Università degli Studi di Padova

Dipartimento Matematica

Dipartimento di Matematica "Tullio Levi-Civita"

Università degli Studi di Padova

Master Degree in Cybersecurity

Report fot the Course Cyber-Physical Systems and IoT Security
Authentication, topic: Internet of Things

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

Report and Project:
Nicola Busato 2119291

Academic year 2024/2025

Contents

1 Objectives

The goal of this project is to simulate the authentication process between an IoT Device and an IoT Server using Secure Vaults.
The objectives of the project are divided into the following sub-tasks:

- **Authentication Development**: Implement an authentication mechanism capable of securely verifying the identity of both the IoT Device and the IoT Server using Secure Vaults.
- **Comparison**: Measure the authentication time and compare its performance with other authentication mechanisms.

2 System Setup

The authentication algorithm was developed using Python 3.12.3 and utilises the following libraries:

- **Threading**: Simulates the server and devices running as separate threads.
- **time**: Manages execution timing and delays.
- **random**: Generates random values for keys, challenges, and numbers.
- **prettytable**: Generates the final table for performance results, `pip install prettytable`.
- **Crypto**: Provides AES encryption and decryption functionality and ECC key generation, `pip install pycryptodome`.
- **hmac**: Implements Hash-based Message Authentication Code (HMAC) for vault updates.
- **hashlib**: Provides secure hashing algorithms for cryptographic operations.

2.1 Design Choices

The paper doesn't provide the way to syncronise the security vault between IoT server and IoT devices. The method `setUpConnection(IoTDevice)` return a common Secure Vault between device and server. The paper doesn't provide a way to this, it's specify that the server and device already have the same security vault. For simplicity, the Secure Vault is returned to the device in plaintext. However, in a real-world scenario this exchange must be done in a secure way, e.g. using public/privat key between device and server: device generate RSA key pair (public and private keys), shares its public key with the server, than the server encrypts the Secure Vault with the device's public key and sends it back to the device. The device decrypts the Secure Vault using its private key.
Additionally, my implementation is written in Python rather than Arduino language.

2.2 Authentication's code

The authentication consists of six Python files:

- **global_variables.py**: Contains global variables:
  - `N`: The number of keys stored in the Secure Vault for each IoT device and server.
  - `M`: The size of each key in bytes (used in AES encryption). Possible values: 16 (AES-128), 24 (AES-192), 32 (AES-256).
  - `P`: The number of keys selected in a challenge request (C1 or C2). It must be less than or equal to N to ensure valid key selection.
  - `SHA512_WITH_HMAC`: Determines how the Secure Vault is regenerated. If True, SHA512-HMAC is used; otherwise, SHA-512 is used. More info in section 2.2.1

- **crypto_utils.py**: Defines the methods `encrypt(key, payload)` and `decrypt(key, payload)`, which use AES-CBC mode for witch respectively, ecrtypts the plaintext using a key, and decrypt the ciphertext using the key, this methods are shared between server and devices
- **secure_vault.py**: Contain `SecureVault` class that contains `N` keys with length `M` byte.
- **iot_device.py**: Contains the `IoTDevice` class, which represents the IoT Device to authenticate.
- **iot_server.py**: Contains the `IoTServer` class, which represents the IoT Server for the authentication.
- **main.py**: This script initialises an IoT server and multiple IoT devices, simulating a real-world IoT authentication process using the Secure Vault mechanism. It also get and prints the performance results.

2.2.1 SecureVault

The `SecureVault` class is used to store `N` keys, each of size `M` bytes. These keys are randomly generated during initialization. The class contains the following methods:

- **getKey(challenge: list[int])**: The `challenge` is a list of indices of the SecureVault. The method returns the session key by XORing the elements of the SecureVault based on the challenge.
- **generateChallenge()**: Generates a challenge, i.e., a list of `P` unique indices, where each index has a value between 0 and `N`.
- **generateRandomNumber()**: Generates a random number of `M` bytes.
- **updateVault(data: bytes)**: Updates the Secure Vault using either SHA512-HMAC or SHA512, depending on the global configuration. It returns the time taken to update the vault. The update can be performed in two ways:
  - **SHA512 with HMAC**: Updates the vault by computing an HMAC over the concatenation of all keys. The HMAC result is partitioned and XORed with existing keys to refresh them.
  - **SHA512**: For each key in the vault, compute its SHA-512 hash and use the first `M` bytes to update the key.

