Enhancing the security of the MQTT protocol in the Internet of Things using the Syracuse conjecture

OUATTARA YACOUBA[1], COMPAORE WENDPUIRE OUSMANE[2], OUEDRAOGO PATINDE VICTOR MARIE JACQUES[3], TRAORE YAYA[4]

[1](*Science and Technology Department, Institut Burkinabè des Arts et des Métiers (IBAM)/ Université Joseph Ki-Zerbo, Burkina Faso)*

[2](*Institut Universitaire de Technologie (IUT)/ Université Nazi BONI, Burkina Faso)*

[3](*Science and Technology Department, Institut Burkinabè des Arts et des Métiers (IBAM)/ Université Joseph Ki-Zerbo, Burkina Faso)*

[4](*Science and Technology Department, Institut Burkinabè des Arts et des Métiers (IBAM)/ Université Joseph Ki-Zerbo, Burkina Faso)*

ABSTRACT: *This article proposes a scientific contribution to strengthen the security of the MQTT protocol in an IoT environment. MQTT is natively a communication protocol that does not embed any security. Messages are transmitted in clear text over the network. Being an IoT protocol, MQTT evolves in an environment with limited resources, where energy remains an important factor in the implementation of security solutions. It is important to increase security without affecting the autonomy of IoT. This is what we propose by using the "Syracuse conjuncture" (or the Collatz sequence) as a pseudo-random key generation mechanism to strengthen security and authentication in MQTT.*

KEYWORDS - *Collatz conjecture, Collatz sequence, dynamic authentication, Internet of Things (IoT), lightweight encryption, MQTT, Security*

I. INTRODUCTION

This paper proposes a new method to enhance the security of MQTT protocols in the Internet of Things. Our solution, based on the Syracuse conjuncture, includes three mechanisms that we detail later. It extends existing research on securing MQTT [1] [2] [3] [4] [5] and is based on fundamental mathematical work relating to the Syracuse/Collatz sequence [6] [7] [8] [9], innovating by using this sequence as a pseudo-random generator.

To facilitate understanding, the article is organized as follows: a review of the work on the security of the MQTT protocol and the Syracuse situation, the research methodology, then the presentation of our “MQTT-SYRACUSE” solution and the results obtained.

II. RELATED WORKS

MQTT is a lightweight messaging protocol based on the publish/subscribe model, designed to facilitate data exchange between connected devices in environments with bandwidth, power, and hardware constraints [10] [11] [12].

Communication is message-oriented, with each message associated with a topic to categorize the data exchanged. This structure promotes decoupled communication between entities, reducing dependencies between senders and receivers [3] [10].

The MQTT protocol integrates Quality of Service (QoS) to ensure message delivery between a client and a broker. It allows communication reliability to be adapted according to the application's needs and network or hardware constraints. According to the MQTT specification, three QoS levels are defined [10] [13].

Current research explores several approaches to enhance MQTT security, such as integrating TLS/DTLS, certificate authentication, or using intrusion detection systems (IDS) [1] [2] [4] [5]. However, object constraints (CPU, memory, energy) sometimes make the use of full TLS/DTLS difficult or expensive, which limits their systematic deployment on constrained devices [4] [12].

Common vulnerabilities identified in the literature include:

- The absence of default encryption of payloads, exposing clear text data to passive observers [11] [14];
- Static/predictable identification of topics (enumeration), facilitating spying and scraping of sensitive topics [4] [15];
- Weak or missing authentication, which allows for spoofing and unauthorized access if strong authentication mechanisms are not in place [16] [18]:
- Centralized points of failure at the broker level, making the ecosystem vulnerable if the broker is compromised or unavailable [14] [17];
- Vulnerability to MITM, spoofing, and replay attacks, especially in the absence of integrity and non-repudiation mechanisms adapted to IoT constraints [2] [4] [11].

