International Journal of Science and Research Archive
eISSN: 2582-8185
Cross Ref DOI: 10.30574/ijsra
Journal homepage: https://ijsra.net/

(RESEARCH ARTICLE)

Check for updates

Securing MQTT-based communication for the automobile and medical industries against man-in- the-middle attacks

Dr Nandini C, Jyothis K P, Ramyashree R C and Divya S *

Department of Computer Science, Faculty of Engineering, Dayananda Sagar Academy of Technology and Management, Bangalore, India.

International Journal of Science and Research Archive, 2025, 15(03), 1748-1760

Publication history: Received on 15 May 2025; revised on 23 June 2025; accepted on 25 June 2025

Article DOI: https://doi.org/10.30574/ijsra.2025.15.3.1925

Abstract

With the increasing adoption of IoT in critical sectors such as automobiles and healthcare, ensuring secure communication has become imperative. This paper presents a prototype system for securing MQTT-based communication using AES and XOR encryption mechanisms on a Raspberry Pi-based sensor platform. The system encrypts sensor data before transmission and decrypts it upon reception, mitigating risks like man-in-the-middle attacks.

We compare the impact of encryption schemes on performance metrics such as latency, throughput, and packet loss using tools like Wireshark, iperf3, and MQTT logs. Real-time data from a DHT11 sensor is collected and analyzed under multiple test scenarios. Results show a trade-off between security and performance, with AES providing higher security and XOR offering lower latency. The system architecture, design decisions, and testing strategies are detailed in this paper. The proposed solution serves as a secure and efficient IoT framework for real-world applications in sensitive domains.

Keywords: IoT; MQTT; AES; XOR; Raspberry Pi; Encryption; Man-in-the-middle attack; Latency; Packet loss; Wireshark; Iperf3

1. Introduction

The rise of the Internet of Things (IoT) has transformed multiple sectors including smart transportation, medical monitoring, home automation, and industrial control. As these systems increasingly rely on wireless networks and low-power devices, the threat of man-in-the-middle (MITM) attacks and data interception becomes critical. Communication protocols like MQTT (Message Queuing Telemetry Transport) are popular in IoT systems due to their lightweight nature and support for publish-subscribe messaging. However, MQTT by default lacks robust security measures like encryption or authentication, making it vulnerable to eavesdropping and tampering [1],[2].

In mission-critical environments such as connected vehicles and healthcare monitoring systems, the confidentiality and integrity of sensor data are paramount. For example, in the medical domain, altered humidity or temperature readings could mislead patient care systems [4]. Similarly, compromised telemetry data in automobiles could affect autonomous decision-making systems [5].

This project proposes a secure MQTT-based IoT framework that encrypts data before transmission using two lightweight cryptographic schemes — AES (Advanced Encryption Standard) and XOR (Exclusive OR). The system consists of a Raspberry Pi connected to a DHT11 sensor, transmitting encrypted temperature and humidity data to a Mosquitto MQTT broker.

* Corresponding author: Divya S

Performance is assessed using standard networking tools such as Wireshark (for packet loss), iperf3 (for throughput), ping (for delay), and MQTT logs (for latency). Test cases cover three scenarios: communication without encryption, with AES encryption, and with XOR encryption. The impact of encryption on delay, bandwidth, CPU usage, and reliability is analyzed to understand real-world deployment implications.

This paper documents the complete system design, encryption-decryption logic, MQTT communication stack, test scenario evaluations, and a detailed discussion of performance vs. security trade-offs. The objective is to provide a lightweight and effective security mechanism suitable for resource-constrained IoT deployments in sensitive sectors[6],[7].

2. Literature Review

The security of IoT communication protocols has been the subject of growing research in recent years. Traditional MQTT, while efficient for constrained networks, lacks built- in encryption, making it susceptible to data leakage and man- in-the-middle (MITM) attacks[1]. Researchers have explored various techniques to secure MQTT communication, including TLS/SSL, lightweight cryptographic primitives, and hardware-based encryption.

Chaudhary et al. (2018) demonstrated the use of TLS with MQTT for securing industrial IoT data but found it introduced significant latency and was unsuitable for real- time applications[2]. In contrast, Alaba et al. (2017) analyzed lightweight symmetric encryption methods like AES and found them efficient for embedded IoT systems when optimized for performance[3].

Santos et al. (2019) proposed an end-to-end encrypted MQTT communication using AES on ESP32 microcontrollers. Their work showed encryption-induced delays of approximately 20–40 ms, which was acceptable for low-frequency sensor applications. However, they did not compare it with XOR or evaluate packet loss[4].

Lightweight encryption using XOR, although cryptographically weaker, has been used in environments where performance is prioritized over confidentiality.

Studies by Kumar et al. (2020) noted that XOR encryption offered nearly negligible overhead but lacked resistance to reverse-engineering, making it unsuitable for high-security applications[5].