2.2.2 IoTDevice

The `IoTDevice` class represents the IoT device that must be authenticated with the server. A device is initialized with a unique device ID, which is assigned sequentially in the main file. The class provides the following methods:

- **getID()**: Returns the device ID.
- **connect(server: IoTServer)**: Establishes a connection with the IoT server. In this method, the device calls `Server.setUpConnection()` (ref: 2.2.3) to send an instance of itself to the server, providing information such as the device ID. The server returns the Secure Vault. In a real-world scenario, the Secure Vault should be exchanged securely, e.g., using encrypted messages. However, in this implementation, the device simply receives a copy of the Secure Vault generated by the server, ensuring both share the same Secure Vault. The device then generates a session ID and sends the message `m1 = (deviceID, sessionID)` to the server to start the authentication process using `Server.authentication(m1)` (ref: 2.2.3).
- **sendMessage2(m2: tuple)**: Processes the message `m2` received from the server and generates `m3`. The message `m2` contains `c1` (challenge) and `r1` (random number). The device uses `c1` to generate the key `k1` using `secureVault.getKey(c1)` (ref: 2.2.1). It then generates a new challenge `c2` and two random numbers `r2` and `t1`. The device returns an encrypted message `m3 = (r1 + t1 + c2 + r2)`, encrypted using the key `k1`.
- **sendMessage4(m4: bytes)**: Processes the message `m4` received from the server. This method verifies the authentication response from the server. The device generates the key `k2` using the challenge `c2`, which was previously generated in the `sendMessage2()` method. It then computes the key `k3` by XORing

k2 and t1. The device decrypts the message m4 and retrieves r2 and t2. If r2 does not match the value previously sent to the server, the server authentication fails. Otherwise, the device generates a session key by XORing t1 and t2. This session key is used to update the Secure Vault.

- **getTimings()**: Returns the timing information for the device. Each phase of encryption, decryption, and Secure Vault update is timed. The method provides the time taken to encrypt m3, decrypt m4, update the Secure Vault, and the total authentication time.

2.2.3 IoTServer

The IoTServer class represents the IoT server responsible for managing authentication requests from IoT devices. The server handles multiple devices concurrently. The class provides the following methods:

- **setUpConnection(device: IoTDevice)**: Registers a new IoT device and initializes the Secure Vault. The server creates a copy of the Secure Vault and associates it with the device. This method returns the Secure Vault to the device, ensuring both the server and the device share the same keys. In a real-world scenario, this exchange should be secured using encryption.
- **authentication(m1: tuple)**: The server start the authentication process. The message m1, containing the device ID and session ID. If the device is already paired, the server rejects the request. Otherwise, the server begins the authentication processing.
  The server performs the following steps for each device:
  First, the server retrieves the device's authentication request, including the device object and its associated Secure Vault. The server then generates a challenge c1 and a random number r1, which are sent to the device as part of the message m2.
  Next, the server receives the encrypted message m3 from the device. It decrypts m3 using the key k1, derived from the Secure Vault using the challenge c1. The server parses the decrypted m3 message to extract r1Received, t1, c2, and r2. It verifies that r1Received matches the original r1 sent to the device. If they do not match, the authentication fails.
  The server then generates the key k2 using the challenge c2 from the device. It computes k3 by XORing k2 with t1 and generates a new random number t2. The server creates the message m4 by encrypting using the key k3 the concatenation of r2 and t2. This message is sent to the device for final verification.
  The server waits for the device to verify m4. If the device successfully verifies the message, the server generates the session key by XORing t1 and t2. The Secure Vault is then updated using the session key, ensuring it is refreshed after each successful authentication. Finally, the server adds the device to the list of paired devices, marking it as successfully authenticated.

2.2.4 Main

The SVauthentication.py script is the entry point for the IoT authentication system. It initializes the IoT server, simulates multiple IoT devices, and measures the performance of the authentication process using the Secure Vault mechanism.
Global Variables:

- IoTDeviceNum: Number of IoT devices to simulate (default: 30).
- DEBUG_LOGS: Flag to enable or disable debug logs. When False, logs are suppressed.

The script starts by initializing the IoT server and running it in a separate thread. Next, it simulates multiple IoT devices (IoTDeviceNum), each assigned a unique ID, connecting to the server through the connect() method. During this process, the script collects timing data for encryption, decryption, Secure Vault updates, and the total authentication time for each device. It also tests duplicate connections by reconnecting certain devices (e.g., device 1 and device 3) to observe how the server handles repeated authentication requests. After gathering the data, the script calculates average times for encryption, decryption, Secure Vault updates, and overall authentication across all devices. Finally, it presents the timing results in a table using the `PrettyTable` library—showing both individual device timings and aggregated averages—and concludes by printing a summary of the results before exiting.

