Testing Security and Performance of MQTT Protocol on Raspberry Pi for IoT Applications

DANIL VOROTILOV, University of Twente, The Netherlands
SUPERVISOR: DR. ING. M.ELHAJJ (MOHAMMED), University of Twente, The Netherlands

Abstract - One of the core requirements for IoT Environments is an M2M protocol, through which devices can communicate with each other. One such protocol, which is widely used in IoT environments is MQTT. MQTT's lightweight design makes it ideal for resource-constrained IoT applications, and Raspberry Pi's affordability and versatility have positioned it as a popular platform for such projects. This research investigates the security and performance of the MQTT protocol on Raspberry Pi 4 model B. To achieve this goal, IoT environments with several levels of bandwidth will be created and analyzed. The Raspberry Pi will be running the broker on a Linux instance. Virtual instances of Linux will act as the clients. Then an analysis of performance and security will be conducted on the data collected from the experiments.

Additional Key Words and Phrases: IoT, MQTT, M2M, Raspberry Pi, Security, Protocol, Linux

1 INTRODUCTION

Internet of Things (IoT) [10] refers to the interconnection of everyday objects with embedded sensors, software, and internet connectivity. These "smart" devices transcend their traditional functionalities, acquiring the ability to collect and transmit data, communicate with each other, and even respond autonomously based on pre-programmed rules or machine learning algorithms. For such devices to communicate with each other, some networking protocol must be used. Many IoT environments are resource-constrained, whether it is limited bandwidth or limited memory of devices. The lightweight MQTT protocol was created exactly for such environments.

1.1 Background

MQTT was first created to monitor oil pipelines Andy Stanford-Clark was made in 1998 [2]. The goal of MQTT was to be bandwidth-efficiency, lightweight, work in unstable network environments, and use little battery power, whereas other application-layer protocols such as HTTP wouldn't work because of their large packet size [3]. To understand how these goals were achieved we should look at the MQTT packet structure. An MQTT packet consists of three parts. The fixed header, variable header, and payload. Both variable header and payload are optional parts of the packet, so in theory the smallest MQTT packet is only 2-5 bytes [4] in length depending on the 'packet type' field.

1.2 Motivation

The Message Queuing Telemetry Transport (MQTT) protocol has emerged as the dominant messaging protocol within the Internet of Things (IoT) domain. Much of the literature is about the performance and security of the protocol. There isn't much research on implementing and analyzing the security of the MQTT protocol on the Raspberry Pi. This is the main goal of this research. To analyze the security, performance, and scalability of the MQTT protocol when the server is run on a Raspberry Pi.

1.3 Structure

This paper splits the major research question into three smaller research questions in section 2.5. A thorough literature review will be conducted in section 3. After the literature review, the methodology in section 4 will mention the proposed solution and how each experiment will answer the research questions. Then the results will be discussed and compared to previous literature in the section 6.

2 PROBLEM STATEMENT

MQTTs reliability and scalability make it a popular choice for IoT systems [7]. Due to its popularity, it is a major target for attackers attempting to steal valuable info or sabotage the system's availability. Security is a major concern for the MQTT protocol, and it's one of the problems this research aims to tackle. The other problem this paper will cover is the performance of the MQTT protocol, and how it compares to an alternative protocol, CoAP.

2.1 MQTT

The Message Queue Telemetry Transport (MQTT) protocol is a protocol used for M2M communication within an IoT environment [12]. It is a publish-subscribe protocol, meaning it utilizes the roles of publisher and subscriber. In the MQTT protocol, the publisher publishes to a designated group on the server. The server then redirects the payload to all endpoints which are subscribed to the publisher's group. When compared to other protocols like HTTP, the MQTT protocol has a considerably smaller header size, making it better suited for IoT applications [7].

2.2 CoAP

CoAP is a messaging protocol for resource-constrained networks like those with low-power sensors. It follows a RESTful client-server model where clients request resources and servers respond. Security is bolstered by DTLS and RSA encryption [13]. CoAP offers confirmable (CON) messages, which are packets that require acknowledgment. This ensures the reliable transfer of packets. Additionally, CoAP offers non-confirmable (NON) messages for situations where confirmation is not critical. The protocol also supports various methods for interacting with resources including GET, PUT, POST, and DELETE, mirroring functionality found in HTTP.

TScIT 41, July 5, 2024, Enschede, The Netherlands

2.3 Scope

For the broker and client implementations, open-source projects will be utilized. This makes debugging, and finding vulnerabilities easier, as the source code is visible. The server will run on a Raspberry Pi 4 model b [11]. The Raspberry Pi itself will be running the 64-bit Raspberry Pi OS Lite. The kernel version of the operating system will be 6.6 and the Debian version will be 12. Version 5.0 of the MQTT protocol will be used.