Recent research has also focused on improving the security of the MQTT protocol taking into account the constraints of IoT devices [1] [2] [3] [4] [5] [10] [11] [12] [14]. Several approaches can be distinguished:

- Encrypted communications security: the integration of TLS/DTLS has been widely studied. However, the computational complexity and energy consumption of these protocols limit their adoption in constrained environments [3] [4] [13] [12].
- Lightweight authentication and access management: several studies propose the use of mechanisms more suited to microcontrollers, such as Pre-Shared Keys (PSK), lightweight HMAC signatures, or even authentication schemes based on the derivation of lightweight keys [1] [5] [16] [18].
- Symmetric encryption of payloads: some authors suggest the use of low computational cost symmetric algorithms, or even pseudo-randomly generated binary masks to protect messages in transit [4] [15] [19] [20].
- Obscuration and protection of topics: in order to limit the enumeration and spying on sensitive topics, dynamic renaming and obscuration mechanisms have been explored [15] [17].
- Intrusion detection system (IDS) and defense in depth: several contributions have studied the integration of IDS systems based on machine learning or behavioral detection to identify anomalies and attacks targeting MQTT [2] [5] [11] [21] [22].

Despite these efforts, few studies have focused on the use of deterministic recursive sequences as sources of pseudo-randomness to enhance security. Exploiting the Syracuse sequence, with its seed sensitivity and computational simplicity, is a still largely unexplored avenue [23], although its mathematical structure has been widely studied in the literature [6] [7] [8] [9] [24].

III. METHODOLOGY

1. Proposed approach

This paper proposes a general objective which is to integrate the Syracuse sequence as a deterministic pseudo-random generation mechanism, in order to strengthen the security of MQTT without compromising the performance constraints specific to IoT environments [20] [23]. The sequence is defined recursively [20] [23], according to the mathematical foundations established by Allouche, Lagarias and Andaloro [6] [8] [9], with modern proofs and demonstrations proposed recently [7] [24].

$F(n)= \{n/2$ if n is even and $3n+1$ if n is odd$\}$

Where n is the initial seed chosen in a shared manner between the communicating entities.

This approach is based on three specific objectives:

- Generate unique and dynamic IDs for topics to reduce collisions and enumeration by attackers.

- Apply lightweight encryption of payloads via Syracuse-derived bitmasks.
- Implement challenge-response authentication based on the Syracuse suite.

1.1. Generation of unique IDs for topics

In a dense IoT network using MQTT, topic management is a critical issue because each sensor publishes its data on a specific topic (e.g. sensor/temp). In such a network two main problems appear [20] [23]:

- Topic collisions: Two sensors may accidentally use the same topic name, leading to confusion in message processing.
- Exposure of sensor information: An attacker could guess the source of messages by systematically enumerating topics, which reduces the anonymity of IoT devices.

The proposed solution relies on the flight time of an integer n_initial in the Syracuse sequence to derive a unique identifier for the topic. The principle is as follows:

- Each sensor has a secret seed n_initial (for example derived from its hardware ID or a timestamp hash).
- We calculate the number of steps necessary for the Syracuse sequence to reach 1 (called flight time).
- This number is used to create a unique topic, for example: /sensors/syracuse/<flight_time>

Examples: n_initial = 10 → flight time = 27 → topic = /sensors/syracuse/27

n_initial = 42 → flight time = 8 → topic = /sensors/syracuse/8

The advantages of this method are:

- Guaranteed uniqueness: different seeds generally produce different flight times, which reduces collisions.
- Dynamic and unpredictable: an attacker cannot easily predict the topic without knowing the initial seed.
- Anonymization: the topic does not contain directly identifiable information about the sensor.
- Lightweight and IoT compatible: time-of-flight calculation is very fast and consumes few resources.

The limitations and precautions to be taken following this method are as follows:

- If multiple sensors use the same seed, collision may occur. Therefore, it is recommended to add a salt or combine the seed with a unique sensor ID (n_initial = hash(sensor_ID + timestamp)).
- This method is not cryptographically secure on its own, it must be used as a complement to security mechanisms (TLS, XOR encryption, etc.).

1.2. Lightweight encryption of payloads

The MQTT protocol, by default, does not encrypt data exchanged between sensors and brokers unless TLS/SSL is used. However, TLS can be computationally expensive and energy-intensive for IoT microcontrollers. To lightly secure non-critical data while remaining lightweight and IoT-friendly, we propose an XOR cipher based on the Syracuse suite.

The principle of the lightweight payload encryption method is as follows:

- Shared seed: the transmitter (sensor) and the receiver (broker) have a secret integer n_initial [19] [20].
- Binary mask: generated from the least significant bits (LSB) of each term of the Syracuse sequence.
- Encryption: The payload is XOR with the generated mask to produce an encrypted message.