Output The script displays a table with the timing results for each IoT device, with the following metrics:

- **Device ID**: The unique identifier of the IoT device.
- **Encrypt (ms)**: The time taken to encrypt messages during the authentication process.
- **Decrypt (ms)**: The time taken to decrypt messages during the authentication process.
- **SV Update (ms)**: The time taken to update the Secure Vault after successful authentication.
- **Auth. (ms)**: The total time taken for the entire authentication process.

The table also includes a row for the average times across all devices, providing a summary of the overall performance. Additionally, the script prints the times to perform the Secure Vault algorithm and the Single Rotating Password.

2.3 Starting the Simulator

To run the code, execute the following command:

```
# Install required dependencies (if not already installed)
pip install pycryptodome prettytable

# Run the simulation of Secure Vault
python3 SecureVaultAuthentication/main.py

# Run the simulation of ECC
python3 ECCauthentication/main.py
```

Listing 1: Shell command to start the authentication simulation.

2.3.1 Simulation Configuration

The simulator allows customization of several parameters to adjust its behavior. These parameters are defined in the `global_variables.py` file and can be modified as needed:

- **N**: The number of keys stored in the Secure Vault. The maximum value is 256.
- **M**: The size of each key in bytes. Valid values are 16 (AES-128), 24 (AES-192), or 32 (AES-256).
- **P**: The number of keys used in each authentication challenge. This value must be less than or equal to `N`.
- **SHA512_WITH_HMAC**: Determines the method used to update the Secure Vault. Set to `True` to use SHA512-HMAC or `False` to use SHA-512.
- **IoTDeviceNum**: The number of IoT devices to simulate. This parameter is defined in the `main.py` file.

3 Experiments

This section describes the methodology used to obtain the results. In the main file, `IoTDeviceNum` define the number of generated devices, and for each of them, the script measures the time taken for encryption, decryption, Secure Vault updates, and the overall authentication process. The script also calculates and prints the average time for each of these metrics.

The reference paper compares the results of Secure Vault authentication with another method that uses Elliptic Curve Cryptography (ECC). To replicate this comparison, I implemented an ECC-based authentication method in the *ECCauthentication* directory. This method authenticates devices and the server using public and private keys, with signing and verification performed using ECC. To run this implementation, use the command `python3 ECCauthentication/main.py`. I will not explain the ECC authentication in detail, as it is not the main focus of this report. However, its implementation is similar to that of Secure Vault authentication, with the key difference being that ECC relies on key generation and sign/verify methods using Elliptic Curve Cryptography.
Similar to the Secure Vault implementation, ECC script prints the timing for devices, including the following metrics:

- **Device ID**: The unique identifier of the IoT device.
- **Key Gen (ms)**: The time taken to generate the ECC key pair.
- **Sign (ms)**: The time taken to sign the messages.
- **Verify (ms)**: The time taken to verify the signature of the message.
- **Auth. (ms)**: The total time taken for the entire authentication process.

Additionally, it prints the overall ECC execution time, which is the sum of Key Generation, Signing, and Verification times.
I'll use these values to compare the algorithms, as the paper focuses on evaluating the authentication time for each method. By analyzing these metrics, I can assess the efficiency and performance differences between Secure Vault authentication and Single Rotating Password (SRP)/ECC-based approach.

4 Results and Discussion

4.1 Outputs

These are the outputs of both the scripts, allowing me to retrieve the timing of each authentication process, such as SV, SRP and ECC.

4.1.1 Secure Vault

```
Timing Results, AES-128 as Encryption/Decryption, SHA-512 with HMAC for secure vault update:
+----------+--------------+--------------+----------------+--+------------+
| DeviceID | Encrypt (ms) | Decrypt (ms) | SV Update (ms) |  | Auth. (ms) |
+----------+--------------+--------------+----------------+--+------------+
|    01    |  0.03814697  |  0.00858307  |   0.05125999   |  | 0.27489662 |
|    02    |  0.03695488  |  0.00762939  |   0.03576279   |  | 0.26965141 |
|    ..    |     ....     |     ....     |      ....      |  |    ....    |
|    30    |  0.03671646  |  0.00786781  |   0.03576279   |  | 0.26750565 |
+----------+--------------+--------------+----------------+--+------------+
| Average  |  0.03468990  |  0.00821749  |   0.04503727   |  | 0.26586223 |
+----------+--------------+--------------+----------------+--+------------+

AES-256: 21.454 us
SHA-512 with HMAC: 45.037 us

AES-256 * 2 + SV Update = 87.945 us
```