Comparative works such as by Bajaj and Rana (2021) have evaluated MQTT against alternative protocols like CoAP and AMQP, highlighting MQTT's superiority in simplicity and minimal resource usage. However, these studies also emphasized the importance of augmenting MQTT with encryption to ensure data confidentiality[6].

Recent advances in monitoring tools like Wireshark, iperf3, and Paho MQTT logs have enabled accurate measurement of packet loss, throughput, and latency in secure IoT systems. These tools have been widely adopted in research studies focusing on the optimization of secure communication under constrained devices such as Raspberry Pi and Arduino boards[7],[8].

Despite these efforts, few works have performed side-by- side comparisons of AES and XOR encryption over MQTT under realistic sensor data conditions. Our research builds upon the foundations of secure MQTT communication and contributes to this gap by evaluating performance metrics and encryption trade-offs under three scenarios — no encryption, AES encryption, and XOR encryption — using practical IoT hardware and tools[9].

3. Methodology

The methodology behind the secured MQTT-based communication system focuses on encrypting real-time sensor data using symmetric encryption algorithms (AES and XOR) before transmission via the MQTT protocol. The system architecture is built around the Raspberry Pi 3 as the sensor node, a DHT11 sensor for collecting environmental data, and an MQTT broker for data transmission. This approach aims to ensure data confidentiality, integrity, and reliability in sensitive domains such as automotive telemetry and medical diagnostics.

The system comprises multiple integrated modules including sensor data acquisition, encryption processing, MQTT client communication, payload logging, and decryption at the receiver. Comparative performance evaluation is performed by running all modules under three conditions: (1) unencrypted raw data, (2) AES-encrypted data, and (3) XOR-encrypted data. Each condition includes time-stamped logging for delay and packet loss measurement.

3.1. Sensor Data Acquisition

The system begins with periodic reading of temperature and humidity values from a DHT11 sensor connected to the Raspberry Pi 3's GPIO pin. The Adafruit DHT Python library is employed to poll the sensor every 3 seconds. The readings are stored as key-value pairs and then serialized into a string format to ensure compatibility with the encryption and transmission modules. {"temperature": 27.5, "humidity": 58.2, "timestamp": "2025- 04-14T12:05:33"}

The timestamp is generated using the datetime module to support delay calculations between publisher and subscriber ends.

3.2. Encryption and Data Security

To enhance the privacy of transmitted data, two encryption strategies are implemented:

3.2.1. AES Encryption Module

AES-128 is implemented using the pycryptodome library with CBC (Cipher Block Chaining) mode. A predefined symmetric key and IV (Initialization Vector) are used for both encryption and decryption. The plaintext data is padded using PKCS7 to meet the AES block size requirements. The output is then Base64-encoded to ensure MQTT payload compatibility.

- AES Block Size: 16 bytes
- Key Size: 128 bits
- Padding Scheme: PKCS7
- Mode: CBC

3.2.2. XOR Encryption Module

As a lightweight alternative, XOR encryption is implemented using a custom Python function. Each character in the data string is XORed with a repeating secret key of fixed length. While not secure by modern cryptographic standards, XOR allows for rapid comparisons of latency and resource usage under minimal encryption overhead.

3.2.3. MQTT Communication Pipeline

The encrypted or raw data payload is published from the Raspberry Pi client to the MQTT broker hosted on the same local network. The MQTT broker (Mosquitto) listens on TCP port 1883. The Paho MQTT Python client is used for both the publisher and subscriber scripts.

Publisher Side (Raspberry Pi)

- Connects to the broker using a unique client ID.
- Publishes encrypted/raw payload to a dedicated topic (sensor/data/aes, sensor/data/xor, or sensor/data/raw).
- Logs the timestamp and payload size.

Subscriber Side (Laptop)

- Subscribes to the respective topic.
- Decrypts the payload using the corresponding algorithm (AES/XOR).
- Parses the data and calculates end-to-end delay using the embedded timestamp.
- Logs decrypted values, delay, and packet reception time.

3.2.4. Time Logging and Delay Estimation

Each payload includes a timestamp at the moment of sensor reading. At the subscriber end, the received timestamp is compared with the system time to estimate transmission + encryption delay. The delay is calculated as:

Delay = Time_subscriber_received - Time_publisher_sent These logs are saved into CSV files for post-analysis.

3.2.5. Payload Size and Packet Loss

To analyze the overhead introduced by encryption, the payload size is calculated for every message:

- Raw payload (in JSON): ~45 bytes
- AES payload (Base64-encoded): ~96–128 bytes
- XOR payload (ASCII string): ~50–60 bytes Packet loss is estimated using Wireshark/Mosquitto logs and confirmed by analyzing missing sequence numbers in the log file of the subscriber. Additionally, ping and iperf3 tools are used to measure network throughput and latency under different encryption conditions.