This technique relies on the sensitivity of time-of-flight and Syracuse terms, even close seeds produce different binary sequences, making the mask difficult to predict without knowing the seed. As a concrete example we have:

- Payload: [1,0,1,0]
- Mask (for n=6): [0,1,0,1]
- Encrypted payload: [1^0, 0^1, 1^0, 0^1] = [1,1,1,1]

This method has certain advantages which are listed below:

- Lightweight: Simple calculation, suitable for low-power microcontrollers.
- Low overhead: No heavy crypto library or full TLS protocol.
- Dynamic: Each transmission can use a different seed to vary the mask.

- IoT Compatible: Works with standard MQTT and limited microcontrollers.

The lightweight payload encryption method has its limitations which are:

- Limited security: if the attacker knows the seed, he can decrypt the messages.
- Non-standard: does not replace TLS/AES for sensitive data.
- Limited payload length: The mask must be regenerated for longer or continuous payloads.

1.3. Lightweight authentication

In MQTT, traditional authentication relies on static login/password, which presents several vulnerabilities. Passwords can be intercepted if TLS is not used. Resource-constrained IoT devices cannot always support heavyweight authentication protocols. To strengthen authentication while remaining lightweight and suitable for microcontrollers, we propose a challenge-response mechanism based on the Syracuse sequence. Following this method, the broker sends an integer n as a challenge to the sensor. The sensor calculates the next term of the Syracuse sequence from n and sends it back to the broker. Finally, the broker checks the response if the term matches the expected sequence; the connection is allowed, otherwise it is denied [19] [23]. This approach takes advantage of the deterministic but short-term unpredictable nature of Syracuse, without knowing the exact challenge or follow-up, an attacker cannot predict the response. A concrete example of the use of lightweight authentication is given below:

- The Challenge: n = 7
- The sensor calculates 3*7 + 1 = 22
- The broker checks: if 22 corresponds to the expected term → connection accepte.

This lightweight authentication method has several advantages, such as being lightweight with simple computation suitable for microcontrollers, improved security preventing static authentication and reducing the risk of simple replay. Quite dynamic, each challenge is unique to each connection and compatible with IoT, the minimal implementation and can be integrated directly into MQTT. But some limitations remain to be deplored, such as those cited:

- If an attacker knows the logic and captures multiple challenge-responses, he can possibly predict what happens next.
- It does not replace TLS or strong cryptographic mechanisms for sensitive data.
- Security depends on the random generation and confidentiality of the initial seeds.

2. Experimental parameters

In this study, a hardware and software environment is required for its implementation. The hardware environment used is a computer with a Windows or Linux operating system acting as a broker. For the software environment, an MQTT client with a Paho-MQTT library (Python), an MQTT broker Mosquitto (open-source, lightweight) and the Syracuse scripts implemented in Python [20] [23]. The Python language was chosen for its flexibility, efficiency and compatibility with existing IoT libraries [23].

3. Experimental validation

The methodology will be validated by software simulation in a classic MQTT environment (Eclipse Mosquitto) [20]. Performance measurements (latency, CPU overload, memory consumption) are performed. Security tests against targeted attacks (topic enumeration, payload sniffing, identity theft) are applied to the simulation [19] [23]. The results will allow comparing the Syracuse approach with existing methods (TLS, PSK, IDS) in terms of trade-off between security and computational lightness (energy, resources) [20].

IV. ALGORITHM OF OUR « MQTT-SYRACUSE »

1. Basic algorithm

Our algorithm for lightweight MQTT security using the Syracuse conjecture follows the following procedure:

- The sensor generates environmental data (temperature).
- The data is encrypted with the Syracuse mask before sending to the broker.
- The message is posted on a topic based on flight time.

• The broker regenerates the Syracuse mask, decrypts the payload and stores the clear data.

• Syracuse authentication is used during connection to verify the identity of the sensor.

2. Comparison of Syracuse conjecture with a standard solution implemented in MQTT

The standard MQTT protocol does not integrate any security solution, the data circulates in clear text on the network. It is necessary to apply a minimum of security such as encryption, authentication and use of secure ID. In the security standards implemented in MQTT, TLS/SSL is used for encryption, Login/Password for authentication and random UUID (Universally Unique Identifier) for generating dynamic IDs for topics. The Syracuse conjecture integrates MQTT for security, XOR + Collatz Mask is used for encryption, Collatz Challenge-Response for authentication and Syracuse Time of Flight for generating dynamic IDs for topics.

