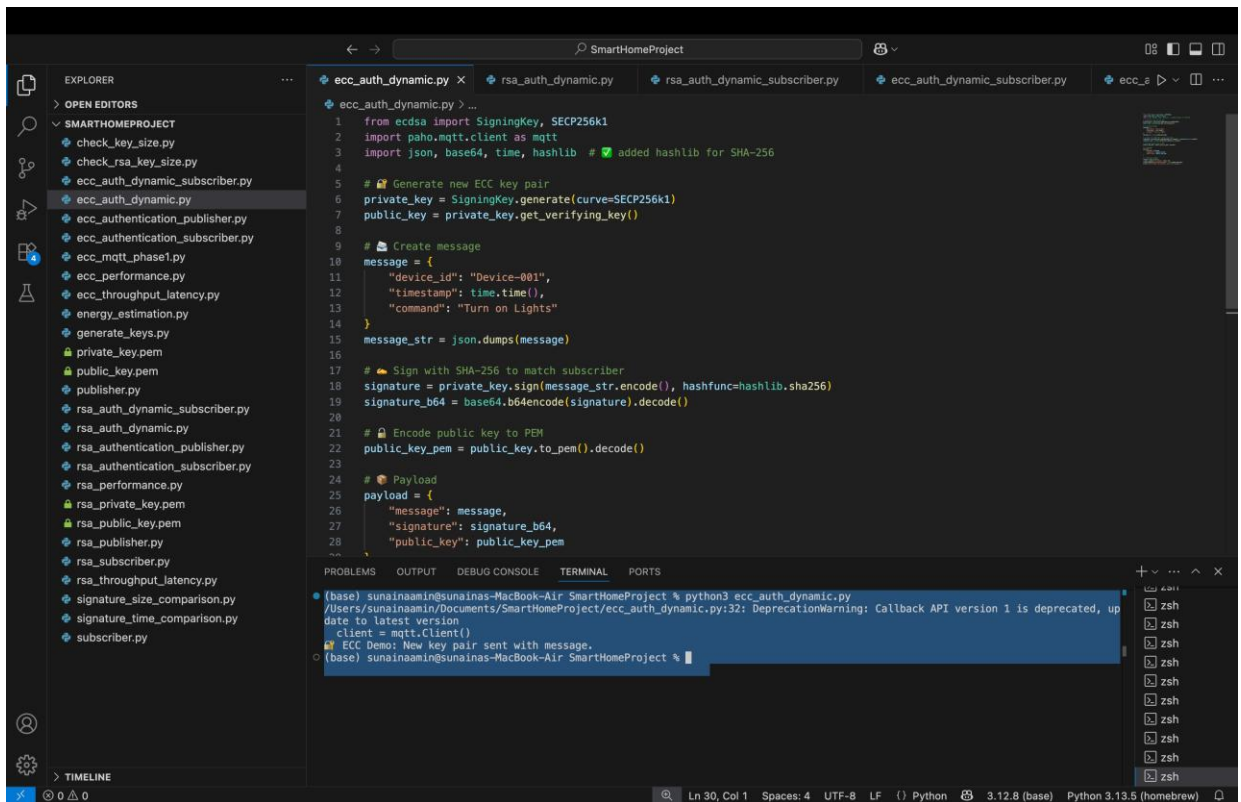


Figure 12: ECC Dynamic Publisher output

The screenshot shows the Visual Studio Code interface with the 'SmartHomeProject' open. The Explorer pane on the left lists various Python files. The main editor displays the code for 'ecc_auth_dynamic_subscriber.py'. The code imports 'paho.mqtt.client as mqtt', 'ecdsa import VerifyingKey', 'import base64, json, time, hashlib', and defines a function 'on_message' that handles incoming MQTT messages. The function decodes the message, checks for replay attacks, and verifies the signature. The terminal at the bottom shows the command 'python3 ecc_auth_dynamic_subscriber.py' and the output 'ECC Dynamic Subscriber is listening...'.

Figure 13: ECC dynamic subscriber output

The screenshot shows the Visual Studio Code interface with the 'SmartHomeProject' open. The Explorer pane on the left lists various Python files. The main editor displays the code for 'ecc_auth_dynamic.py'. The code imports 'from ecdsa import SigningKey, SECP256k1', 'import paho.mqtt.client as mqtt', 'import json, base64, time, hashlib', and defines a function 'on_message' that handles incoming MQTT messages. The function decodes the message, checks for replay attacks, and verifies the signature. The terminal at the bottom shows the command 'python3 ecc_auth_dynamic_subscriber.py' and the output 'Auth Success in 0.005839s' and 'Command: Turn on Lights'.

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

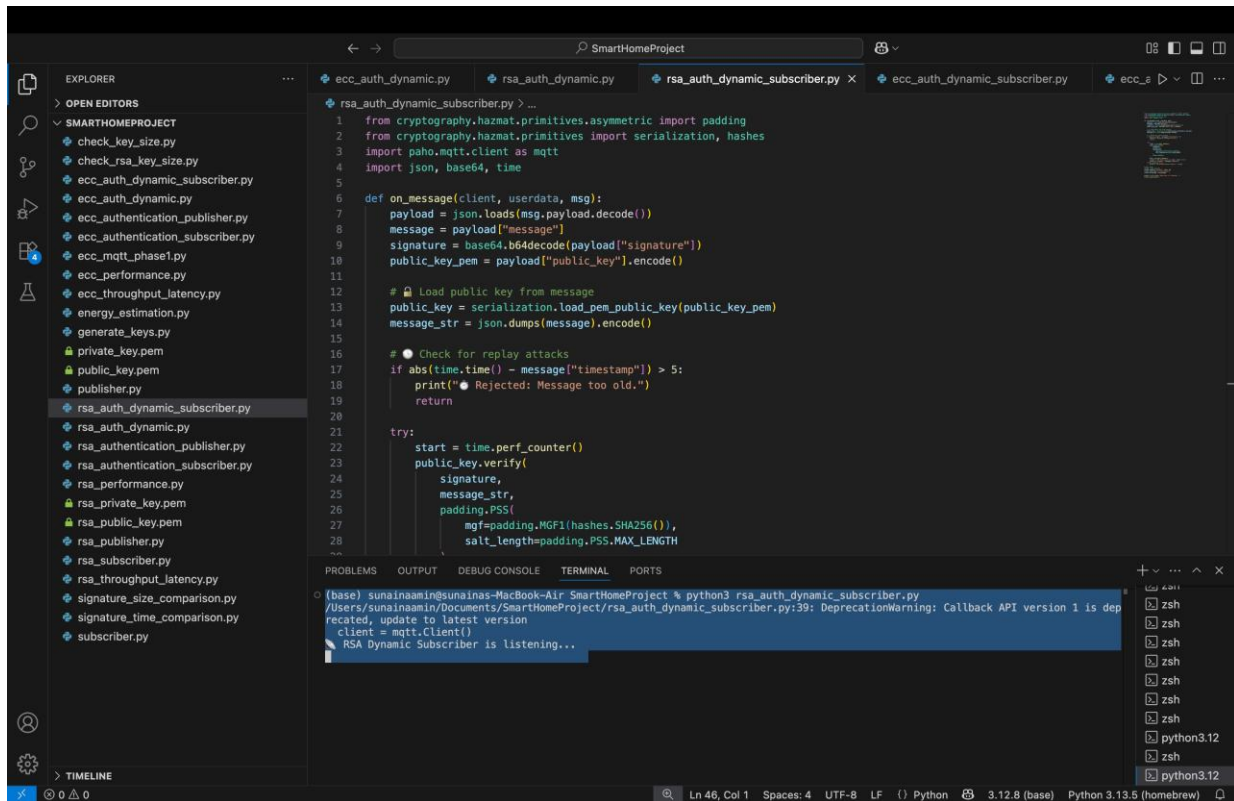


Figure 15: RSA Dynamic Subscriber

The screenshot shows the VS Code editor with the 'SmartHomeProject' workspace. The Explorer pane on the left lists various files, including 'rsa_auth_dynamic.py'. The main editor displays the code for 'rsa_auth_dynamic.py', which generates an RSA key pair, creates a message, signs it, and publishes it. The terminal at the bottom shows the command to run the script and the output, which includes a deprecation warning and a confirmation that the RSA Demo is published with a fresh key.

```

1 from cryptography.hazmat.primitives.asymmetric import rsa, padding
2 from cryptography.hazmat.primitives import serialization, hashes
3 import paho.mqtt.client as mqtt
4 import json, base64, time
5
6 # Generate new RSA key pair each time
7 private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
8 public_key = private_key.public_key()
9
10 # Create message
11 message = {
12     "device_id": "Device-001",
13     "timestamp": time.time(),
14     "command": "Turn on Lights"
15 }
16 message_str = json.dumps(message).encode()
17
18 # Sign the message
19 signature = private_key.sign(
20     message_str,
21     padding.PSS(
22         mgf=padding.MGF1(hashes.SHA256()),
23         salt_length=padding.PSS.MAX_LENGTH
24     ),
25     hashes.SHA256()
26 )
27 signature_b64 = base64.b64encode(signature).decode()
28

```

```

(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject$ python3 rsa_auth_dynamic.py
/Users/sunaina@sunaina-MacBook-Air:~/SmartHomeProject/rsa_auth_dynamic.py:43: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
RSA Demo: Published with fresh key.
(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject$

```

Figure 16: RSA dynamic Publisher

The screenshot shows the VS Code editor with the 'SmartHomeProject' workspace. The Explorer pane on the left lists various files, including 'rsa_auth_dynamic_subscriber.py'. The main editor displays the code for 'rsa_auth_dynamic_subscriber.py', which connects to the MQTT broker, receives the message, verifies the signature, and prints the command. The terminal at the bottom shows the command to run the script and the output, which includes a deprecation warning and a confirmation that the RSA Dynamic Subscriber is listening and successfully received the command 'Turn on Lights'.

```

1 from cryptography.hazmat.primitives.asymmetric import rsa, padding
2 from cryptography.hazmat.primitives import serialization, hashes
3 import paho.mqtt.client as mqtt
4 import json, base64, time
5
6 # Generate new RSA key pair each time
7 private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
8 public_key = private_key.public_key()
9
10 # Create message
11 message = {
12     "device_id": "Device-001",
13     "timestamp": time.time(),
14     "command": "Turn on Lights"
15 }
16 message_str = json.dumps(message).encode()
17
18 # Sign the message
19 signature = private_key.sign(
20     message_str,
21     padding.PSS(
22         mgf=padding.MGF1(hashes.SHA256()),
23         salt_length=padding.PSS.MAX_LENGTH
24     ),
25     hashes.SHA256()
26 )
27 signature_b64 = base64.b64encode(signature).decode()
28

```

```

(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject$ python3 rsa_auth_dynamic_subscriber.py
/Users/sunaina@sunaina-MacBook-Air:~/SmartHomeProject/rsa_auth_dynamic_subscriber.py:39: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
RSA Dynamic Subscriber is listening...
RSA Auth Success in 0.000158s
Command: Turn on Lights

```

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.