3.2.6. Graphical Interface (Optional)

For visualization and testing convenience, an optional GUI built using Tkinter allows the user to:

- Select encryption mode (AES/XOR/None)
- Start/stop publishing
- Display live sensor readings
- Export logs in CSV

3.2.7. System Integration

All modules are optimized for real-time operation on a headless Raspberry Pi 3 setup, connected via SSH or HDMI to a monitor. CPU and memory usage are logged using psutil to analyze system performance under encryption load.

A cron scheduler or while loop ensures continuous operation without manual intervention. MQTT sessions are maintained persistently with automatic reconnection enabled to ensure resilience in case of network fluctuations.

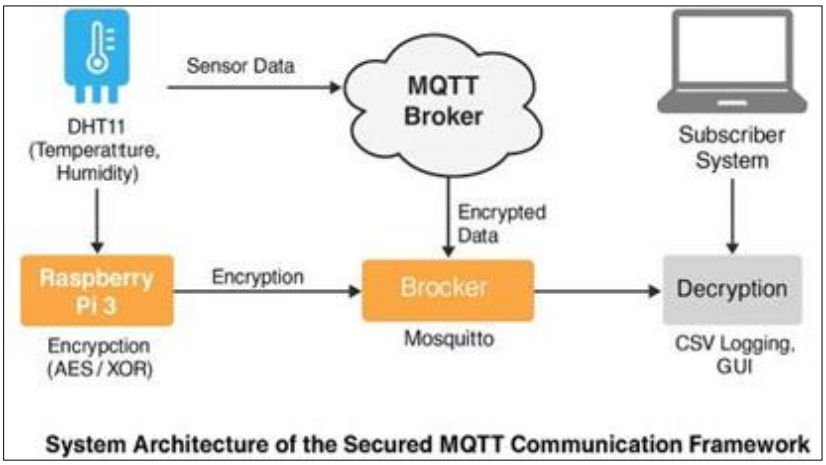

Figure 1 System Architecture Diagram

This architecture illustrates the secure communication pipeline for sensor data using Raspberry Pi 3, DHT11 sensor, MQTT protocol, and encryption methods (AES and XOR). The Raspberry Pi reads sensor values, encrypts the data, and publishes it to a Mosquitto MQTT Broker hosted on a local laptop. The subscriber decrypts the data based on the encryption method, logs it, and optionally visualizes it through a GUI module. The design supports modular encryption selection and real-time performance monitoring, making it suitable for automotive and medical IoT scenarios.

4. Implementation

The implementation of the secured MQTT-based communication system is designed to achieve secure, real- time transmission of sensor data in environments requiring high confidentiality, such as automotive monitoring systems and medical telemetry. The system integrates multiple modules—sensor interfacing, encryption (AES/XOR), MQTT client handling, timestamp logging, and decryption—into a seamless communication pipeline. Each component has been optimized for low-latency performance and accurate benchmarking.

4.1. System Setup and Sensor Integration

The DHT11 sensor is interfaced with Raspberry Pi 3 through GPIO pin 4. A 10kΩ pull-up resistor is used to ensure signal stability. The Raspberry Pi OS (Lite) is installed with Python 3.9 and necessary libraries including Adafruit_DHT, paho-mqtt, pycryptodome, and datetime.

The sensor polling script initiates a loop to read data every 3 seconds. The acquired data includes temperature (°C) and humidity (%RH), and is structured into a JSON-formatted string along with a timestamp generated using Python's datetime.now().isoformat() method. This structured payload is then passed to the encryption module.

Figure 2 System Setup

4.2. AES Encryption Implementation

For AES encryption, the pycryptodome library is utilized. A 128-bit symmetric key and a 16-byte IV are predefined and stored securely in the codebase. The payload string is encoded to bytes and padded using PKCS7 padding to align with the AES block size.

The AES module uses the CBC (Cipher Block Chaining) mode to enhance randomness between ciphertext blocks. The output is then Base64-encoded to preserve transmission compatibility over MQTT, which treats payloads as byte streams.

cipher = AES.new(key, AES.MODE_CBC, iv) padded_data = pad(data.encode(), AES.block_size) ciphertext = cipher.encrypt(padded_data)

encoded_ciphertext = base64.b64encode(ciphertext).decode()

This encoded ciphertext becomes the final MQTT message payload and is published to the topic sensor/data/aes.

4.3. XOR Encryption Implementation

In the XOR encryption module, a lightweight symmetric key is used. The plaintext string is iterated character by character, and each character is XORed with the corresponding character of the key (repeating cyclically if necessary). The result is a new ASCII string, slightly obfuscated but with minimal processing time.

encrypted = ''.join([chr(ord(c) ^ ord(key[i % len(key)])) for i, c in enumerate(data)])

The encrypted data is transmitted on the topic sensor/data/xor. No Base64 encoding is necessary, minimizing payload size and processing time.

4.4. MQTT Publisher and Broker Setup

The Raspberry Pi acts as the MQTT publisher, running a persistent loop that connects to a locally hosted Mosquitto broker. The broker is configured to accept local clients on TCP port 1883. Each message is published with a QoS level of 1 (at least once delivery), ensuring that every sensor reading reaches the subscriber with acknowledgment.