2.4 Aim

This research will aim to make a security analysis of the MQTT protocol, as well as record data on the performance of the MQTT protocol with the use of RPi 4 as the server. The MQTT protocol itself will be analyzed instead of its implementation. This is why pre-existing libraries for the server and the client will be used. Developing a custom MQTT implementation for this project would be unnecessarily complex and is therefore outside of the scope of this research.

2.5 Research Questions

(1) To what extent does the implementation of the MQTT protocol on Raspberry Pi 4 Model B ensure security?
(2) What are the resource utilization implications on the server when subjected to varying Quality of Service (QoS) levels in MQTT communication, with specific attention to CPU clock cycles and memory consumption?
(3) How does the performance of MQTT compare to that of CoAP in terms of latency, throughput, and reliability?

3 LITERATURE REVIEW

Authors in [6] address the critical issue of security in the Internet of Things (IoT) landscape, particularly focusing on the Message Queue Telemetry Transport (MQTT) protocol commonly used for communication among IoT devices. The authors identify a specific vulnerability within MQTT that enables clients to manipulate server behavior, thus introducing SlowITe, a novel low-rate denial of service (DoS) attack tailored to exploit this weakness. Through empirical validation against real MQTT services, both encrypted and plaintext, the authors demonstrate the effectiveness of SlowITe in orchestrating DoS attacks with minimal resources. The paper underscores the significance of the identified vulnerability and calls for further refinement of the MQTT protocol to mitigate such threats. Additionally, it advocates for the development of detection and mitigation systems to safeguard against similar attacks in the future. By conducting tests on various MQTT services and exploring potential enhancements to the SlowITe attack, the study provides valuable insights into the vulnerabilities of MQTT and sets the stage for future research in IoT security.

Building on the work of [5], this paper investigates the application of machine learning for DoS attack detection in MQTT networks. The study aims to identify the most effective method for detecting anomalous device behavior based on MQTT traffic characteristics. The authors propose a modular DoS attack detection system specifically designed for the MQTT protocol. This system collects and stores message-related information, linking traffic data to network addresses, device IDs, and usernames. The researchers then evaluate system performance using the F1-score to determine the optimal detection approach. Their findings suggest that Support Vector Machines with RBF kernel and SMO optimization, or Multilayer Perceptron artificial neural networks, achieve the most effective DoS mitigation. This study's findings can significantly improve the accuracy and efficiency of DoS attack detection in MQTT-based IoT networks.

A performance analysis by [13] compared the Message Queue Telemetry Transport (MQTT) and Constrained Application Protocol (CoAP) protocols. In lossless, unsecured networks, CoAP messages averaged only 226 bytes, while MQTT messages required an average of 1560 bytes. This significant difference stems from the underlying transport protocols: MQTT utilizes TCP, known for its reliability but larger overhead, while CoAP leverages UDP's lighter-weight design. This research underscores the importance of protocol selection for bandwidth-constrained applications in the Internet of Things (IoT). By adopting CoAP, resource-limited devices can conserve valuable network resources, ultimately contributing to a more scalable and efficient IoT ecosystem.

Building upon prior work [8], this study investigates the performance benefits of utilizing the QUIC transport protocol for MQTT messaging. Compared to the traditional MQTT over TCP approach, MQTT over QUIC demonstrates significant improvements in connection establishment efficiency. The research found a 56% reduction in the number of packets exchanged during handshake, directly translating to reduced processor and memory usage on the server side. Specifically, eliminating half-open connections yielded up to 83% lower processor utilization and 50% less memory consumption. Additionally, MQTT over QUIC eliminates head-of-line blocking, a phenomenon that can significantly impact message delivery latency. This study demonstrates a latency reduction of up to 55%, highlighting the potential of QUIC to improve the responsiveness and real-time capabilities of MQTT-based applications. These findings suggest that MQTT over QUIC offers a compelling alternative for scenarios demanding efficient and reliable data exchange in resource-constrained environments. The research in [9] investigated the performance of the Message Queue Telemetry Transport (MQTT) protocol running on a Raspberry Pi Zero W configured as an IoT gateway. The study focused on three key performance indicators: processor temperature, CPU usage, and message reception rate under varying Quality of Service (QoS) settings. The findings revealed that the maximum processor temperature remained well below critical thresholds, ensuring the safe operation of the Raspberry Pi. Interestingly, the number of received messages exhibited some random variation, suggesting potential network or client-side factors at play. CPU utilization remained consistently low throughout the experiments, indicating sufficient processing headroom for the chosen workload. These results demonstrate the suitability of the Raspberry Pi Zero W as a low-power gateway for resource-constrained MQTT-based IoT deployments.