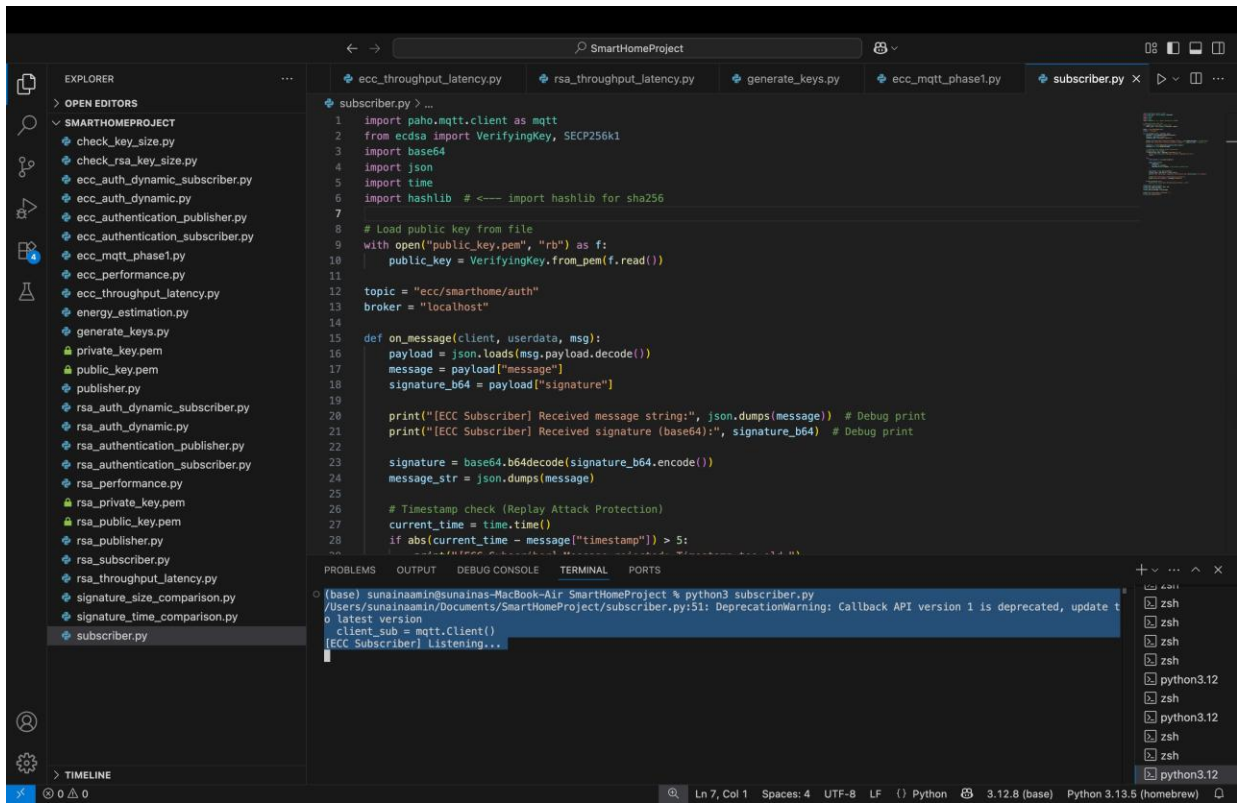
Terminal Output:

[ECC Subscriber] Listening...

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

[ECC Subscriber] Successful authentication

Command: Turn on Lights



```
1 import paho.mqtt.client as mqtt
2 from ecdsa import VerifyingKey, SECP256k1
3 import base64
4 import json
5 import time
6 import hashlib # <---- import hashlib for sha256
7
8 # Load public key from file
9 with open("public_key.pem", "rb") as f:
10     public_key = VerifyingKey.from_pem(f.read())
11
12 topic = "ecc/smarthome/auth"
13 broker = "localhost"
14
15 def on_message(client, userdata, msg):
16     payload = json.loads(msg.payload.decode())
17     message = payload["message"]
18     signature_b64 = payload["signature"]
19
20     print("[ECC Subscriber] Received message string:", json.dumps(message)) # Debug print
21     print("[ECC Subscriber] Received signature (base64):", signature_b64) # Debug print
22
23     signature = base64.b64decode(signature_b64.encode())
24     message_str = json.dumps(message)
25
26     # Timestamp check (Replay Attack Protection)
27     current_time = time.time()
28     if abs(current_time - message["timestamp"]) > 5:
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject$ python3 subscriber.py
o latest version
o client.sub = mqtt.Client()
[ECC Subscriber] Listening...
```

Figure 18: ECC subscriber.py

```

1 import paho.mqtt.client as mqtt
2 from ecdsa import SigningKey, SECP256k1
3 import base64
4 import json
5 import time
6 import hashlib # <-- added
7
8 # Load private key from file
9 with open("private_key.pem", "rb") as f:
10     private_key = SigningKey.from_pem(f.read())
11
12 public_key = private_key.get_verifying_key()
13
14 print("Loaded Private Key and Public Key from file")
15
16 # MQTT Setup
17 broker = "localhost"
18 topic = "ecc/smarthome/auth"
19
20 # Prepare secure message
21 message = {
22     "device_id": "Device-001",
23     "timestamp": time.time(),
24     "command": "Turn on Lights"
25 }
26
27 message_str = json.dumps(message)
28

```

```

(base) sunainaamingsunainas-MacBook-Air SmartHomeProject % python3 subscriber.py
/Users/sunainaamingsunainas/SmartHomeProject/subscriber.py:51: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client_sub = mqtt.Client()
[ECC Subscriber] Listening...
[ECC Subscriber] Received message string: {"device_id": "Device-001", "timestamp": 1754560481.904972, "command": "Turn on Lights"}
[ECC Subscriber] Received signature (base64): FfcMc247yql9zn0KgFVx4/VWwoKrkBbteyMh9sAVs4Ap/FqXk0tLTiuj47pvac929iA8TvwfMVSyn1Kil7PAD==
[ECC Subscriber] Signature verification took 0.005908 seconds
[ECC Subscriber] Authentication Success
Received command: Turn on Lights

```

Figure 19: ECC Successful authentication

```

1 import paho.mqtt.client as mqtt
2 from ecdsa import SigningKey, SECP256k1
3 import base64
4 import json
5 import time
6 import hashlib # <-- added
7
8 # Load private key from file
9 with open("private_key.pem", "rb") as f:
10     private_key = SigningKey.from_pem(f.read())
11
12 public_key = private_key.get_verifying_key()
13
14 print("Loaded Private Key and Public Key from file")
15
16 # MQTT Setup
17 broker = "localhost"
18 topic = "ecc/smarthome/auth"
19
20 # Prepare secure message
21 message = {
22     "device_id": "Device-001",
23     "timestamp": time.time(),
24     "command": "Turn on Lights"
25 }
26
27 message_str = json.dumps(message)
28

```

```

(base) sunainaamingsunainas-MacBook-Air SmartHomeProject % python3 publisher.py
Loaded Private Key and Public Key from file
[ECC Publisher] Signature generation took 0.000467 seconds
[ECC Publisher] Sent secure message: {"message": {"device_id": "Device-001", "timestamp": 1754560481.904972, "command": "Turn on Lights"}, "signature": "FfcMc247yql9zn0KgFVx4/VWwoKrkBbteyMh9sAVs4Ap/FqXk0tLTiuj47pvac929iA8TvwfMVSyn1Kil7PAD=="}
(base) sunainaamingsunainas-MacBook-Air SmartHomeProject %