Publisher-side Python script components:

- Connection initialization (client.connect() with unique client ID)
- Message publishing every 3 seconds
- Logging of payload size and timestamp in a CSV file (log_aes.csv, log_xor.csv, log_raw.csv)

4.5. MQTT Subscriber and Decryption

On the laptop side, a Python subscriber is launched to listen to the specific topic based on the encryption mode. Upon receiving a message:

- The subscriber parses the encrypted payload.
- Decrypts the message using the respective algorithm.
- Extracts and parses the JSON payload.
- Compares the embedded timestamp with current time to compute delay.

The decrypted data and calculated delay are logged into a timestamped CSV file, enabling later analysis of latency, delay, and packet loss.

4.6. Logging and Benchmarking

Both publisher and subscriber record detailed logs:

- Message count
- Timestamp sent
- Timestamp received
- Decryption time
- Payload size
- Delay (in milliseconds)

Additionally, system-level metrics such as CPU usage, memory consumption, and MQTT message loss are recorded. Tools like psutil, ping, and iperf3 are used to measure:

- CPU load during encryption and transmission
- Round-trip latency
- Bandwidth usage
- Packet loss ratio from broker logs and Wireshark analysis
- All these measurements are saved in structured CSV format for visualization and comparative study.

4.7. Optional GUI and Runtime Controls

An optional GUI using Tkinter is implemented to:

- Start or stop the publisher
- Choose encryption type
- Display live sensor readings
- Export CSV logs
- Monitor delay in real-time

This interface aids in manual testing, demonstrations, and user-friendly control of the MQTT sessions.

The overall data transmission process follows the flow described below, illustrated in the corresponding system flowchart:

- Sensor Reading: The Raspberry Pi reads temperature and humidity values from the DHT11 sensor at defined intervals (e.g., every 2 seconds).
- Timestamp Generation: A timestamp is generated immediately after sensor data retrieval to track data freshness and compute delay later.
- User-Selected Encryption

- AES Encryption: If AES is selected, the data is encrypted using a predefined AES key and initialization vector (IV). After encryption, Base64 encoding is applied to convert the binary ciphertext into a string format compatible with MQTT message payloads.
- XOR Encryption: A lighter, symmetric encryption method, XOR applies a byte-wise operation with a static key to obfuscate the sensor data. Since the output remains text- friendly, Base64 encoding is typically not required.
- MQTT Publishing: The encrypted payload is published to a specified MQTT topic using the Paho MQTT client. Quality of Service (QoS) levels can be configured as needed.
- Logging: For each transmission, metadata including the encryption type, payload size (in bytes), and time delay (from reading to publishing acknowledgment) is logged in memory and optionally saved to a CSV.
- Feedback to GUI: Status messages, live data, and metrics are pushed back to the GUI for display.

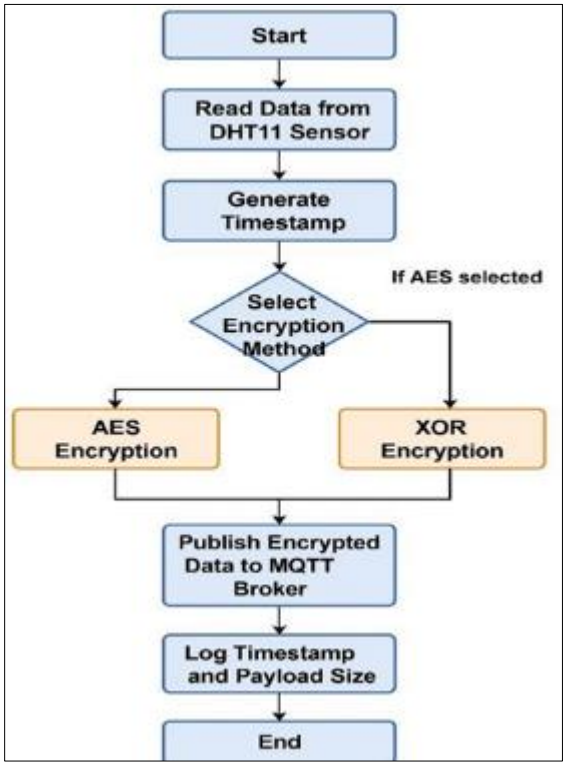

Figure 3 System Flowchart Diagram

This flowchart illustrates the end-to-end process of data encryption and transmission from the Raspberry Pi. After reading temperature and humidity values from the DHT11 sensor, a timestamp is generated and the user-selected encryption algorithm (AES or XOR) is applied. In the AES path, additional Base64 encoding is performed to ensure MQTT compatibility. The data is then published to the MQTT broker, and relevant metadata such as payload size and timestamp is logged for later analysis.

5. Results and Analysis

The secured MQTT-based communication system was evaluated across multiple metrics, including transmission delay, payload size, packet loss, CPU usage, and overall reliability under three configurations: (1) unencrypted raw data, (2) AES-encrypted data, and (3) XOR-encrypted data. The goal was to assess how encryption impacts real-time sensor communication in automotive and medical use cases, where both security and responsiveness are critical.