Figure 1: Sample output from *SecureVaultAuthentication/main.py*

4.1.2 ECC

```
Timing Results, ECC
+-----------+--------------+------------+-------------+--+------------+
| Device ID | Key Gen (ms) | Sign (ms)  | Verify (ms) |  | Auth. (ms) |
+-----------+--------------+------------+-------------+--+------------+
|    01     |  0.02098083  | 0.32758713 |  0.75745583 |  | 2.17008591 |
|    02     |  0.02169609  | 0.32329559 |  0.73170662 |  | 2.12645531 |
|    ..     |      ....    |    ....    |      ....   |  |    ....    |
|    30     |  0.02074242  | 0.31876564 |  0.73552132 |  | 2.11095810 |
+-----------+--------------+------------+-------------+--+------------+
|  Average  |  0.02071428  | 0.31966388 |  0.74338746 |  | 2.12877221 |
+-----------+--------------+------------+-------------+--+------------+

ECC: Key gen + sing + verify = 1083.76562595 us
```

Figure 2: Sample output from *ECCauthentication/main.py*

4.1.3 Results in the paper

The reference paper calculates energy consumption using an Arduino board with an average power consumption of 99.5 mW (19.9 mA at 5V). Since my tests were conducted on a PC, direct comparisons of execution times are not equivalent. However, I can compare the ratio between the Secure Vault and the SRP/ECC algorithms. Table 1 shows the execution time and energy consumption values from the reference paper.

Algorithm	Average time to execute	Energy consumed
AES - 128 bits	2.5 ms	248.75 μJ
AES - 256 bits	4 ms	398 μJ
SHA 512	1 ms	99.5 μJ
SHA 512 with HMAC	1.5 ms	149.25 μJ
ECC	1105 ms	109.95 mJ

Table 1: Execution time and energy consumption table from the paper

Using these values, the Secure Vault time is calculated as:

$$\text{SV: AES - 128 bits } \times 3 + \text{ SHA-512 with HMAC} = 6.5 \text{ ms}, \text{ resulting in } 646.75\,\mu J$$

For the Single Rotating Password (SRP):

$$\text{SRP: AES - 128 bits} \times 3 = 7.5\,\text{ms}, \text{ resulting in } 746.25\,\mu\text{J}$$

For ECC, the energy consumed is:

$$\text{ECC: time} \times \text{avg consumption} = 109950\,\mu\text{J}$$

Note : I think that the paper contains an error, as in Figure 3, "Energy Consumption Comparison" the ECC energy consumption should be 109,950 μJ (=109.95 mJ) and not 10,995 μJ (=10.995 mJ) as stated in the paper.

In the paper, the Single Rotating Password consumes 1.15 (746.25/646.75) times the energy of the Secure Vault algorithm, while ECC consumes 170 (109950/646.75) times the energy of the Secure Vault algorithm.

4.2 My experimental results

Since the paper does not specify the values of N, M, and P, I used: $\mathbf{N = 16, M = 16}$ **and** $\mathbf{P = 6}$
Different values may increase or decrease the performance gap between the methods.
Running both scrips for 10,000 devices, I obtained the results in Table 2:
These results are 100 to 200 times smaller than the paper's results. This discrepancy is due to the fact that I ran the code on a PC rather than on an Arduino or IoT device, which have significantly lower computational power.

Algorithm	Average time to execute
AES - 128 bits	17.321 μS
AES - 256 bits	20.703 μS
SHA 512	14.675 μS
SHA 512 with HMAC	31.720 μS
ECC	1.094 mS

Table 2: Execution time and energy consumption table from my experiments

Notes : These values may vary based on PC hardware and usage. The AES - *** values are the average between encryption and decryption.

4.2.1 Secure Vault

Secure Vault authentication with AES - 128 and SHA 512 with HMAC :

```
...
AES-128: 17.321 us
SHA-512 with HMAC: 31.720 us

SV algorithm: AES-128 * 2 + SV Update = 66.362 us
Single Rotating Password: AES-128 * 3 = 51.963 us
```

Figure 3: Output from *SecureVaultAuthentication/main.py* with 10000 devices, all the times in file *other/SV.out*

The SV method requires 66.362 μS to perform the most computationally intensive operations, even for IoT devices, such as AES encryption/decryption and the Secure Vault update. One key advantage of the SV authentication method is that key generation is simple, as there is no need for private and public key pairs that depend on each other. The SV method can be compared to a symmetric key approach since both the device and the server share the same key, the secure vault, and only need to send a challenge inside an encrypted message to verify that the device and server share the same Secure Vault in order to authenticate.