```

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**

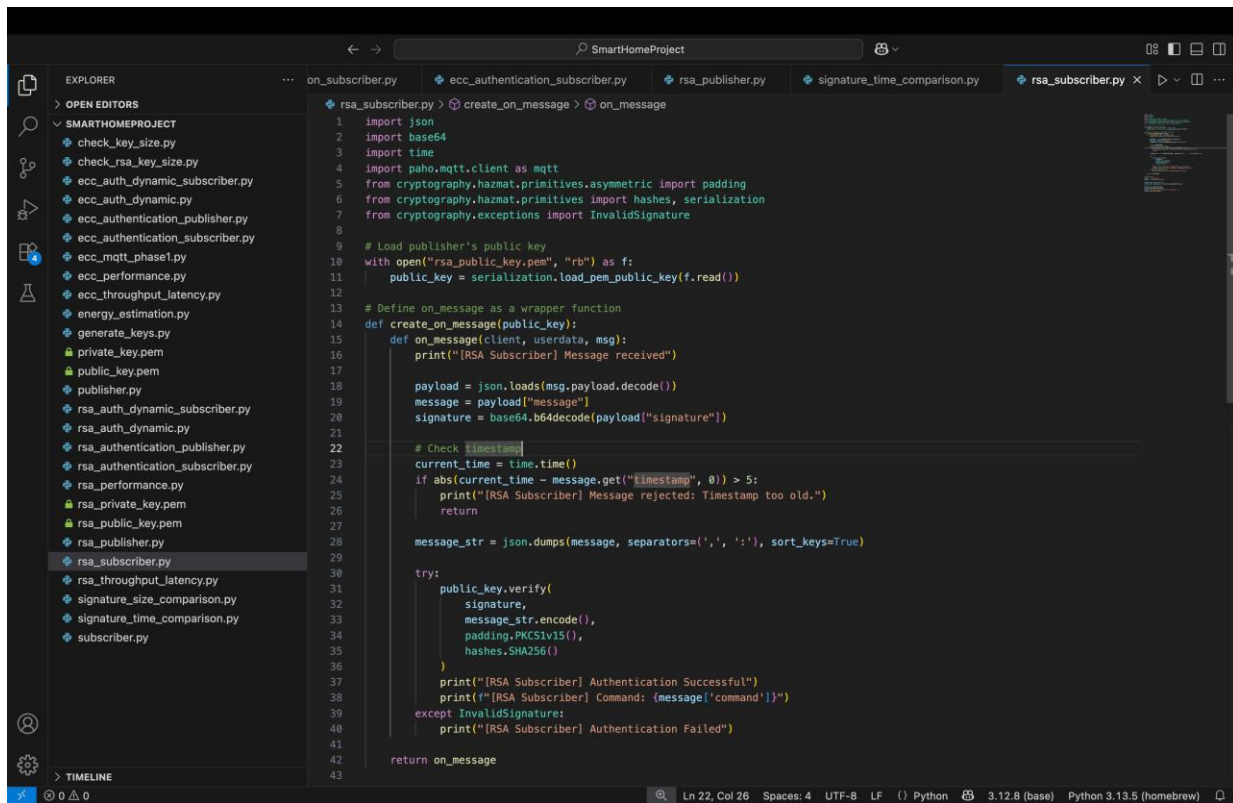
Make use of: **rsa_public_key.pem**.

Commands:

Write in Terminal

python3 rsa_publisher.py

python3 rsa_subscriber.py



```
1 import json
2 import base64
3 import time
4 import paho.mqtt.client as mqtt
5 from cryptography.hazmat.primitives.asymmetric import padding
6 from cryptography.hazmat.primitives import hashes, serialization
7 from cryptography.exceptions import InvalidSignature
8
9 # Load publisher's public key
10 with open("rsa_public_key.pem", "rb") as f:
11     public_key = serialization.load_pem_public_key(f.read())
12
13 # Define on_message as a wrapper function
14 def create_on_message(public_key):
15     def on_message(client, userdata, msg):
16         print("[RSA Subscriber] Message received")
17
18         payload = json.loads(msg.payload.decode())
19         message = payload["message"]
20         signature = base64.b64decode(payload["signature"])
21
22         # Check timestamp
23         current_time = time.time()
24         if abs(current_time - message.get("timestamp", 0)) > 5:
25             print("[RSA Subscriber] Message rejected: Timestamp too old.")
26             return
27
28         message_str = json.dumps(message, separators=(',', ':'), sort_keys=True)
29
30         try:
31             public_key.verify(
32                 signature,
33                 message_str.encode(),
34                 padding.PKCS1v15(),
35                 hashes.SHA256()
36             )
37             print("[RSA Subscriber] Authentication Successful")
38             print(f"[RSA Subscriber] Command: {message['command']}")
39         except InvalidSignature:
40             print("[RSA Subscriber] Authentication Failed")
41
42     return on_message
```

Figure 21: RSA subscriber.py

```

1 import json
2 import time
3 import base64
4 import paho.mqtt.client as mqtt
5 from cryptography.hazmat.primitives.asymmetric import rsa, padding
6 from cryptography.hazmat.primitives import hashes, serialization
7
8 # Load private key
9 with open("rsa_private_key.pem", "rb") as f:
10     private_key = serialization.load_pem_private_key(f.read(), password=None)
11
12 # MQTT Setup
13 broker = "localhost"
14 topic = "rsa/smarthome/auth"
15
16 # Prepare message
17 message = {
18     "device_id": "Device_RSA_001",
19     "timestamp": time.time(),
20     "command": "open garage door"
21 }
22
23 # Serialize message and sign it
24 message_str = json.dumps(message, separators=(',', ':'), sort_keys=True)
25 signature = private_key.sign(
26     message_str.encode(),
27     padding.PKCS1v15(),
28     hashes.SHA256()
29 )
30
31 # Encode signature in base64 for transport
32 signature_b64 = base64.b64encode(signature).decode()
33
34 # Prepare final payload
35 payload = json.dumps({
36     "message": message_str,
37     "signature": signature_b64
38 })
39
40 # Publish to MQTT

```

Figure 22: RSA publisher.py

```

1 import json
2 import base64
3 import time
4 import paho.mqtt.client as mqtt
5 from cryptography.hazmat.primitives.asymmetric import padding
6 from cryptography.hazmat.primitives import hashes, serialization
7 from cryptography.exceptions import InvalidSignature
8
9 # Load publisher's public key
10 with open("rsa_public_key.pem", "rb") as f:
11     public_key = serialization.load_pem_public_key(f.read())
12
13 # Define on_message as a wrapper function
14 def create_on_message(public_key):
15     def on_message(client, userdata, msg):
16         print("RSA Subscriber Message received")
17
18         payload = json.loads(msg.payload.decode())
19         message = payload["message"]
20         signature = base64.b64decode(payload["signature"])
21
22         # Check timestamp
23         current_time = time.time()
24         if abs(current_time - message.get("timestamp", 0)) > 5:
25             print("RSA Subscriber Message rejected: Timestamp too old.")
26             return
27
28         message_str = json.dumps(message, separators=(',', ':'), sort_keys=True)

```

Figure 23: RSA subscriber.py

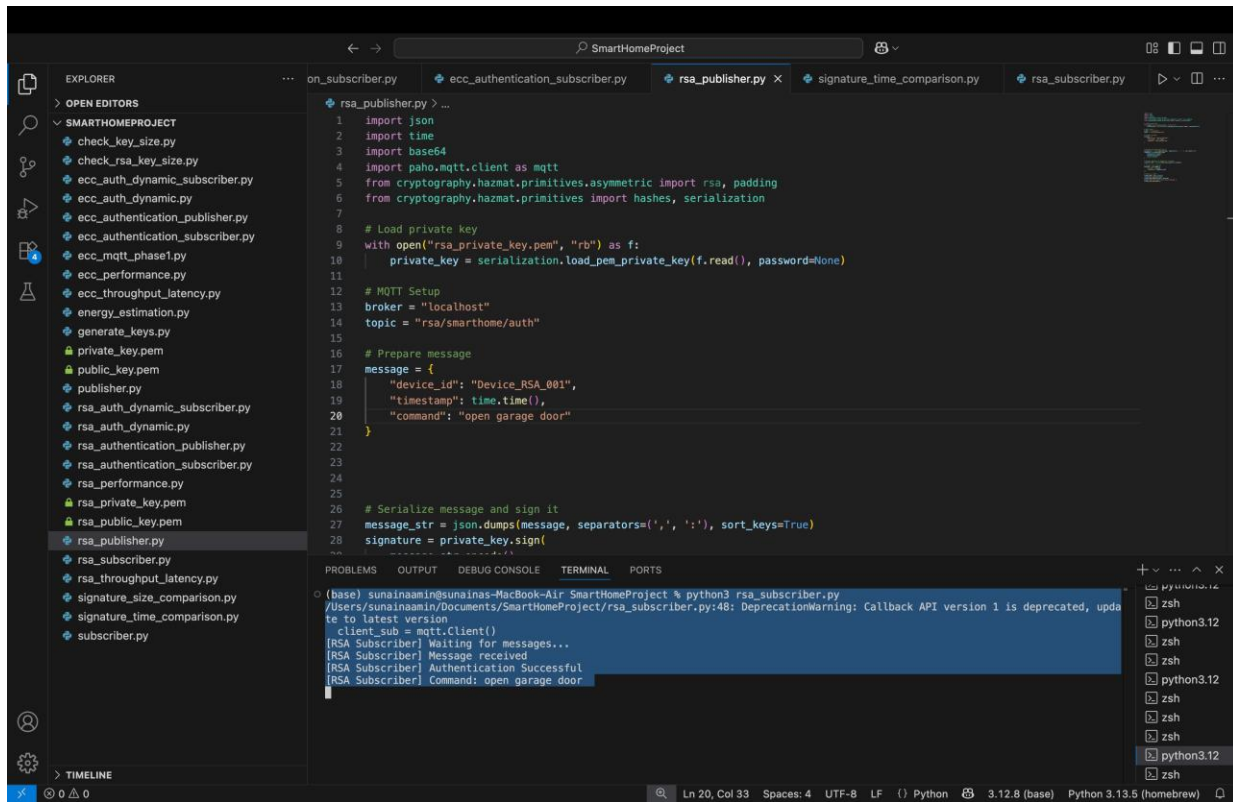


Figure 24: RSA Successful Authentication

The screenshot shows a VS Code editor window titled 'SmartHomeProject'. The Explorer panel on the left lists various files, including 'rsa_publisher.py'. The main editor area displays the code for 'rsa_publisher.py', which imports necessary modules, loads a private key, sets up an MQTT broker, and prepares a message. The terminal at the bottom shows the command executed: `python3 rsa_publisher.py`, and the output: `(base) sunaina@sunaina-MacBook-Air: SmartHomeProject % python3 rsa_publisher.py`, `DeprecationWarning: Callback API version 1 is deprecated, update to latest version`, `client_pub = mqtt.Client()`, `[RSA Publisher] Sent secure message.`

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)