4 METHODOLOGY

To conduct the experiments, an MQTT broker and client will need to be chosen. The broker and client must fit the following requirements:

- Open-source
- Include MQTT version 5.0
- Python library

The reason an open-source broker is a requirement is because open-source projects are more transparent. This is due to the open nature of an open-source project, where source code is publicly available on GitHub. My second requirement is to find a broker and client solution that adheres to version 5.0 of the MQTT protocol. Most of the current literature on the MQTT protocol uses version 3.1.1, therefore utilizing version 5.0 of MQTT would expand the current literature on the subject. The last requirement exists to gather the latency data as explained in section 4.3 of the paper. For this, features of the Python language are needed such as the "time" library to calculate the latency of a packet.

4.1 Proposed Solution

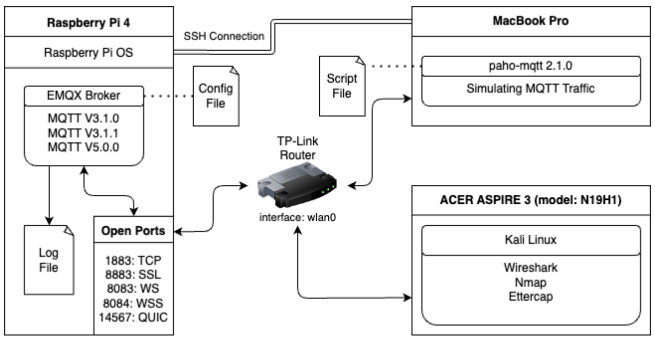

Fig. 1. Testbed setup

Many solutions for an MQTT broker exist. For this research, the EMQX broker will be used. The EMQX broker is made with Erlang and supports MQTT version 5.0. For the client, the Python "mqtt-paho" library will be utilized. This library gives a synchronous and asynchronous API to communicate with the MQTT broker.

4.2 Hardware and Software setup

The broker implementation that will be used is the EMQX broker. EMQX Broker is open-source and made with Erlang. The broker will be running on the Raspberry Pi 4 model b.

An SSH connection will be utilized to control the RPi 4 with the MacBook Pro. Additionally, the MacBook Pro will generate traffic throughout the experiments. For the security analysis, the MacBook Pro will act as the legitimate client.

The ACER laptop will be run on Kali Linux. This Linux distribution is common for security analysis as it provides many tools to aid in security analysis such as Nmap, Ettercap, and Wireshark. Nmap will be used for scanning used ports, Ettercap will be used for ARP spoofing attacks, and Wireshark will be used for monitoring network traffic. Additionally, this laptop will be used to answer the research question in 3, as it is easy to control packet loss probability with the Linux command "tc".

Table 1. Hardware Specification

Broker	Client	Attacker	Router
RPi 4	MacBook Pro	ACER ASPIRE 3	Archer AX10

Table 2. Software Specification

MQTT Broker	MQTT Client	CoAP	Operating System
EMQX	paho-mqtt	aiocoap	RPi OS Lite (64-bit)

All devices will be on the same private network, communicating through the 'wlan0' interface. The router which enables this communication is the tp-link router.

For the research question in section 3, the Python CoAP library will be used called "aiocoap". Several measurements must be conducted regarding the performance of the RPi 4.

4.3 Measurement Metrics

To answer the research questions, the following measurement metrics will be recorded: CPU clock cycles, memory utilization, battery usage, packet latency, throughput, and reliability.

To perform stress-testing for the research question in section 2, a shell script will be utilized to execute the "top" command for 10 iterations. This command will record CPU and memory usage in percentage values. The stress test will consist of a certain number of client instances subscribing to the broker. The number of subscribers will start from 10 and go up to 100, in increments of 10. Throughout all iterations, there will be one publisher, constantly sending as many MQTT packets as possible in a Python script using the paho-mqtt library. The results will be piped into a text file. Then the mean of the 10 instances will be taken, with the equation 2 below.

$$\frac{1}{10}\sum_{n=1}^{10} a_n \tag{1}$$

The top command gives CPU usage concerning the power of 1 core. So when this result reaches 100%, one core is being fully utilized. Since the RPi 4 has four cores, the full CPU usage is represented by 400%.