4.2.2 Single Rotating Password

In my experiments, the Single Rotating Password (SRP) outperformed the Secure Vault (SV) method, with SRP taking 51.963 μS compared to SV's 66.362 μS, so SRP required 0.75 times the execution time of SV. This contrasts with the paper's results, where SRP was 1.15 times slower than SV. The discrepancy likely arises because, in my Python implementation, SHA-512 with HMAC is slower than AES, whereas in the paper's Arduino-based implementation, SHA-512 with HMAC may perform better compared to AES. This may due to hardware optimizations and differences in cryptographic library efficiency on embedded systems. To verify this hypothesis, the algorithms could be reimplemented in Arduino language. This would help determine whether the performance difference is due to the programming language or other factors.

4.2.3 ECC

```
...
ECC: Key gen + sing + verify = 1094.69288074 us
```

Figure 4: Output from *ECCauthentication/main.py* with 10000 devices, all the times in file *other/ECC.out*

The ECC authentication process is slower compared to the Secure Vault method, as ECC involves more computationally intensive operations such as key generation, signing, and verification. This aligns with the paper, where ECC-based authentication consumes more energy and takes longer to execute.

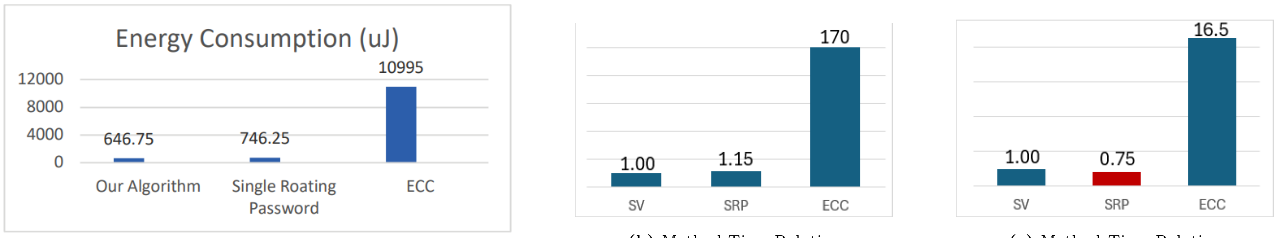
4.2.4 Results Analysis

We have already observed that the Single Rotating Password (SRP) performs better than the Secure Vault (SV) in my experiments, which contrasts with the paper's findings. Regarding ECC, I found that SV significantly outperforms ECC, with SV having an execution time of 66.362 μS compared to ECC's 1094.69 μS. This means ECC is approximately 16.49 times slower than SV in my tests, whereas in the paper, ECC is 170 times slower than SV. This discrepancy could be attributed to several factors, including:

- **Hardware Differences**: The paper's results are based on Arduino/IoT devices, which have significantly lower computational power compared to a PC.
- **Implementation Details**: Differences in how the algorithms are implemented (e.g., optimizations, libraries, or code structure) could also contribute to the performance gap.
- **Parameter Differences**: The paper does not specify the values of N, M, and P, which could significantly affect the performance of the algorithms. Different values of these parameters may lead to variations in execution speed and resource usage.

4.2.5 Plots comparison

I compared the plot generated by my simulator with the one from the paper.



(a) Paper's "Figure 3 Energy Consumption Comparison"

(b) Method Time Relative to SV Time (Paper's Results)

(c) Method Time Relative to SV Time (My Results)

Figure 5: Comparison between the plots

In Figure 5, we can see the paper's plot (a), which shows the execution time of each method relative to the SV time. It illustrates that SV is the fastest, SRP is slightly slower, and ECC takes 170 times longer than SV. These execution times also reflect the corresponding energy consumption.
For plot (c), which represents my results, we observe that SRP has a lower execution time compared to SV, and ECC is only 16.5 times slower than SV, rather than 170 times as reported in the paper.

4.3 Conclusion

In summary, my experimental results indicate that the Secure Vault method offers better performance than ECC-based authentication, but it is slightly slower than Single Rotating Password under the specific conditions of my Python-based implementation on a PC. These differ from the reference paper's results, where SRP was slower than SV and ECC was found to be 170 times slower than SV (compared to 16.5 times in my tests). Several factors likely account for these discrepancies, including differences in hardware (Arduino and PC), variations in cryptographic library efficiencies, and parameter choices.
Overall, the results suggest that the Secure Vault approach is an efficient solution for authentication in IoT environments.

4.4 Future Work

- Re-implement the algorithms on Arduino/IoT hardware to eliminate hardware bias.
- Benchmark cryptographic libraries across platforms to identify efficiency variations.
- Verify the performance with different parameters (N, M, and P).