Needs	**Standard Solution**	**Syracuse solution**
Encryption	TLS/SSL	XOR + Collatz Mask [19]
Authentication	Login/Password	Collatz Challenge-Response [23]
ID Generation	Random UUIDs	Flight time from Syracuse [20] [23]

3. Example code (Python + Paho-MQTT)

The Syracuse conjecture is integrated into MQTT for lightweight security. To validate the feasibility of the proposed approach, an experimental implementation was performed using the MQTT protocol and an integration of the Syracuse suite for lightweight encryption and authentication.

The implementation requires a hardware and software environment. The hardware environment uses a computer with a Windows or Linux operating system acting as a broker.

For the software environment:

• MQTT client: Paho-MQTT library (Python).

• MQTT Broker: Mosquitto (open-source, lightweight).

• Syracuse scripts: implemented in Python for mask generation and authentication mechanism.

The implementation covers three (3) main components.

• Lightweight encryption of payloads: generation of a binary mask from the Syracuse sequence and application of an XOR between the mask and the sensor payload.

• Generate unique IDs for topics: Use Syracuse's time of flight as a dynamic topic suffix.

• Example: sensors/temp/27 for an initial_n = 10 (time of flight = 27).

• Lightweight authentication: Challenge-response where the broker sends n, and the sensor responds with the next term in the sequence.

Example code (Python):

```
import paho.mqtt.client as mqtt
# Generation of the Syracuse mask
def syracuse_mask(n, length):
    mask = []
    for _ in range(length):
        mask.append(n % 2)  # LSB
        n = n // 2 if n % 2 == 0 else 3 * n + 1
    return mask
# XOR encryption function
def encrypt(payload, mask):
    return bytes([p ^ m for p, m in zip(payload, mask)])
# Initial settings
broker = "127.0.0.1"
n_shared = 42              # shared key
payload = b"25.6"          # sensor data
# Mask generation and encryption
mask = syracuse_mask(n_shared, len(payload))
encrypted_payload = encrypt(payload, mask)
#  MQTT Connection and Publication
client = mqtt.Client()
client.connect(broker, 1883, 60)
topic = f"sensors/syracuse/{len(mask)}"
client.publish(topic, encrypted_payload)
print(f" Message published on {topic} : {encrypted_payload}")
```

V. Result and Discussion

1. Result

The python code example has been integrated connection time to the broker, the percentage of CPU and memory used by Mosquitto in order to obtain comparable data based on native MQTT implementations, MQTT with Username/Password and TLS, and finally MQTT with the Syracuse conjuncture.

1.1. Test 1 native MQTT

Test 1 concerns the connection time to the broker, the percentage of CPU and memory used by native Mosquitto. A python script simulates an MQTT client that establishes a connection with the broker and returns the connection data.

```
import time
import psutil
import paho.mqtt.client as mqtt
# ========================
# MQTT SETTINGS
# ========================
BROKER = "127.0.0.1"
PORT = 1883
TOPIC = "temperature"
PAYLOAD = "25.6°C"
# ========================
# CALLBACKS
# ========================
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connection successful ✅ ")
    else:
        print("Connection failure, code:", rc)
def on_disconnect(client, userdata, rc):
    print("Logged out with code:", rc)
# ========================
# START OF CPU/MEMORY MONITORING
# ========================
# Search for the Mosquitto process
mosquitto_proc = None
for proc in psutil.process_iter(['name']):
    if proc.info['name'] and "mosquitto" in proc.info['name'].lower():
        mosquitto_proc = proc
        break
if mosquitto_proc is None:
    raise Exception("Process mosquitto.exe not found!")
# ========================
# MEASURING CONNECTION TIME + CPU/MEMORY
# ========================
client = mqtt.Client()
client.on_connect = on_connect
client.on_disconnect = on_disconnect
# Measurements before connection
cpu_before = mosquitto_proc.cpu_percent(interval=None)
mem_before = mosquitto_proc.memory_percent()
start_time = time.time()
# Login to the broker
client.connect(BROKER, PORT, keepalive=60)
end_time = time.time()
latency_ms = (end_time - start_time) * 1000
# Measurements after connection
cpu_after = mosquitto_proc.cpu_percent(interval=0.5) # average over 0.5s
mem_after = mosquitto_proc.memory_percent()
print(f" Broker connection time: {latency_ms:.2f} ms")
print(f"CPU used by Mosquitto : {cpu_after:.2f} %")
print(f" Memory used by Mosquitto: {mem_after:.2f} %")
# ========================
# PUBLISHING THE MESSAGE
# ========================
client.loop_start()  # needed for callbacks
result = client.publish(TOPIC, PAYLOAD)
if result.rc == mqtt.MQTT_ERR_SUCCESS:
    print(f" Message published {TOPIC} : {PAYLOAD}")
else:
    print("Failed to publish ")
time.sleep(1)  # allow time to send
client.loop_stop()
client.disconnect()
```

