UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

UNIVERSITÀ DEGLI STUDI DI PADOVA

MASTER DEGREE IN CYBERSECURITY

COURSE: CYBER-PHYSICAL SYSTEMS AND IOT SECURITY

Report for the Course on Cyber-Physical Systems and IoT Security

Reference Paper: **Authentication of IoT Device and IoT Server Using Secure Vaults**

Report and Project:

Jacopo Momesso 2123874

Academic year 2024/2025

Contents

1 Objectives

The reference paper "Authentication of IoT Device and IoT Server Using Secure Vaults" discusses discusses the implementation of a new multi-key based mutual authentication mechanism, where the multi-keys are called a secure vault, i.e. a collection of equal-sized keys.
The aim of the project is to implement the above mentioned authentication mechanism for the authentication of IoT devices. In order to implement the mechanism, we therefore need to simulate:

- an IoT device
- a server through which the device needs to authenticate itself

2 System Setup

The tool I used in this project is Python (version 3.12.8, but python 3.11 should also be acceptable), and three of its libraries:

- PyCryptdome, which is used for the encryption with the AES algorithm
- PrettyTable, which is used to display the result in a well-formatted table
- ecdsa, which is used for the encryption in the simulation of the ECC-based authentication mechanism, which is used for comparison with the multi-key-based mechanism.

With Python installed, to run the simulator, the user should first of all install the above mentioned libraries using the following commands:

```
pip install pycryptodome prettytable ecdsa
```

Once the libraires are installed, to run the program should be sufficient to open the terminal and type:

```
python3 Multidevices_AES_Authentication.py
```

or

```
python Multidevices_AES_Authentication.py
```

If it doesn't work, I created a virtual Python environment in my OS using the following terminal command:

```
python -m venv /path/to/new/virtual/environment
```

Once the virtual environment was created, I installed the libraries with the command above, and then I activated it using:

```
source ./path/to/new/virtual/environment/bin/activate
```

Finally, running the simulator should not give any errors. Even if there are two different files in the folder (Multidevices_AES_Authentication.py and Multidevices_ECC_Authentication.py), launching the AES one will automatically launch also the other. Therefore in the terminal will be displayed the results of both programs.

3 Experiments

The main simulator (Multidevices_AES_Authentication.py) has 3 different classes that represent:

- the secure vault
- the server
- the IoT device

Along with the 3 classes, there's the main function, which creates the server and 10 devices to connect to it. Then it starts the simulation and when it's finished it displays the results in a table. After the main one has created the result table, it starts the second simulator (Multidevices_ECC_Authentication.py). Despite the main simulator, Multidevices_ECC_Authentication.py has two classes that represent:

- the server
- the IoT device

In addition to the two classes, there's a simulate function, which simulates the authentication mechanism, and the main function, which creates the server and 10 devices, connects them to the server, starts the simulation and then prints the results in a table.
(Each file has comments that explain the code)

3.1 Multidevices_AES_Authentication.py

3.1.1 SecureVault class

```
class SecureVault:
    """
    Class representing secure cryptographic key storage and management system.
    Implements dynamic key updates using HMAC-based key derivation
    """

    def __init__(self, n=N, m=M):…

    def key_xor(self, indices):…

    def update_vault(self, data):…
```

Figure 1: SecureVault class

The SecureVault class represents the multi-keys that will be used by server and devices to authenticate the themself. The class has a constructor with paramenters:

- N: number of keys in the secure vault
- M: size of each key in bytes (e.g. 16 means that the keys are 128 bits long)

The constructor generates N random keys of M size. The *key_xor()* function computes the bitwise xor between the keys corresponding to the indices passed as argument to the function. Finally, the *update_vault()* function updates the secure vault based on the data passed as argument.

3.1.2 IoTServer class

```python
class IoTServer(threading.Thread):
    """
    Class representing a server able to handle multiple IoT devices.
    Manages secure sessions and cryptographic challenges
    """

    def __init__(self):…

    def register_device(self, device):…

    def is_valid_session(self, device_id, session_id):…

    def generate_challenge(self, device_id):…

    def validate_challenge_response(self, device_id, encrypted_response):…

    def generate_final_response(self, device_id):…

    def finalize_auth(self, device_id):…
```

Figure 2: Server class

The IoTServer class is a simplified representation of a server that manages the authentication of IoT devices through a simplified implementation of the session, sending challenges, validating them and updating the vault once authentication is complete.
The *register_device()* function connects the device passed as an argument to the server and stores it in a dictionary that keeps track of the devices connected to the server and each device's vault.
The *is_valid_session()* function checks whether the session sent by a device at the start of authentication is valid or not.
The *generate_challenge()* function generates a random challenge (*c1*: array of integers between 1 and N since the integers represent the indicies of the vault's elements that have to be xored) and a random integer, called nonce (*r1*), after a device has started authentication with a valid session id.
The *validate_challenge_response()* function checks whether the device's response to the first challenge is correct or not. The validation consists of decoding the response, retrieving the values and finally comparing the *r1* received with the *r1* previously generated by the server.
The *generate_final_response()* function generates the final message of the authentication mechanism after retrieving the values from the device response and encrypting the final message.
The *finalize_auth()* function updates the vault once authentication is complete and synchronises the device vault with the new one.