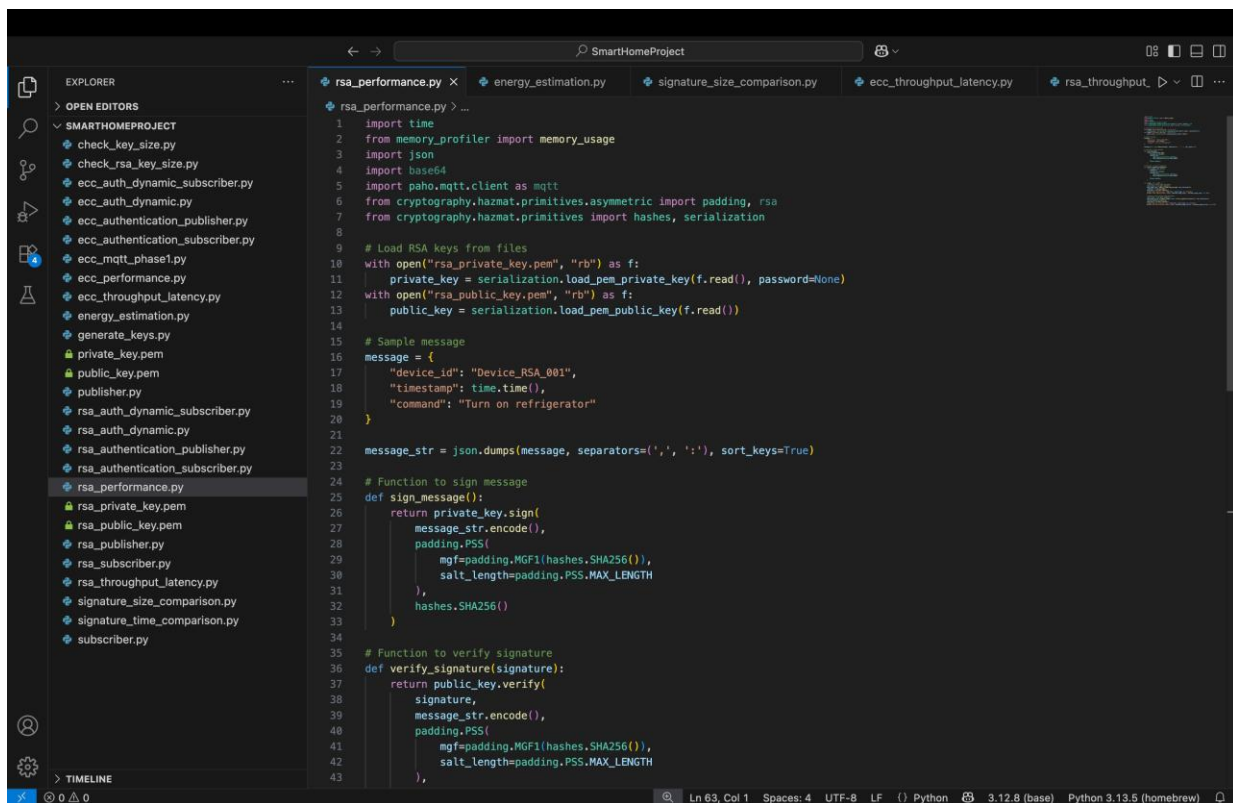
Time of Verification (in seconds)

Verification Memory Usage (in MiB)

Command to Execute:

Write in Terminal

```
python3 rsa_performance.py
```



```
1 import time
2 from memory_profiler import memory_usage
3 import json
4 import base64
5 import paho.mqtt.client as mqtt
6 from cryptography.hazmat.primitives.asymmetric import padding, rsa
7 from cryptography.hazmat.primitives import hashes, serialization
8
9 # Load RSA keys from files
10 with open("rsa_private_key.pem", "rb") as f:
11     private_key = serialization.load_pem_private_key(f.read(), password=None)
12 with open("rsa_public_key.pem", "rb") as f:
13     public_key = serialization.load_pem_public_key(f.read())
14
15 # Sample message
16 message = {
17     "device_id": "Device_RSA_001",
18     "timestamp": time.time(),
19     "command": "Turn on refrigerator"
20 }
21
22 message_str = json.dumps(message, separators=(',', ':'), sort_keys=True)
23
24 # Function to sign message
25 def sign_message():
26     return private_key.sign(
27         message_str.encode(),
28         padding.PSS(
29             mgf=padding.MGF1(hashes.SHA256()),
30             salt_length=padding.PSS.MAX_LENGTH
31         ),
32         hashes.SHA256()
33     )
34
35 # Function to verify signature
36 def verify_signature(signature):
37     return public_key.verify(
38         signature,
39         message_str.encode(),
40         padding.PSS(
41             mgf=padding.MGF1(hashes.SHA256()),
42             salt_length=padding.PSS.MAX_LENGTH
43         ),
44         hashes.SHA256()
```

Figure 26: RSA performance.py

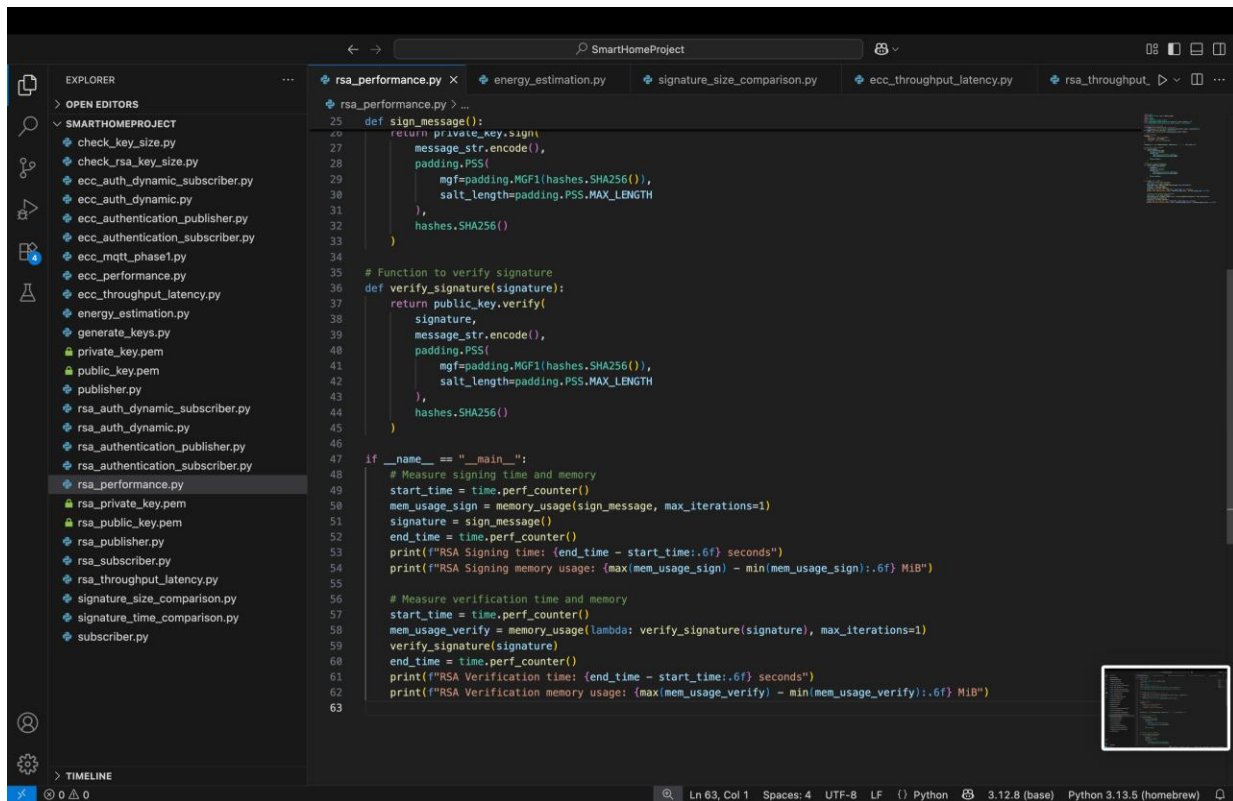


Figure 27: RSA performance code

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

```

25 def sign_message():
26     return private_key.sign(
27         message_str.encode(),
28         padding.PSS(
29             mgf=padding.MGF1(hashes.SHA256()),
30             salt_length=padding.PSS.MAX_LENGTH
31         ),
32         hashes.SHA256()
33     )
34
35 # Function to verify signature
36 def verify_signature(signature):
37     return public_key.verify(
38         signature,
39         message_str.encode(),
40         padding.PSS(
41             mgf=padding.MGF1(hashes.SHA256()),
42             salt_length=padding.PSS.MAX_LENGTH
43         ),
44         hashes.SHA256()
45     )
46
47 if __name__ == "__main__":
48     # Measure signing time and memory
49     start_time = time.perf_counter()
50     mem_usage_sign = memory_usage(sign_message, max_iterations=1)
51     signature = sign_message()
52     end_time = time.perf_counter()
53     print(f"RSA Signing time: {end_time - start_time:.6f} seconds")

```

```

(base) sunaina@sunaina-MacBook-Air: SmartHomeProject % python3 rsa_performance.py
RSA Signing time: 0.092685 seconds
RSA Signing memory usage: 0.343750 MiB
RSA Verification time: 0.007189 seconds
RSA Verification memory usage: 0.000000 MiB
(base) sunaina@sunaina-MacBook-Air: SmartHomeProject %

```

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 :