The test is carried out five (5) times, in order to have the average time and percentage. The results are as follows:

1.2. Test 2: MQTT with Username/Password, TLS and Self-Signed Certificate

Test 2 concerns the connection time to the broker, the percentage of CPU and memory used by Mosquitto using password, TLS and a self-signed certificate. A python script simulates an MQTT client that establishes a connection with the secure broker and returns the connection data.

```
import time
import psutil
import paho.mqtt.client as mqtt
# ========================
# MQTT SETTINGS
# ========================
BROKER = "127.0.0.1"
PORT = 8883                # Port TLS
USERNAME = "victor"
PASSWORD = "root"
CA_CERT = r"C:\Program Files\mosquitto\certs\mqtt_ca.crt"  # Path to CA certificate
payload = r"25.6°C"
topic = "temperature"
# ========================
# CALLBACKS MQTT
# ========================
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connection successful ✅ ")
    else:
        print("Connection failure, code:", rc)
def on_disconnect(client, userdata, rc):
    print("Logged out with code:", rc)
# ========================
# Search for the Mosquitto process
# ========================
mosquitto_proc = None
for proc in psutil.process_iter(['name']):
    if proc.info['name'] and "mosquitto" in proc.info['name'].lower():
        mosquitto_proc = proc
        break
if mosquitto_proc is None:
    raise Exception("Process mosquitto.exe not found!")
```

	Trial 1	Trial2	Trial 3	Trial 4	Trial 5
Connection time (ms)	3.43	6.70	7.66	7.78	8.96
CPU used (%)	0.00	0.00	0.00	0.00	0.00
Memory used	0.14	0.14	0.14	0.14	0.14

```
# ========================
# MEASURING CONNECTION TIME + CPU/MEMORY
# ========================
client = mqtt.Client(protocol=mqtt.MQTTv311)
client.username_pw_set(USERNAME, PASSWORD)
client.tls_set(ca_certs=CA_CERT)
client.on_connect = on_connect
client.on_disconnect = on_disconnect
# Start of measurement
start_time = time.time()
cpu_before = mosquitto_proc.cpu_percent(interval=None)
mem_before = mosquitto_proc.memory_percent()
client.connect(BROKER, PORT, keepalive=60)
end_time = time.time()
cpu_after = mosquitto_proc.cpu_percent(interval=0.5)
mem_after = mosquitto_proc.memory_percent()
latency_ms = (end_time - start_time) * 1000
print(f"\n Secure connection time to the broker: {latency_ms:.2f} ms")
print(f" CPU used by Mosquitto: {cpu_after:.2f} %")
print(f" Memory used by Mosquitto: {mem_after:.2f} %")
# ========================
# PUBLISHING THE MESSAGE
# ========================
client.loop_start()
client.publish(topic, payload)
print(f"Message publié sur {topic} : {payload}")
client.loop_stop()
client.disconnect()
```

The test is carried out five (5) times, in

order to have the average time and percentage. The results are as follows:

	Trial1	Trial2	Trial3	Trial4	Trial5
Connection time (ms)	19.56	21.07	34.40	20.18	18.63
CPU used (%)	0.00	0.00	0.00	0.00	0.00
Memory used	0.16	0.16	0.16	0.16	0.16