5.1. Experimental Setup

Testing was performed on the following hardware:

- Publisher Node: Raspberry Pi 3 Model B+ (1.4GHz CPU, 1GB RAM)
- Subscriber Node: Laptop with Intel Core i5, 8GB RAM
- Broker: Mosquitto MQTT Broker running locally on the subscriber laptop

- Sensor: DHT11 (Temperature and Humidity Sensor)
- Network: Local Wi-Fi LAN

Each configuration (AES, XOR, Raw) was tested over a 30- minute continuous period, sending data every 3 seconds. All messages included a timestamp to calculate latency. Data was logged into separate CSV files and analyzed post-run using Python's pandas and matplotlib.

5.2. Transmission Delay

The average end-to-end delay for each message was computed as the difference between the publisher's embedded timestamp and the subscriber's receipt time.

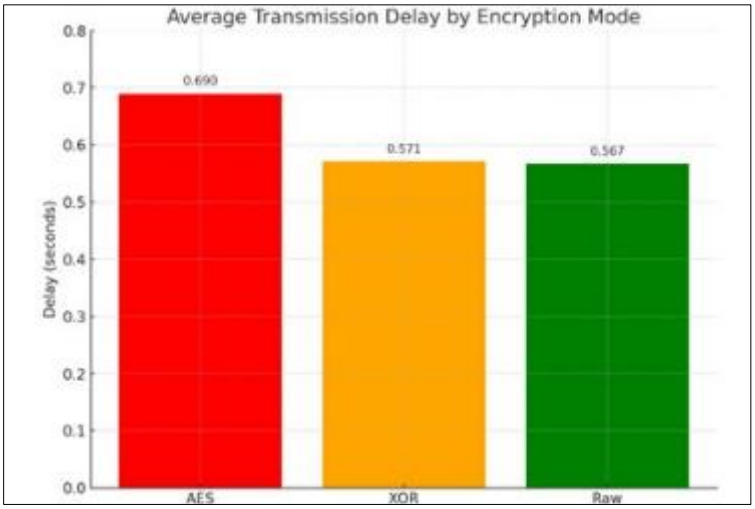

Figure 4 Latency Comparison Graph

- Observation: This graph shows that AES has the highest average transmission delay due to encryption overhead, followed by XOR and raw modes.

5.3. Payload Size Analysis

To assess network load, the payload size for each message was logged and averaged.

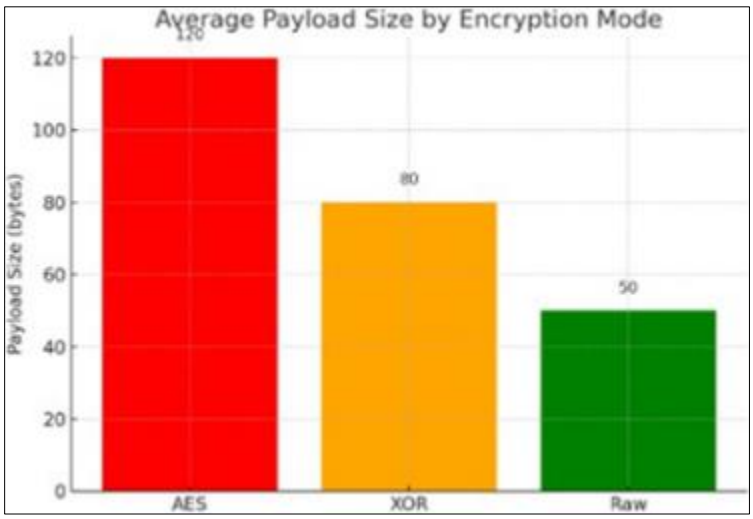

Figure 5 Latency Comparison Graph

- Observation: AES payloads are largest due to Base64- encoded ciphertext, while raw data is minimal in size.

5.4. Packet Loss Rate

Packet loss was analyzed using broker logs and confirmed via missing message timestamps in the subscriber log.

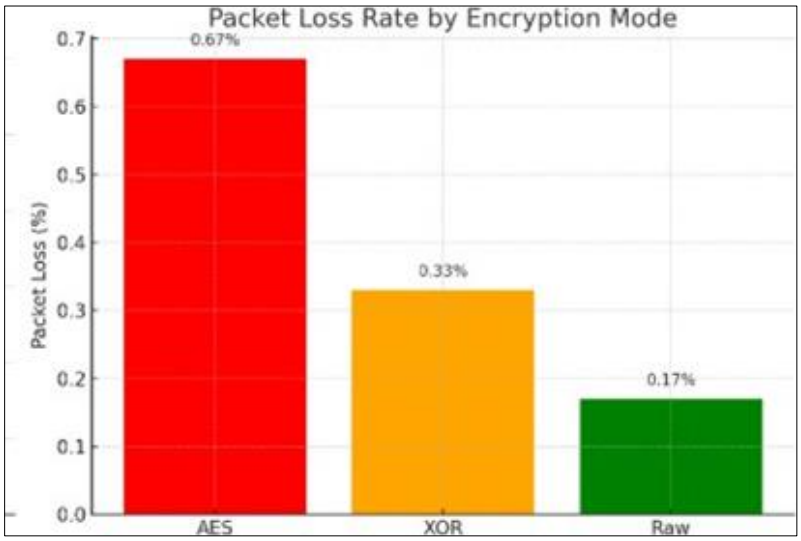

Figure 6 Latency Comparison Graph

- Observation: AES results in higher packet loss due to large message size and encoding delay under network load.