Write 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

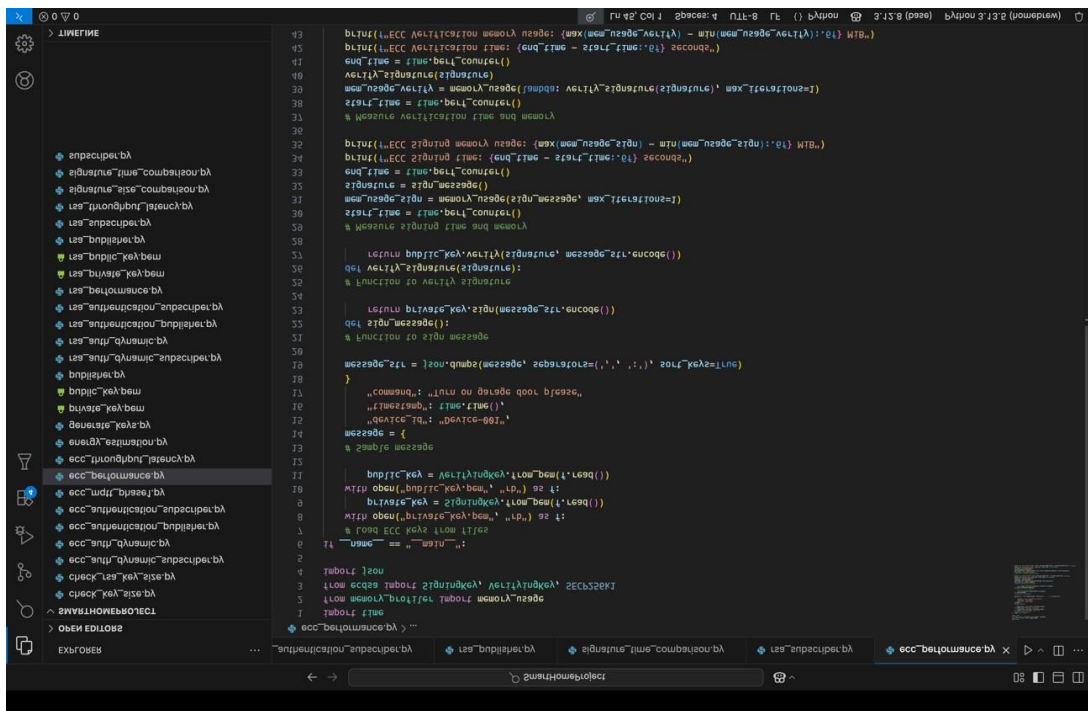


Figure 29: ECC performance.py

```
1 import time
2 from memory_profiler import memory_usage
3 from ecdsa import SigningKey, VerifyingKey, SECP256k1
4 import json
5
6 if __name__ == "__main__":
7     # Load ECC keys from files
8     with open("private_key.pem", "rb") as f:
9         private_key = SigningKey.from_pem(f.read())
10    with open("public_key.pem", "rb") as f:
11        public_key = VerifyingKey.from_pem(f.read())
12
13    # Sample message
14    message = {
15        "device_id": "Device-001",
16        "timestamp": time.time(),
17        "command": "Turn on garage door please"
18    }
19    message_str = json.dumps(message, separators=(',', ':'), sort_keys=True)
20
21    # Function to sign message
22    def sign_message():
23        return private_key.sign(message_str.encode())
24
25    # Function to verify signature
26    def verify_signature(signature):
27        return public_key.verify(signature, message_str.encode())
28
29    # Measure signing time and memory usage
30    signing_time = time.time()
31    signing_memory = memory_usage()[0]
32    signature = sign_message()
33    signing_time = time.time() - signing_time
34    signing_memory = memory_usage()[0] - signing_memory
35
36    # Measure verification time and memory usage
37    verification_time = time.time()
38    verification_memory = memory_usage()[0]
39    verify_signature(signature)
40    verification_time = time.time() - verification_time
41    verification_memory = memory_usage()[0] - verification_memory
42
43    print(f"ECC Signing time: {signing_time} seconds")
44    print(f"ECC Signing memory usage: {signing_memory} MiB")
45    print(f"ECC Verification time: {verification_time} seconds")
46    print(f"ECC Verification memory usage: {verification_memory} MiB")
```

(base) sunainaamin@sunainas-MacBook-Air SmartHomeProject % python3 ecc_performance.py
ECC Signing time: 0.056825 seconds
ECC Signing memory usage: 0.234375 MiB
ECC Verification time: 0.053235 seconds
ECC Verification memory usage: 0.015625 MiB
(base) sunainaamin@sunainas-MacBook-Air SmartHomeProject %

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:

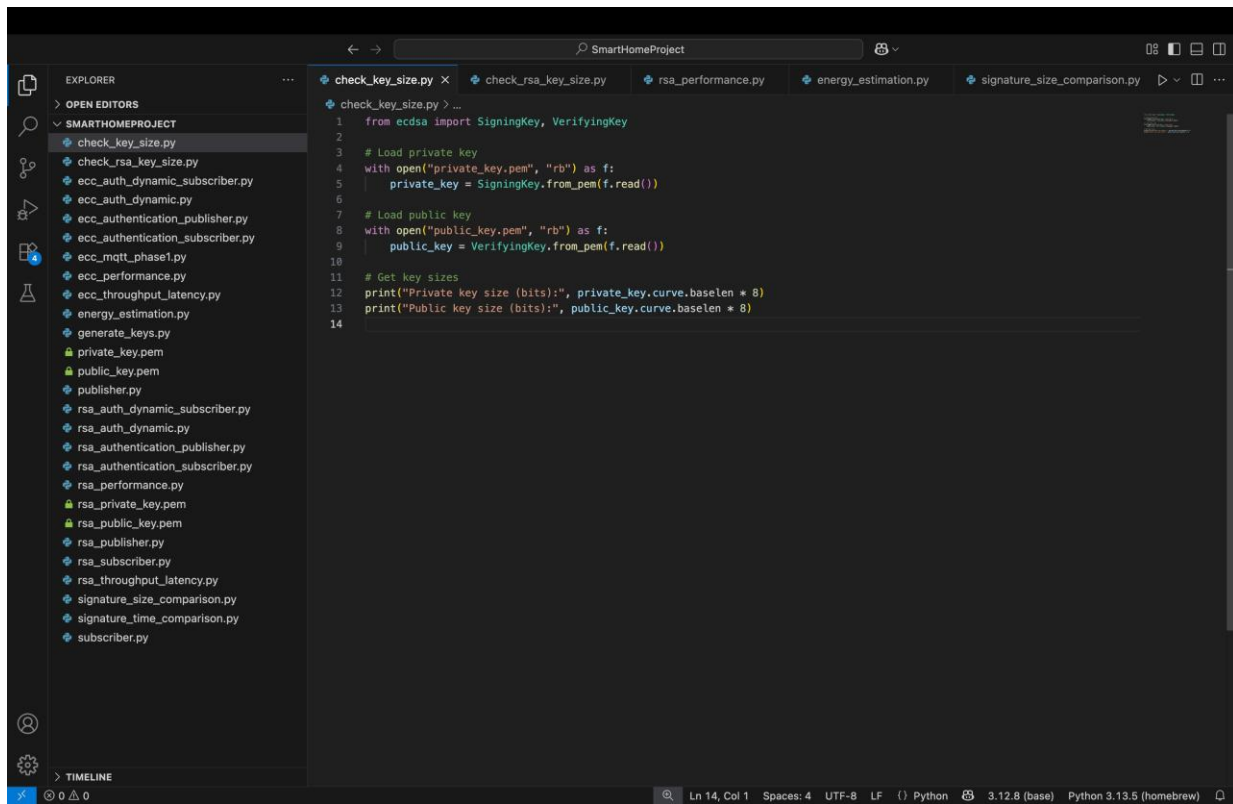
Write in Terminal

python3 check_key_size.py

Results:

Private key size (bits): 256

Public key size (bits): 256



```
1 from ecdsa import SigningKey, VerifyingKey
2
3 # Load private key
4 with open("private_key.pem", "rb") as f:
5     private_key = SigningKey.from_pem(f.read())
6
7 # Load public key
8 with open("public_key.pem", "rb") as f:
9     public_key = VerifyingKey.from_pem(f.read())
10
11 # Get key sizes
12 print("Private key size (bits):", private_key.curve.base_n * 8)
13 print("Public key size (bits):", public_key.curve.base_n * 8)
14
```

Figure 31: ECC key size check

The screenshot shows a VS Code editor window titled 'SmartHomeProject'. The Explorer sidebar on the left lists various files, including 'check_key_size.py'. The main editor area displays the code for 'check_key_size.py', which imports 'SigningKey' and 'VerifyingKey' from 'ecdsa', loads private and public keys from PEM files, and prints their sizes. The terminal at the bottom shows the command 'python3 check_key_size.py' being executed, resulting in the output: 'Private key size (bits): 256' and 'Public key size (bits): 256'.

```
1 from ecdsa import SigningKey, VerifyingKey
2
3 # Load private key
4 with open("private_key.pem", "rb") as f:
5     private_key = SigningKey.from_pem(f.read())
6
7 # Load public key
8 with open("public_key.pem", "rb") as f:
9     public_key = VerifyingKey.from_pem(f.read())
10
11 # Get key sizes
12 print("Private key size (bits):", private_key.curve.base_n * 8)
13 print("Public key size (bits):", public_key.curve.base_n * 8)
14
```

```
(base) sunainaamin@sunainas-MacBook-Air: SmartHomeProject % python3 check_key_size.py
Private key size (bits): 256
Public key size (bits): 256
(base) sunainaamin@sunainas-MacBook-Air: SmartHomeProject %
```