1.3. Test 3: MQTT with the Syracuse conjuncture

Test 3 is for broker connection time, CPU and memory usage percentage by Mosquitto using Syracuse, TLS and self-signed certificate. A python script simulates an MQTT client that establishes a connection with the secure broker and returns the connection data.

```
import time
import psutil
import paho.mqtt.client as mqtt
# ========================
# FUNCTIONS
# ========================
def syracuse_mask(n, length):
    """ Generates a binary mask from Syracuse """
    mask = []
    for _ in range(length):
        mask.append(n % 2)  # LSB
        n = n // 2 if n % 2 == 0 else 3 * n + 1
    return mask

def encrypt(payload, mask):
    """ XOR encryption of the payload with the mask """
    return bytes([p ^ m for p, m in zip(payload, mask)])
# ========================
# MQTT SETTINGS
# ========================
BROKER = "127.0.0.1"
PORT = 1883
TOPIC = "temperature"
N_SHARED = 42
PAYLOAD = "25.6°C"
# ========================
# GENERATION OF THE ENCRYPTED MESSAGE
# ========================
payload_bytes = PAYLOAD.encode("utf-8")
mask = syracuse_mask(N_SHARED, len(payload_bytes))
encrypted_payload = encrypt(payload_bytes, mask)
decrypted_payload = encrypt(encrypted_payload, mask).decode("utf-8")
# ========================
# CALLBACKS MQTT
# ========================
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connection successful ✅", end=" ")
    else:
        print("Connection failure, code:", rc)
# ========================
# MEASURE TIME + RESOURCES
# ========================
mosquitto_proc = None
for proc in psutil.process_iter(attrs=["pid", "name"]):
    if "mosquitto" in proc.info["name"].lower():
        mosquitto_proc = proc
        break
if mosquitto_proc is None:
    print("⚠️ Unable to find the Mosquitto process. Check that it is running.")
    exit(1)
cpu_before = mosquitto_proc.cpu_percent(interval=None)
mem_before = mosquitto_proc.memory_percent()
start_time = time.time()
client = mqtt.Client()
client.on_connect = on_connect
client.connect(BROKER, PORT, keepalive=60)
end_time = time.time()
latency_ms = (end_time - start_time) * 1000
```

```
        cpu_after = mosquitto_proc.cpu_percent(interval=None)
        mem_after = mosquitto_proc.memory_percent()
        print(f"\n Broker connection time: {latency_ms:.2f} ms")
        print(f" CPU used by Mosquitto: {cpu_after:.2f} %")
        print(f" Memory used by Mosquitto: {mem_after:.2f} %")
        # ========================
        # PUBLISHING THE MESSAGE
        # ========================
        client.loop_start()
        result = client.publish(TOPIC, encrypted_payload)
        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            print(f"✅ Clear message: {PAYLOAD}")
            print(f"🔒 Published encrypted message sur {TOPIC} : {encrypted_payload}")
            print(f"🔓 Decrypted message (control): {decrypted_payload}")
        else:
            print("❌ Failed to publish ")
        time.sleep(1)
        client.loop_stop()
        client.dionnect()
```

The test is carried out five (5) times, in order to have the average time and percentage. The results are as follows:

	Trial1	Trial2	Trial3	Trial4	Trial5
Connection time (ms)	14.79	11.95	14.07	12.56	13.89
CPU used (%)	0.00	0.00	0.00	0.00	0.00
Memory used	0.14	0.14	0.14	0.14	0.14

1.4. Comparative balance sheet

The analysis of the three (3) tests reveals that the resource consumption data of native MQTT and MQTT with the Syracuse situation are much closer than that of MQTT using passwords and TLS. The percentage of CPU used by the three (3) tests is zero, that is to say 0.00%.

	Native MQTT	Secure MQTT	MQTT with Syracuse
Average connection time (ms)	10.675	26.515	13.37
Average CPU used (%)	0.00	0.00	0 .00
Average memory used	0.14	0.16	0.14

2. Discussion

After following the experimental implementation procedure, the following experimental results were obtained:

- Encryption/decryption is immediate,
- Memory consumption is negligible (a few bytes for the mask),
- Challenge-response authentication works correctly without network overhead,
- The system remains compatible with a standard MQTT broker, without protocol modifications.

The experimental implementation carried out with the integration of the Syracuse conjecture in MQTT made it possible to highlight the contributions, but also the limits, of this approach.