3.1.3 IoTDevice class

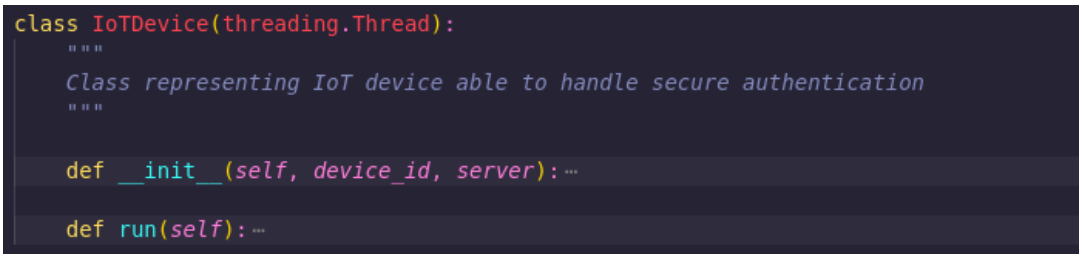

Figure 3: IoT device class

The IoTDevice class is a simplified representation of an IoT device that asks to a server to be authenticated. The class has constructor that creates the device generating a vault and requesting the server to connect itself sending the vault.
The *run()* function simulates the authentication mechanism, in fact the class uses threads, so when a device object is created and started in the main, the *run()* function starts the authentication. The authentication starts by sending the session (a random integer that must be between 1 and 100) to the server. The server validates it and then sends the challenge, which is processed by the device. The device gets the values sent by the server and it generates a key ($k1$), a nonce ($t1$), a new challenge ($c2$) and another nonce ($r2$). Then the device concatenates all these values and it encrypts them. Therefore, the device responds to the server with the new challenge and the server generates the final response. When the device receives the final response, it checks that it meets the condition and tells the server to update the vault, thereby completing the authentication.

3.2 Multidevices_ECC_Authentication.py

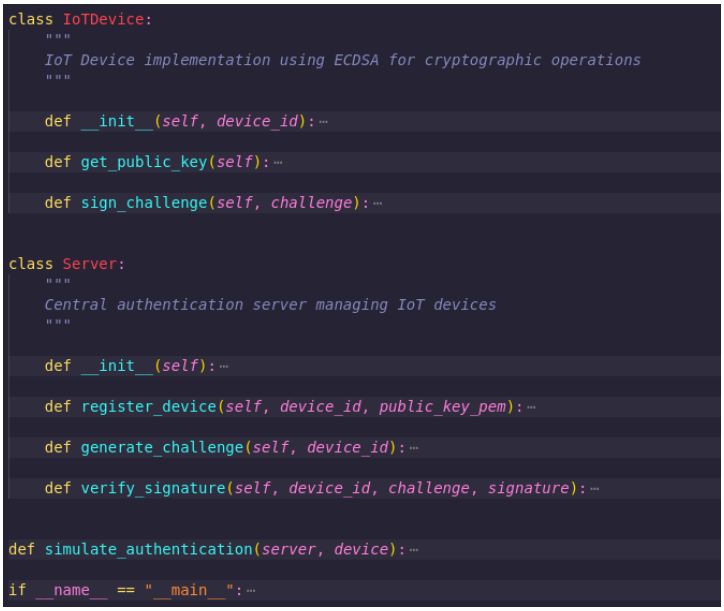

Figure 4: MultiDevices_ECC_Authentication.py

The Multidevices_ECC_Authentication.py simulates an IoT authentication system using ECDSA (Elliptic Curve Digital Signature Algorithm), a cryptographic method based on ECC (Elliptic Curve Cryptography). ECC is a public-key cryptography approach that offers strong security with smaller key sizes, making it ideal for resource-constrained IoT devices. The script creates multiple IoT devices, each with a unique ECDSA key pair, and a central server that registers their public keys. The server generates cryptographic challenges for each device, which sign and return them for verification. The script measures execution times for challenge generation, signing, and verification in order to analyze authentication performance. The results are displayed in a table, along with average times for successful authentications.
This script serves as a comparison to the main one (MultiDevices_AES_Authentication.py) to demonstrate its efficiency with respect to ECC-based authentication.