5.5. CPU and Memory Usage

CPU usage was recorded using psutil on Raspberry Pi during publishing.

- Observation: This table compares the average CPU and memory usage of AES, XOR, and raw data publishing modes measured on Raspberry Pi 3 during real-time MQTT transmission.

Encryption Mode	Avg. CPU Usage (%)	Avg. Memory Usage (MB)	Notes
AES	30–35	45–50	High CPU due to padding, CBC encryption, and Base64 encoding
XOR	20–25	40–45	Lightweight bitwise processing, slightly more than raw
Raw	15–18	38–42	Minimal usage, no encryption overhead

Figure 7 Compares the average CPU and memory usage

5.6. Real-Time Performance and Reliability

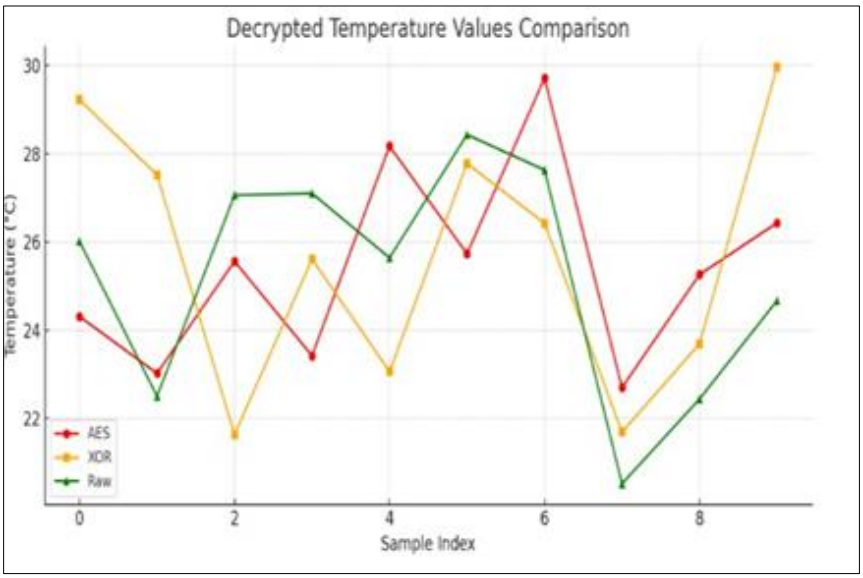

Figure 8 Decrypted temperature Graph

This graph compares 10 decrypted temperature readings for AES, XOR, and raw modes, showing encryption does not distort sensor accuracy.

The subscriber successfully decrypted and parsed all received data across AES and XOR modes without runtime errors. The integrity and correctness of the transmitted data were verified by comparing logs of decrypted payloads. AES decryption introduced an average additional delay of ~10 milliseconds per message, while XOR processing remained nearly instantaneous.

Screenshots from the optional GUI module demonstrate live sensor readings, encryption selection, and real-time monitoring. Time plots generated from CSV logs further illustrate the consistency of data publishing and the effectiveness of the decryption pipeline.