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

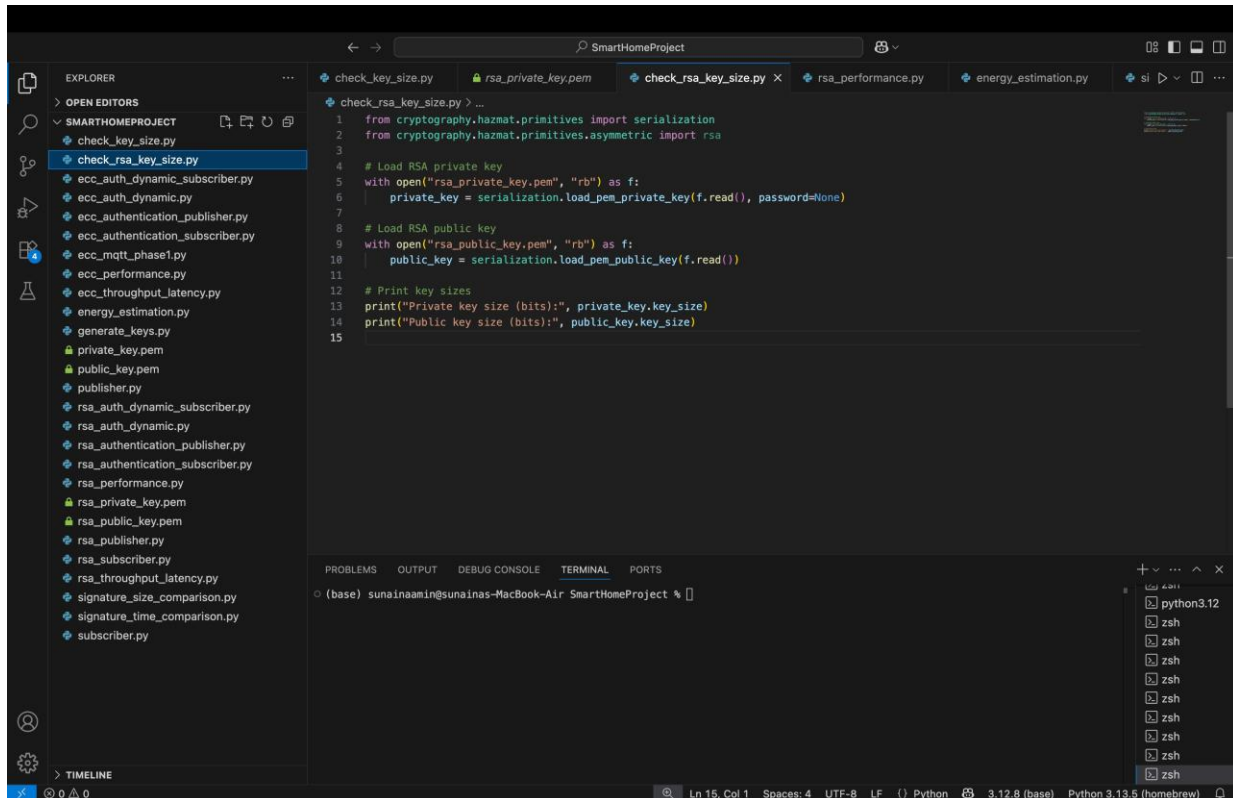
```
python3 check-rsa-key-size.py
```

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.

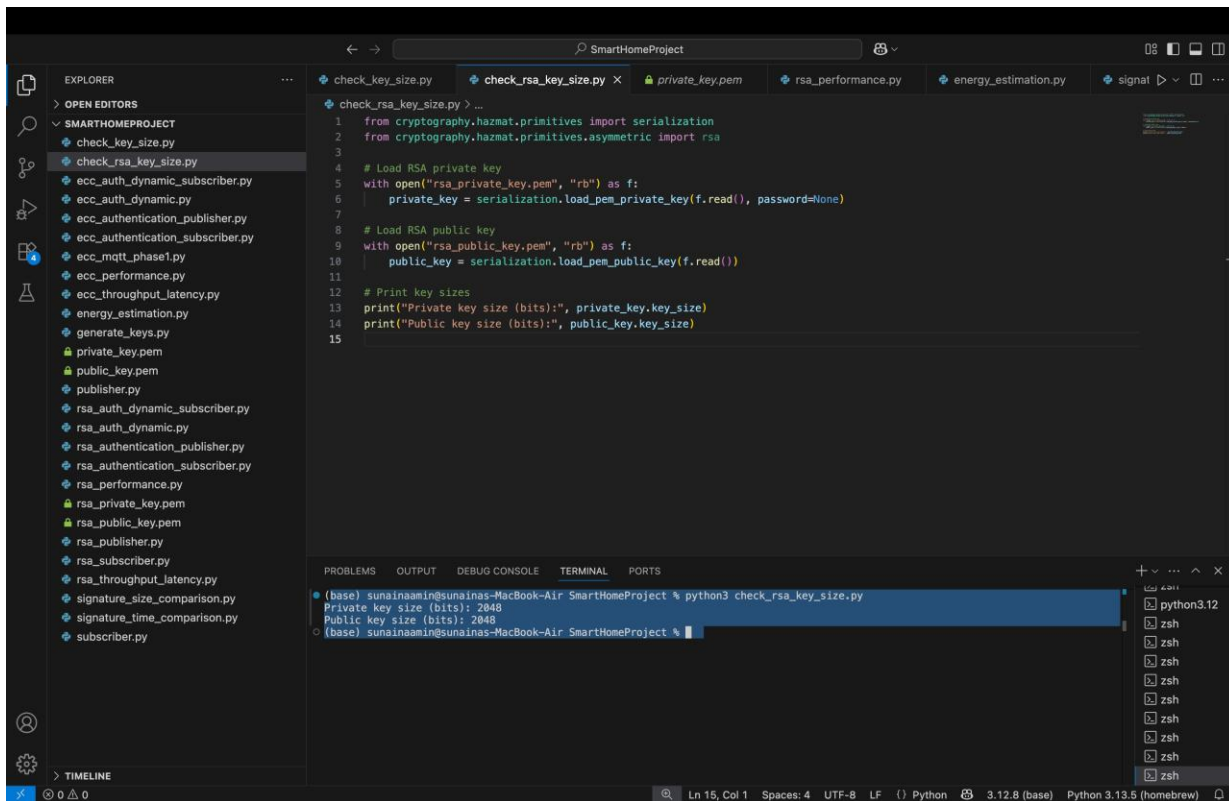


The screenshot shows a Visual Studio Code editor window with the project 'SmartHomeProject' open. The Explorer sidebar on the left lists various files, including 'check_rsa_key_size.py' which is currently selected. The main editor area displays the code for 'check_rsa_key_size.py', which imports 'serialization' and 'rsa' from 'cryptography.hazmat.primitives', loads an RSA private key from 'rsa_private_key.pem' and an RSA public key from 'rsa_public_key.pem', and then prints their key sizes. The output of the script is visible in the terminal at the bottom, showing 'Private key size (bits): 2048' and 'Public key size (bits): 2048'. The terminal also shows a list of open terminals on the right, including 'python3.12' and several 'zsh' shells.

```
1 from cryptography.hazmat.primitives import serialization
2 from cryptography.hazmat.primitives.asymmetric import rsa
3
4 # Load RSA private key
5 with open("rsa_private_key.pem", "rb") as f:
6     private_key = serialization.load_pem_private_key(f.read(), password=None)
7
8 # Load RSA public key
9 with open("rsa_public_key.pem", "rb") as f:
10    public_key = serialization.load_pem_public_key(f.read())
11
12 # Print key sizes
13 print("Private key size (bits):", private_key.key_size)
14 print("Public key size (bits):", public_key.key_size)
15
```

```
(base) sunainaam@sunainas-MacBook-Air SmartHomeProject %
python3.12
zsh
zsh
zsh
zsh
zsh
zsh
zsh
zsh
zsh
zsh
```

Figure 33: RSA key size check



```

1 from cryptography.hazmat.primitives import serialization
2 from cryptography.hazmat.primitives.asymmetric import rsa
3
4 # Load RSA private key
5 with open("rsa_private_key.pem", "rb") as f:
6     private_key = serialization.load_pem_private_key(f.read(), password=None)
7
8 # Load RSA public key
9 with open("rsa_public_key.pem", "rb") as f:
10     public_key = serialization.load_pem_public_key(f.read())
11
12 # Print key sizes
13 print("Private key size (bits):", private_key.key_size)
14 print("Public key size (bits):", public_key.key_size)
15

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

(base) sunaina@sunaina-MacBook-Air: SmartHomeProject % python3 check_rsa_key_size.py
Private key size (bits): 2048
Public key size (bits): 2048

```

Ln 15, Col 1 Spaces: 4 UTF-8 LF Python 3.12.0 (base) Python 3.13.5 (homebrew)

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

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:

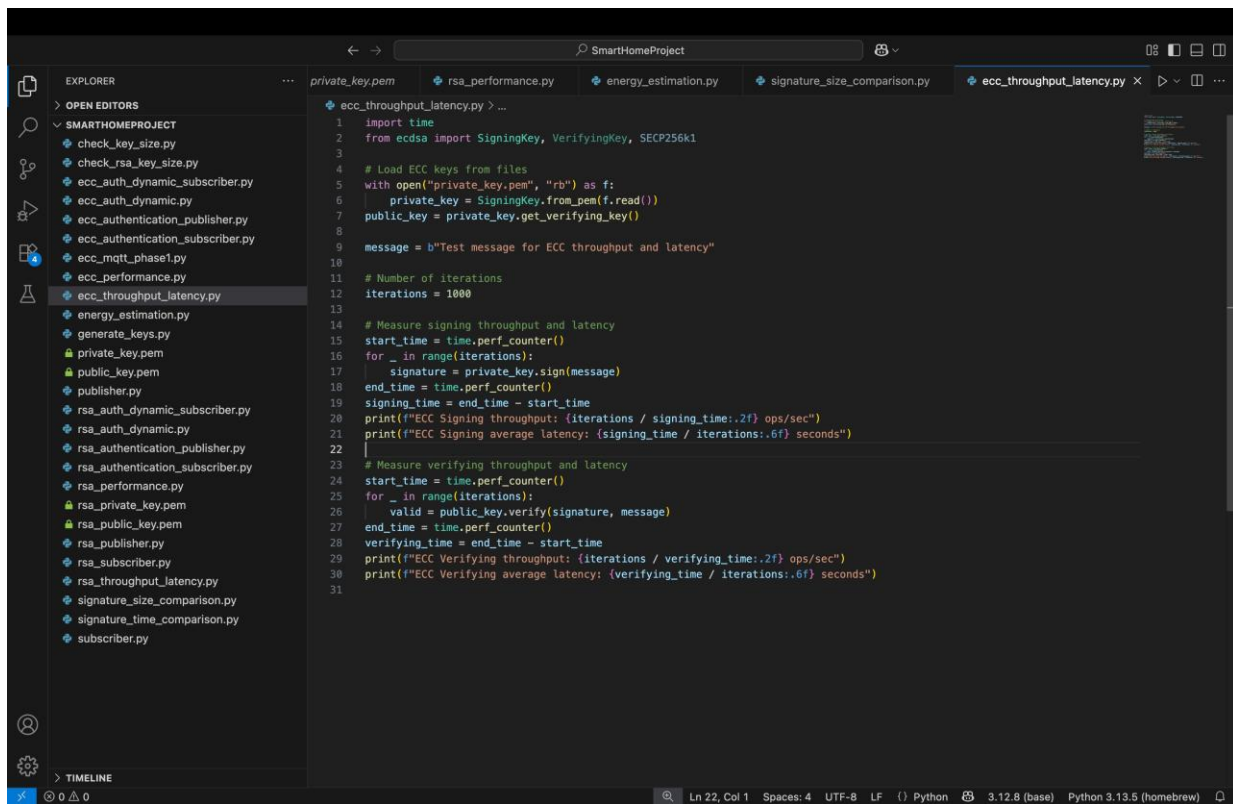
Write in Terminal

```
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.



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer lists various Python files, including `ecc_throughput_latency.py`, which is currently selected. The code editor displays the following Python code:

```
1 import time
2 from ecdsa import SigningKey, VerifyingKey, SECP256k1
3
4 # Load ECC keys from files
5 with open("private_key.pem", "rb") as f:
6     private_key = SigningKey.from_pem(f.read())
7     public_key = private_key.get_verifying_key()
8
9 message = b"Test message for ECC throughput and latency"
10
11 # Number of iterations
12 iterations = 1000
13
14 # Measure signing throughput and latency
15 start_time = time.perf_counter()
16 for _ in range(iterations):
17     signature = private_key.sign(message)
18 end_time = time.perf_counter()
19 signing_time = end_time - start_time
20 print(f"ECC Signing throughput: {iterations / signing_time:.2f} ops/sec")
21 print(f"ECC Signing average latency: {signing_time / iterations:.6f} seconds")
22
23 # Measure verifying throughput and latency
24 start_time = time.perf_counter()
25 for _ in range(iterations):
26     valid = public_key.verify(signature, message)
27 end_time = time.perf_counter()
28 verifying_time = end_time - start_time
29 print(f"ECC Verifying throughput: {iterations / verifying_time:.2f} ops/sec")
30 print(f"ECC Verifying average latency: {verifying_time / iterations:.6f} seconds")
31
```

Figure 35: ECC throughput latency

```

1 import time
2 from ecdsa import SigningKey, VerifyingKey, SECP256k1
3
4 # Load ECC keys from files
5 with open("private_key.pem", "rb") as f:
6     private_key = SigningKey.from_pem(f.read())
7     public_key = private_key.get_verifying_key()
8
9 message = b"Test message for ECC throughput and latency"
10
11 # Number of iterations
12 iterations = 1000
13
14 # Measure signing throughput and latency
15 start_time = time.perf_counter()
16 for _ in range(iterations):
17     signature = private_key.sign(message)
18 end_time = time.perf_counter()
19 signing_time = end_time - start_time
20 print(f"ECC Signing throughput: {iterations / signing_time:.2f} ops/sec")
21 print(f"ECC Signing average latency: {signing_time / iterations:.6f} seconds")
22
23 # Measure verifying throughput and latency
24 start_time = time.perf_counter()
25 for _ in range(iterations):
26     valid = public_key.verify(signature, message)
27 end_time = time.perf_counter()
28 verifying_time = end_time - start_time
29 print(f"ECC Verifying throughput: {iterations / verifying_time:.2f} ops/sec")
30 print(f"ECC Verifying average latency: {verifying_time / iterations:.6f} seconds")

```

```

(base) sunainaamingsunainas-MacBook-Air SmartHomeProject % python3 ecc_throughput_latency.py
ECC Signing throughput: 2560.84 ops/sec
ECC Signing average latency: 0.000398 seconds
ECC Verifying throughput: 662.18 ops/sec
ECC Verifying average latency: 0.001510 seconds
(base) sunainaamingsunainas-MacBook-Air SmartHomeProject %

```

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
```

```

1 import time
2 import base64
3 from cryptography.hazmat.primitives.asymmetric import padding, rsa
4 from cryptography.hazmat.primitives import hashes, serialization
5
6 # Load RSA keys from files
7 with open("rsa_private_key.pem", "rb") as f:
8     private_key = serialization.load_pem_private_key(f.read(), password=None)
9 with open("rsa_public_key.pem", "rb") as f:
10    public_key = serialization.load_pem_public_key(f.read())
11
12 message = b"Test message for RSA throughput and latency"
13
14 iterations = 1000
15
16 # Signing throughput and latency
17 start_time = time.perf_counter()
18 for _ in range(iterations):
19     signature = private_key.sign(
20         message,
21         padding.PSS(
22             mgf=padding.MGF1(hashes.SHA256()),
23             salt_length=padding.PSS.MAX_LENGTH,
24         ),
25         hashes.SHA256()
26     )
27 end_time = time.perf_counter()
28 signing_time = end_time - start_time
29 print(f"RSA Signing throughput: (iterations / signing_time:.2f) ops/sec")
30 print(f"RSA Signing average latency: (signing_time / iterations:.6f) seconds")
31
32 # Verifying throughput and latency
33 start_time = time.perf_counter()
34 for _ in range(iterations):
35     valid = public_key.verify(
36         signature,
37         message,
38         padding.PSS(
39             mgf=padding.MGF1(hashes.SHA256()),
40             salt_length=padding.PSS.MAX_LENGTH,
41         ),
42         hashes.SHA256()
43     )

```

Figure 37: RSA throughput latency

```

(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject % python3 rsa_throughput_latency.py
RSA Signing throughput: 2466.78 ops/sec
RSA Signing average latency: 0.00045 seconds
RSA Verifying throughput: 59534.15 ops/sec
RSA Verifying average latency: 0.000017 seconds

```

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
```

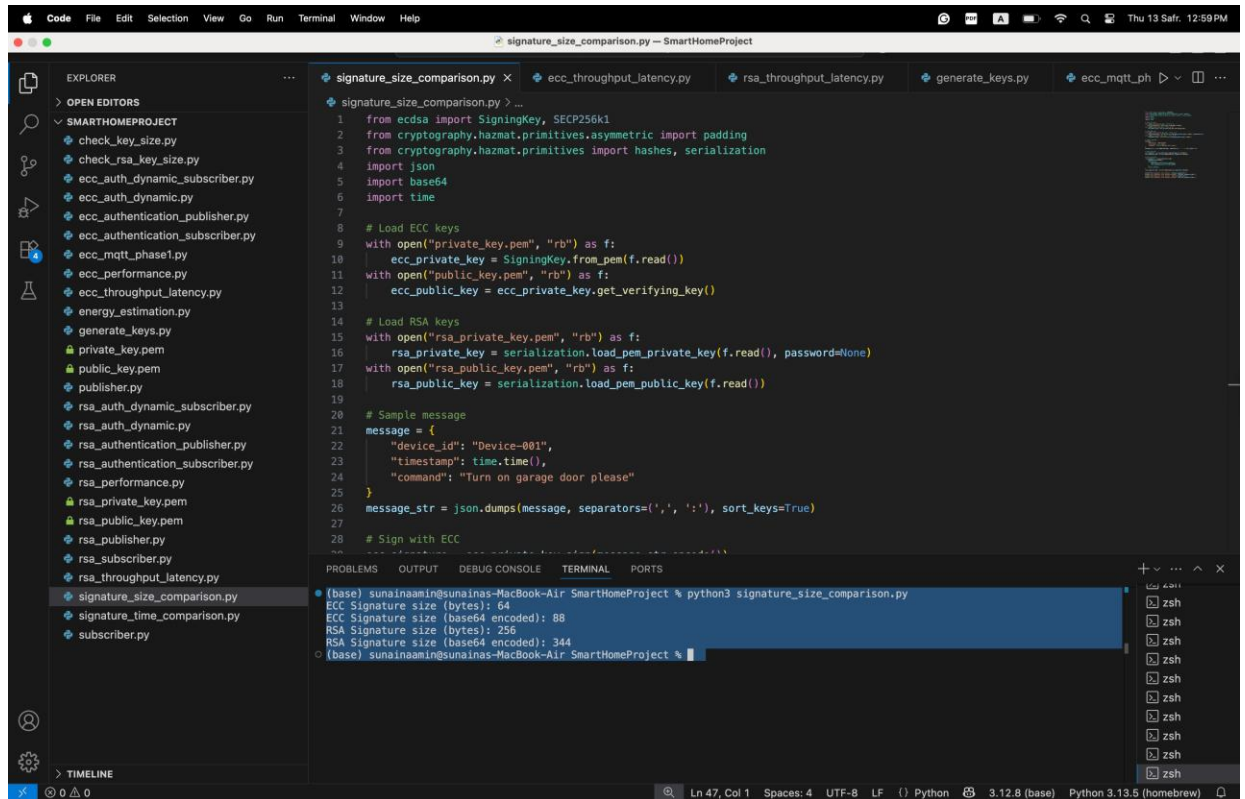


Figure 39: Signature size comparison for ECC and RSA

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

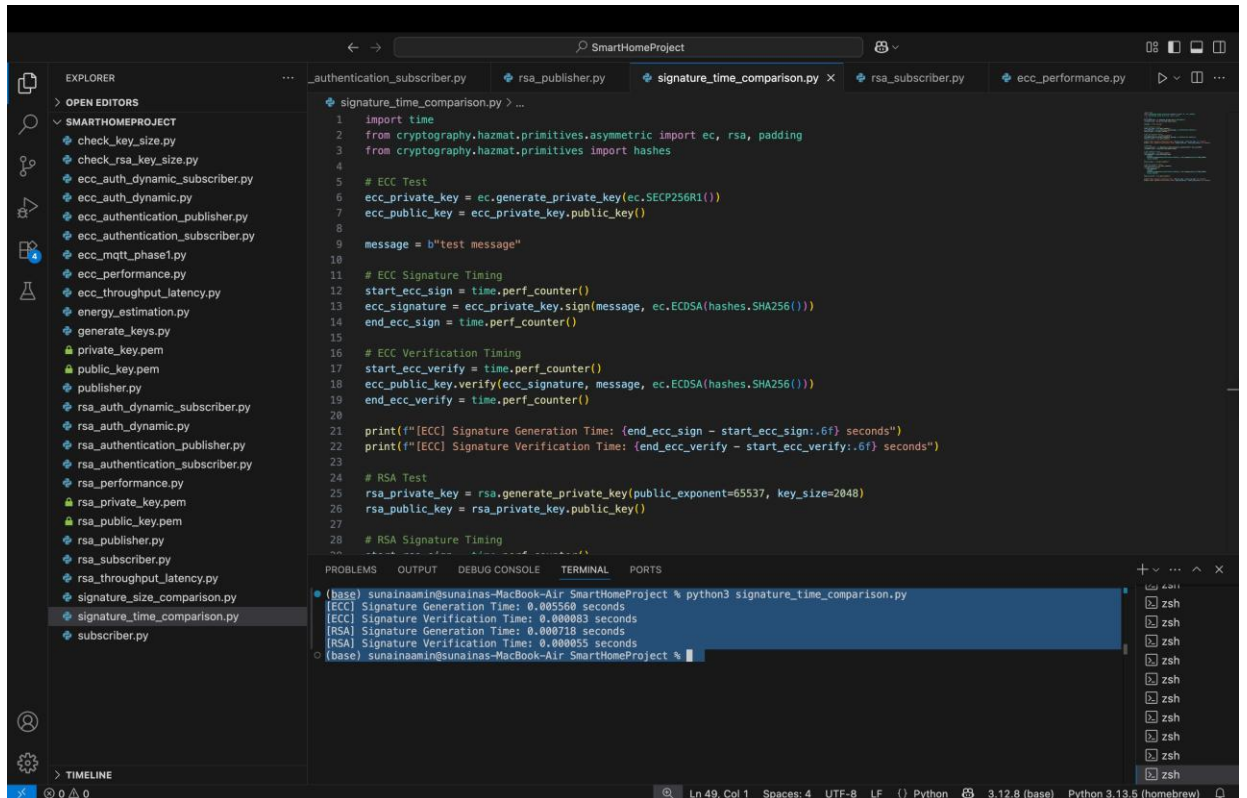
Key Insights:

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.



The screenshot shows a VS Code editor with a project named 'SmartHomeProject'. The Explorer panel on the left lists various files, including 'signature_time_comparison.py'. The main editor window displays the code for this file, which compares the performance of ECC and RSA for signature generation and verification. The code includes imports for time, cryptography primitives, and hashes. It generates ECC and RSA keys, signs a message, and then verifies the signature, timing each step. The terminal at the bottom shows the output of running the script, indicating that ECC signature generation took 0.005560 seconds, ECC verification took 0.000003 seconds, RSA signature generation took 0.000718 seconds, and RSA verification took 0.000055 seconds.