4 Results and Discussion

The output after running the main script is shown in Figure 5. As we can see from the output, the results are quite different from those presented in the reference paper. In fact, we found that our results for multi-key AES authentication are below the ms. This is probably due to the fact that the results in the paper are computed using the authentication mechanism with Arduino, while our results are computed using a PC, which has a more powerful CPU than the Arduino. Therefore, hardware differences may explain discrepancies in the results. The same differences can be seen for ECC-based authentication, and again, they are likely to be due to differences in hardware computing power. However, even with these discrepancies, it's clear that the multi-key based authentication mechanism is better than the ECC based one, in fact it's more or less 10 times faster than the rival, showing that even with different hardware it's still a faster solution.

4.1 Limitations

It's important to note that the script simulates an IoT device and a server within the same computer, which results in completely removing the latency that exists in a real-world scenario due to the network. Furthermore, the simulator only keeps track of time, while the paper also highlights the energy consumption of the authentication mechanism. Obviously, using a script in a computer doesn't allow the calculation of energy consumption, so we can't compare the two measures, but still, since the time of the multi-key authentication mechanism is lower than the ECC one, we can think that the energy consumption will probably also be lower.

```
AES-128 Authentication Performance Metrics:
+-----------+--------------+--------------+-------------------+---------+
| Device ID | Encrypt (ms) | Decrypt (ms) | Vault Update (ms) |  Status |
+-----------+--------------+--------------+-------------------+---------+
|  Device01 |    1.5845    |    0.1276    |       1.0839      | Success |
|  Device02 |    0.1926    |    0.0281    |       0.0975      | Success |
|  Device03 |    0.0725    |    0.0706    |       0.9744      | Success |
|  Device04 |    0.5293    |    0.0217    |       0.6678      | Success |
|  Device05 |    0.0405    |    0.0291    |       1.9641      | Success |
|  Device06 |    0.1612    |    0.0238    |       0.8667      | Success |
|  Device07 |    0.0975    |    0.2124    |       1.2391      | Success |
|  Device08 |    0.0732    |    0.2229    |       0.7226      | Success |
|  Device09 |    0.1757    |    0.1018    |       0.6461      | Success |
|  Device10 |    0.0877    |    0.0417    |       0.1855      | Success |
+-----------+--------------+--------------+-------------------+---------+

Average Operation Times:
+--------------+-------------------+
|    Metric    | Average Time (ms) |
+--------------+-------------------+
|  Encryption  |       0.3015      |
|  Decryption  |       0.0880      |
| Vault Update |       0.8448      |
|  Total Time  |       1.2342      |
+--------------+-------------------+

Initiating ECC Authentication Comparison...


ECC Authentication Performance Metrics
+-----------+--------------+--------------+-------------+------------+---------+
| Device ID | Key Gen (ms) | Signing (ms) | Verify (ms) | Total (ms) |  Status |
+-----------+--------------+--------------+-------------+------------+---------+
|  Device01 |    8.7370    |    0.7463    |    2.9701   |   3.7227   | Success |
|  Device02 |    0.6649    |    0.6935    |    2.7581   |   3.4577   | Success |
|  Device03 |    0.6810    |    0.7471    |    2.8265   |   3.5782   | Success |
|  Device04 |    0.7254    |    0.7326    |    2.7448   |   3.5057   | Success |
|  Device05 |    0.6902    |    0.6690    |    2.8472   |   3.5194   | Success |
|  Device06 |    0.6534    |    0.7114    |    2.8048   |   3.5196   | Success |
|  Device07 |    0.6648    |    0.6686    |    2.7392   |   3.4113   | Success |
|  Device08 |    0.6390    |    0.7340    |    2.7855   |   3.5224   | Success |
|  Device09 |    0.6829    |    0.6819    |    2.8088   |   3.4939   | Success |
|  Device10 |    0.6746    |    0.7280    |    2.7353   |   3.4667   | Success |
+-----------+--------------+--------------+-------------+------------+---------+

Performance Averages (Successful Authentications Only)
+-----------------------+-------------------+
|         Metric        | Average Time (ms) |
+-----------------------+-------------------+
|     Key Generation    |       1.4813      |
|   Signature Creation  |       0.7112      |
| Signature Verification |       2.8020      |
|  Total Authentication |       3.5198      |
+-----------------------+-------------------+
```

Figure 5: Simulator's output