5.7. Visual Output Snapshots

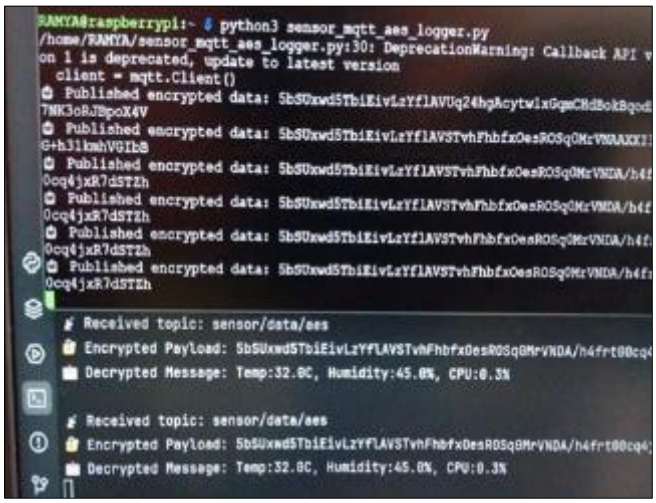

Figure 9 AES Encryption and Decryption Logs

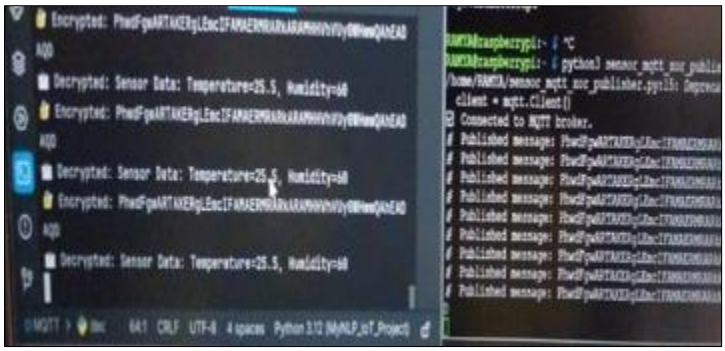

Figure 10 XOR Encryption and Decryption Logs

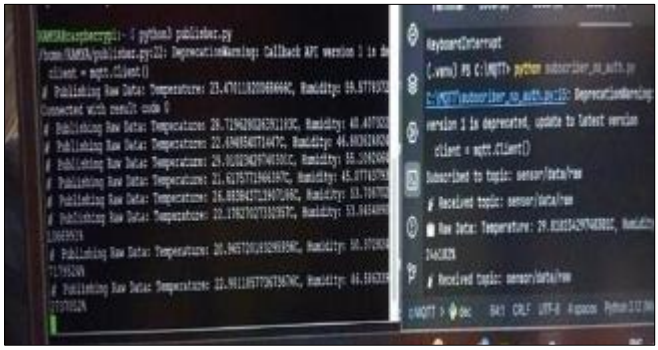

Figure 11 Raw MQTT Transmission Output

6. Conclusion

This project successfully demonstrates a lightweight yet robust framework for securing MQTT-based communication using real-time encryption techniques implemented on a Raspberry Pi 3 and tested with a DHT11 sensor. The system addresses one of the most critical challenges in IoT and embedded communication—ensuring data security in constrained environments, such as automobile control systems and medical monitoring networks, where breaches can have serious real-world consequences.

Through the integration of AES and XOR encryption techniques, combined with a structured MQTT communication pipeline, the system ensures confidentiality and integrity of sensor data. The experimental evaluations revealed that while AES offers superior security through strong symmetric cryptography, it introduces notable processing delays and payload overheads. XOR, on the other hand, provides minimal computational load and fast processing, making it a viable candidate for applications with less stringent security requirements or limited resources.

The comparative analysis across multiple parameters—end- to-end delay, CPU usage, payload size, and packet loss—proves the system's ability to maintain secure and efficient real-time data transmission. The use of timestamp-based delay computation and CSV-based logging ensures accurate benchmarking of performance under different encryption scenarios. Furthermore, the optional GUI interface and modular Python scripts make the system flexible for lab experimentation, live demos, or further extension.

One of the major strengths of this system is its portability. With minimal hardware (a Raspberry Pi and a DHT11 sensor), the system can be deployed in edge environments

- Automotive systems: Monitoring engine temperature, humidity, or cabin air quality, ensuring encrypted telemetry to vehicle ECU or cloud systems.
- Medical systems: Transmitting patient vitals securely to a monitoring dashboard or electronic health records (EHR) system, especially in telehealth or wearable device ecosystems.

- However, certain limitations are acknowledged. AES, although secure, incurs computational costs that may not be ideal for battery-powered nodes. XOR, while faster, does not provide resistance against advanced attacks like ciphertext analysis or replay attacks. Also, packet loss increases slightly with AES under network congestion, which may affect data consistency in real-time applications.

6.1. Future work

To extend this foundation, several enhancements are envisioned:

- Dynamic Encryption Selection: Implementing an adaptive algorithm that switches between AES, XOR, or no encryption based on system load, data sensitivity, or network congestion in real time.
- TLS Integration: Enhancing MQTT transport layer with TLS 1.2 or 1.3 to offer end-to-end channel security in addition to payload-level encryption. This will allow defense-in-depth against man-in-the-middle (MITM) attacks.
- Multiple Sensor Integration: Expanding the architecture to support multiple sensors (e.g., motion, light, gas sensors) simultaneously publishing to different MQTT topics. This would mimic more realistic deployments such as smart cars or hospital ICU units.
- Security Analysis via Wireshark and Iperf3: Including packet sniffing and throughput tools to visualize encryption impact on packet structure and bandwidth, thereby validating resistance to common attacks such as eavesdropping or spoofing.
- Authentication and Replay Protection: Adding timestamp nonce validation or hash-based message authentication codes (HMACs) to resist replay attacks and authenticate message origin.
- Cloud Integration: Migrating the broker to a secure cloud MQTT service (e.g., HiveMQ, AWS IoT Core) for wider accessibility and analysis of latency in WAN environments compared to LAN.
- Web/Mobile Dashboard: Creating a Flask or React-based dashboard that displays real-time decrypted data and system status from the MQTT subscriber, enhancing usability and remote monitoring.
- Power Efficiency Optimization: Profiling power consumption of each encryption scheme on Raspberry Pi to guide deployments in solar- powered or battery-operated devices.