```
1 import time
2 from cryptography.hazmat.primitives.asymmetric import ec, rsa, padding
3 from cryptography.hazmat.primitives import hashes
4
5 # ECC Test
6 ecc_private_key = ec.generate_private_key(ec.SECP256R1())
7 ecc_public_key = ecc_private_key.public_key()
8
9 message = b"test message"
10
11 # ECC Signature Timing
12 start_ecc_sign = time.perf_counter()
13 ecc_signature = ecc_private_key.sign(message, ec.ECDSA(hashes.SHA256()))
14 end_ecc_sign = time.perf_counter()
15
16 # ECC Verification Timing
17 start_ecc_verify = time.perf_counter()
18 ecc_public_key.verify(ecc_signature, message, ec.ECDSA(hashes.SHA256()))
19 end_ecc_verify = time.perf_counter()
20
21 print(f"[ECC] Signature Generation Time: {end_ecc_sign - start_ecc_sign:.6f} seconds")
22 print(f"[ECC] Signature Verification Time: {end_ecc_verify - start_ecc_verify:.6f} seconds")
23
24 # RSA Test
25 rsa_private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
26 rsa_public_key = rsa_private_key.public_key()
27
28 # RSA Signature Timing
29 start_rsa_sign = time.perf_counter()
30 rsa_signature = rsa_private_key.sign(message, padding.PKCS1v15(), hashes.SHA256())
31 end_rsa_sign = time.perf_counter()
32
33 start_rsa_verify = time.perf_counter()
34 rsa_public_key.verify(rsa_signature, message, padding.PKCS1v15(), hashes.SHA256())
35 end_rsa_verify = time.perf_counter()
36
37 print(f"[RSA] Signature Generation Time: {end_rsa_sign - start_rsa_sign:.6f} seconds")
38 print(f"[RSA] Signature Verification Time: {end_rsa_verify - start_rsa_verify:.6f} seconds")
```

```
(base) sunaina@sunaina-MacBook-Air:~/SmartHomeProject % python3 signature_time_comparison.py
[ECC] Signature Generation Time: 0.005560 seconds
[ECC] Signature Verification Time: 0.000003 seconds
[RSA] Signature Generation Time: 0.000718 seconds
[RSA] Signature Verification Time: 0.000055 seconds
```

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:

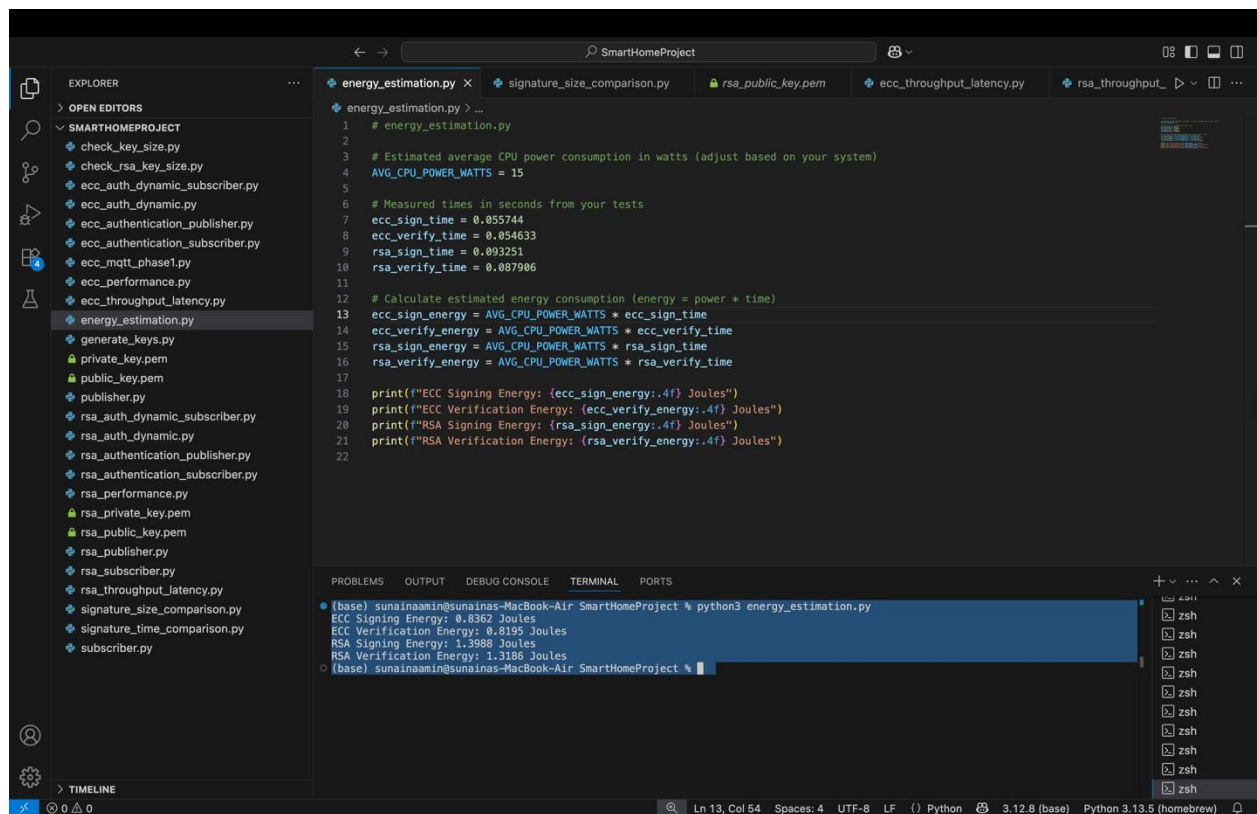
Operation	ECC Energy (J)	RSA Energy (J)
Signing	0.3026	0.5034
Verification	0.0896	1.3180

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.



The screenshot shows a VS Code editor window with the file explorer on the left and the editor on the right. The file explorer shows a project named 'SMARTHOMEPROJECT' with various Python files. The editor displays the 'energy_estimation.py' file, which contains the following code:

```
1 # energy_estimation.py
2
3 # Estimated average CPU power consumption in watts (adjust based on your system)
4 AVG_CPU_POWER_WATTS = 15
5
6 # Measured times in seconds from your tests
7 ecc_sign_time = 0.055744
8 ecc_verify_time = 0.054633
9 rsa_sign_time = 0.093251
10 rsa_verify_time = 0.087906
11
12 # Calculate estimated energy consumption (energy = power * time)
13 ecc_sign_energy = AVG_CPU_POWER_WATTS * ecc_sign_time
14 ecc_verify_energy = AVG_CPU_POWER_WATTS * ecc_verify_time
15 rsa_sign_energy = AVG_CPU_POWER_WATTS * rsa_sign_time
16 rsa_verify_energy = AVG_CPU_POWER_WATTS * rsa_verify_time
17
18 print(f"ECC Signing Energy: {ecc_sign_energy:.4f} Joules")
19 print(f"ECC Verification Energy: {ecc_verify_energy:.4f} Joules")
20 print(f"RSA Signing Energy: {rsa_sign_energy:.4f} Joules")
21 print(f"RSA Verification Energy: {rsa_verify_energy:.4f} Joules")
22
```

The terminal at the bottom shows the output of the script:

```
(base) sunainaaming@sunainas-MacBook-Air SmartHomeProject % python3 energy_estimation.py
ECC Signing Energy: 0.8362 Joules
ECC Verification Energy: 0.8199 Joules
RSA Signing Energy: 1.3988 Joules
RSA Verification Energy: 1.3186 Joules
(base) sunainaaming@sunainas-MacBook-Air SmartHomeProject %
```

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