Computational performance:

- The operations related to the Syracuse sequence (division by 2 or 3n+1 calculation) are very light, even for microcontrollers,
- XOR encryption/decryption is near-instantaneous (< 1 ms for a 10-byte payload),
- Memory consumption remains negligible, because the mask is generated in real time without massive storage.

This makes the approach suitable for resource-constrained environments.

Energy consumption:

- Unlike TLS/SSL, which requires heavy cryptographic calculations (RSA, AES, key negotiation), the Syracuse approach consumes little energy,
- Les mesures montrent une réduction significative du temps processeur utilisé,

donc une meilleure autonomie énergétique des capteurs.

Robustness and security:

- Lightweight encryption: prevents trivial interception of messages, but remains vulnerable to attack if the seed n_initial is discovered,
- Challenge-response authentication: improves protection against unauthorized connections, but is not resilient to prolonged observation attacks (replay or sequence learning).
- Dynamic topics: using Syracuse flight time as an identifier prevents simple topic enumeration by an attacker, but does not provide a strong guarantee of anonymization.

The approach is effective for obfuscation and basic security, but does not replace TLS or AES for critical data (e-health, financial transactions).

- Compatibility and integration: The approach is fully compatible with standard MQTT, as it relies solely on the transformation of payloads and topics. No changes are required on the Mosquitto broker side, the Syracuse logic is integrated only in the client and receiver. This allows for gradual deployment without disrupting existing infrastructures.
- Comparison with standard mechanisms:

Criteria	**TLS/SSL (standard)**	**Syracuse (proposed)**
Security	Strong (AES, RSA)	Low to medium (simple XOR)
Energy consumption	High	Very low
Complexity	High (use of libraries)	Very low (simple operations)
IoT Compatibility	Average	Excellent (Limited MCUs/ Microcontroller Units)
Recommended use	Critical data	Non-sensitive data / prototyping

This assessment highlights that integrating Syracuse into MQTT is a pragmatic compromise. It is not a robust encryption solution, but an additional layer of obfuscation and lightweight authentication. It is ideal for scenarios with low-criticality data (weather, environmental sensors). However, it can be used in addition to standard protocols (TLS, AES) to enhance the diversity of security mechanisms and complicate the work of an attacker.

VI. CONCLUSION ET PERSPECTIVES

In this paper, we explored a novel approach to enhance the security of the MQTT protocol in constrained IoT environments, by exploiting the deterministic but unpredictable properties of the Syracuse suite. The proposed methodology covers three main axes:

- The generation of dynamic identifiers for topics, reducing the risks of enumeration and collisions, while remaining compatible with the publication/subscription logic of MQTT,
- Lightweight encryption of payloads using binary masks derived from Syracuse, offering minimal confidentiality adapted to the CPU, memory and energy constraints of connected objects.
- Syracuse-based challenge-response authentication, introducing a dynamic and renewable mechanism, avoiding the systematic use of full TLS or heavy certificates.

This approach has several advantages: simplicity of implementation on microcontrollers, low computational overhead and significant improvement in the security of MQTT flows against passive and active attacks (topic enumeration, sniffing, spoofing, replay).

The perspectives offered by this study are numerous:

- Integration with other lightweight cryptographic primitives to enhance resistance to cryptanalytic attacks,
- Evaluation on larger and more heterogeneous IoT networks, with multi-broker and multi-device scenarios,
- Formal security analysis and quantification of the effective entropy generated by Syracuse iterations.

In conclusion, the use of deterministic recursive suites such as Syracuse constitutes a promising avenue for reconciling lightness and security in MQTT-based IoT systems, offering an interesting compromise between performance and protection of sensitive data.

REFERENCES

Journal Papers:

[1] SAHMI I. Amélioration de la sécurité des Objets Connectés (IoT) utilisant le protocole MQTT dans le domaine de la E-santé. Toubkal – IMIST (2022).

[2] THIANDOUM A. (2024) L'intelligence artificielle pour la détection d'intrusions dans l'Internet des Objets : le cas du Protocole MQTT. Université Assane Seck de Ziguinchor (2024).

[3] LAAROUSSI Z., NOVO O. (2021) Performance Analysis of the Security Communication in CoAP and MQTT. IEEE CCNC 2021.