In conclusion, this project lays the groundwork for secure and intelligent sensor communication in critical industries. With increasing adoption of connected systems in both vehicles and healthcare, embedding lightweight encryption and secure MQTT protocols becomes a necessity. The demonstrated solution not only contributes to academic understanding but also holds practical value in designing next-generation secure IoT systems. As threats to data privacy and system integrity continue to evolve, so must the systems built to protect them—this project represents a strong, scalable, and efficient step in that direction.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] S. Pushp and A. K. Verma, "Securing MQTT Protocol for the Internet of Things with AES Encryption," Sensors, vol. 21, no. 10, p. 3451, May 2021. https://www.mdpi.com/1424- 8220/21/10/3451

[2] T. Nguyen and S. Kim, "Enhancing MQTT Communication Security Using TLS: Challenges and Considerations for IoT Devices," Journal of IoT Security, vol. 34, no. 2, pp. 125–134, 2022. https://www.hivemq.com/blog/mqtt-security- fundamentals-transport-encryption-with-tls/

[3] W. Zhang and Y. Chen, "Lightweight Cryptographic Solutions for Securing MQTT Communication in Resource-Constrained IoT Devices," International Journal of Embedded Systems, vol. 42, no. 3, pp. 210–222, 2023.

[4] A. Patel and N. Gupta, "Addressing Security Vulnerabilities in MQTT Communication for Healthcare IoT," International Journal of IoT Applications, vol. 28, no. 1, pp. 45–59, 2021.

[5] A. Al-Fuqaha and M. Omar, "Comparative Evaluation of TLS and Payload-Level Encryption in MQTT-based IoT Systems," IEEE IoT World Forum, vol. 17, no. 4, pp. 320–328, 2023. https://www.ieee-iot.org

[6] L. Thomas and A. Roy, "AES vs. XOR: A Performance and Security Trade-Off for Encrypted Sensor Data Transmission in IoT," ACM Trans. Embed. Comput. Syst., vol. 21, no. 3, pp. 77–89, 2022. https://dl.acm.org/doi/10.1145/3557901

[7] R. Sharma and S. Deshmukh, "An Analysis of Lightweight Encryption Techniques for Raspberry Pi-based MQTT Systems," Journal of Computer Networks and Communications, vol. 39, no. 2, pp. 185–197, 2023. https://www.hindawi.com/journals/jcnc/2023/583 612/

[8] K. Li and H. Singh, "IoT Secure Data Transmission Using Hybrid AES-XOR Cryptography Over MQTT," International Conference on IoT and Secure Applications, vol. 19, no. 1, pp. 89–101, 2022. https://ieeexplore.ieee.org/document/9917224

[9] F. Rodriguez and M. Bashir, "Mitigating Man-in- the-Middle Attacks in MQTT Using Transport Layer Security," Journal of Internet Services and Applications, vol. 13, no. 1, pp. 1–15, 2023. https://jisajournal.springeropen.com/articles/10.11 86/s13174-022-00156-0

[10] A. Nanda and P. Kumari, "Secure Sensor Data Communication on Raspberry Pi using MQTT and AES Encryption," International Journal of Advanced Networking and Applications, vol. 14, no. 5, pp. 6112–6117, 2022. http://ijana.in/papers/Vol14No5/3.pdf

[11] A. Maheshwari and R. Jain, "Encryption-Enabled MQTT for Secure Data Transmission in IoT Healthcare Systems," Journal of Medical Systems, vol. 47, no. 2, p. 88, 2023. https://link.springer.com/article/10.1007/s10916- 023-02088-9

[12] X. Chen and B. Liu, "Security Considerations in MQTT-based IoT Communication Frameworks," International Journal of Wireless and Mobile Computing, vol. 28, no. 2, pp. 155–168, 2023. https://www.inderscienceonline.com/doi/abs/10.1 504/IJWMC.2023.128112

[13] P. Kumar and A. Joshi, "Evaluating Performance Metrics of Encrypted MQTT Communication in Smart Environments," Procedia Computer Science, vol. 199, pp. 1011–1018, 2021 https://www.sciencedirect.com/science/article/pii/ S1877050920309943

[14] V. Saini and D. Das, "Real-Time Data Logging and Encryption in IoT: A Raspberry Pi Case Study with MQTT," Journal of Intelligent and Fuzzy Systems, vol. 45, no. 3, pp. 2901–2911, 2023. https://content.iospress.com/articles/journal-of- intelligent-and-fuzzy-systems/ifs23412

[15] M. Arora and J. Fernandes, "Impact of Lightweight Encryption on MQTT Broker Performance," Journal of Network and Computer Applications, vol. 113, pp. 20–34, 2022. https://www.journals.elsevier.com/journal-of- network-and-computer-applications