[4] AL-OTAIBI N. S., SAYED AHMED H. I., KAMEL S. O. M., EL-KABBANY G. F. Secure Enhancement for MQTT Protocol Using Distributed Machine Learning Framework. Sensors, 24(5):1638 (2024).

[5] HUSNAIN M., HAYAT K., CAMBIASO E., FAYYAZ U. U., MONGELLI M., AKRAM H., ABBAS S. G., SHAH G. A. *Preventing MQTT Vulnerabilities Using IoT-Enabled Intrusion Detection System*. Sensors, 22(2):567. (2022)

[6] ALLOUCHE J.-P. Sur la conjecture de "Syracuse-Kakutani-Collatz". Séminaire de Théorie des Nombres de Bordeaux, vol. 8 (1978-1979), pp. 1–16.

[7] IDOWU M. A., A novel theoretical framework formulated for information discovery from number system and Collatz conjecture data. Procedia Computer Science, vol. 61, pp. 105–111, 2015.

[8] ANDALORO P. J., The 3x + 1 problem and directed graphs. Fibonacci Quarterly, pp. 43–54, 2000.

[9] LAGARIAS J. C., The 3x + 1 problem and its generalizations. The American Mathematical Monthly, vol. 92, no. 1, pp. 3–23, 1985.

[10] KHELILI F. K., BENCHOULA R., HANACHI N. E. Étude comparative de protocoles de communication dans l'IOT. UNIVERSITY OF KASDI MERBAH OUARGLA. 2020

[11] Dinculeană D., Cheng X. Vulnerabilities and limitations of MQTT protocol used between IoT devices. Applied Sciences, 9(5), 848. 2019

[12] Nebbione G., Calzarossa M. C. Security of IoT Application Layer Protocols: Challenges and Findings. Future Internet, 12(3), 55. 2020.

[13] V Seoane, Carlos García-Rubio, F Almenares, C Campo. (3 août 2021). Performance evaluation of CoAP and MQTT with security support for IoT environments. Carlos III de Madrid, Av. de la Universidad, 30, Leganés (Madrid), Espagne. 2021.

[14] Shahwan, A., Mohammed Z., Smith, B. K. (3 janvier 2025). Enhancing IoT communication security: Analysis and mitigation of vulnerabilities in MQTT, CoAP, and XMPP protocols. Authorea.2025.

[15] SEGARRA C., DELGADO G. R., SCHIAVONI V. MQT-TZ: Hardening IoT Brokers Using ARM TrustZone. 2022

[16] HANIF A., ILYAS M. (2) Effective Feature Engineering pour MQTT Security. Sensors, 24(6) :1782. 2024.

[17] Di Paolo E., Bassetti E., Spognardi A. Security assessment of common open source MQTT brokers and clients. In Proceedings of the Italian Conference on Cybersecurity (ITASEC 2023).

[18] TOÉ É. Renforcement de la cyber-résilience des brokers MQTT contre les attaques DoS et les défaillances grâce à une architecture décentralisée basée sur la blockchain et smartcontrat. Université du Québec à Chicoutimi. 2024.

[19] Buccafurri F., De Angelis V., Lazzaro S., Vangala A. MQTT-E: E2E encryption in MQTT via proxy re-encryption avoiding broker overloading. Ad Hoc Networks, 176, Article 103878. 2025.

[20] De Rango F., Spina M. G., Iera A. DLST-MQTT : Dynamic and lightweight security over topics MQTT. Future Generation Computer Systems, 166, Article 107625. 2025

[21] Alencar R. C., Fernandes B. J. T., Lima P. H. E. S., Silva C. M. R. da. AI techniques for automated penetration testing in MQTT networks. International Journal of Computers and Applications, 47(1), 1–16. 2024.

[22] Swain M., Tripathi N., Sethi K. Identifying communication sequence anomalies to detect DoS attacks against MQTT. Computers & Security, 105, Article 104526. 2025.

[23] Rodríguez-Muñoz J. D., Tlelo-Cuautle E., de la Fraga L. G. Chaos-based authentication of encrypted images under MQTT for IoT protocol. Integration, 102(4), Article 102378. 2025.

[24] HADEMINE A. V. E., Démonstration de la conjecture de Syracuse. Nantes Univ – UFR ST – Nantes Université – UFR des Sciences et des Techniques, 2024.