- **Packet Size:** Wireshark will be utilized to capture packets on 'wlan0' interface. The 'length' field will be used to analyze packet size.
- **For CPU and Memory Usage:** A shell script that will execute the "top" command every 2 seconds and store the output in a log file will be utilized.
- **For Packet Latency:** An Python client library on the ACER will be utilized. The difference between the timestamp of the last byte received and the last byte sent will be displayed in a box plot in milliseconds. These differences will be recorded in several controlled environments. The independent variable in these environments would be packet loss. To simulate packet loss, a Linux command "tc" will be employed before the server processes the packet. The script will drop a certain percentage of packets ranging from 0% to 10% in increments of 2%. The

results will then be displayed in a box plot, where the unit of time measurement will be milliseconds.

5 SECURITY ANALYSIS

Firstly, we will set the security requirements for the MQTT protocol. Security requirements define the essential security goals that the system must achieve. These goals encompass data confidentiality, integrity, and availability. Identifying critical assets and understanding potential vulnerabilities is crucial to establishing these requirements. Then, the security design section will show a possible translation of the established security requirements into a technical implementation. Lastly, threat modeling will be conducted, where a proactive identification of potential threats and vulnerabilities the system might face will be performed.

5.1 Security Requirements

One major goal in security is cryptographic communication. The goal of cryptographic communication is to make sure that the content can only be seen in its original form only by the sender and the receiver. Any third party intercepting the packet should not be able to interpret the content meaningfully.

The availability of the MQTT protocol is an important aspect of the IoT field. Much research has been conducted on conducting DoS attacks towards brokers implementing the MQTT protocol. For example, the authors in [6] identify a specific vulnerability within MQTT that enables clients to manipulate the time the attacker connection will be in a 'alive' state. This DoS attack uses low bandwidth rates, thus the name 'SlowITe' was given. The researchers have attempted to do this attack in MQTT Version 3.1.1.

5.2 Security Design

The MQTT Protocol offers three Quality of Service options. The first option, which has code 0 is "fire and forget". With this option, MQTT packets are sent without being acknowledged. The main advantage of using this configuration is the low overhead of MQTT packets.

Quality of Service options 1 and 2 offer reliable transmission of packets, by using acknowledgments. This mechanism ensures the integrity of the system, by making sure the packets get transmitted with acknowledgments.

MQTT allows cryptographic communication through TLS. If Confidentiality is a security goal for the IoT system, all connections to the broker must be made through TLS. Alternatively, TCP connections could be used for publishing or subscribing to topics that exchange non-crucial data. Any attacker performing a man-in-the-middle attack can intercept MQTT packets over a TCP connection, and be able to see all the MQTT metadata as well as the full payload in plain text. However, TLS has its own drawbacks. The main disadvantage is larger packet sizes. This can be seen in figure 3, which depicts the number of bytes required for a protocol to achieve a certain state.

Table 3. Bytes required to achieve a certain state

Protocol	CON	SUB	PUB	DISC
TCP	194	148	424	268
TLS	3382	262	512	380

5.3 Threat Modeling

MQTT Version 5.0 takes a different approach to clean sessions than version 3.1.1. Instead of a single clean session flag, version 5.0 provides two fields: Clean Start and Session Expiry Interval. Clean Start lets the server know whether the session is new or not, and Session Expire Interval is a field that represents how long the session will expire after the network connection is disconnected. Compared to the specification of version 3.1.1, where the sessions remained on the server indefinitely, the sessions in version 5.0, are removed after Session Expire Interval, freeing up the memory. However, an attacker could set the Session Expire Interval to '0xFFFFFFFF', which would lead to the session never being expired even after the client disconnects. The protocols explain that both the client and server "must" save the session if the client disconnects. Additionally, a "Will Message" is a message that must be stored on the server and associated with a session. MQTT version 3.1.1 suggests Persistent Sessions should never expire [1]. The implementation as per OASIS standard description, a denial of service attack could be employed via the mass creation of Persistent Sessions, causing exhaustion of memory. This would violate the Availability of the MQTT Broker, therefore, in practice, the implementations of the MQTT broker should provide a global configuration where the user can set the session expiration time.

figure 2 shows a graph of CPU % usage in relation to 1 core. Since the RPi 4 has 4 cores, the maximum CPU usage when using the command "top" can be 400%. The graph shows a clear linear relationship between subscribers connected to a topic and CPU usage by the broker.

The figure in 2 shows that connections over TLS were the most CPU demanding. Both graphs are scaled between 0 and 400% on the y-axis and 0 to 100 subscriber connections on the x-axis. The most CPU-intensive TLS traffic for the broker was with 'QOS 0'. The broker reaches a maximum usage of 325.34 at 100 subscribers with 'QOS 0'. Since the maximum CPU usage is 400%, 3.25 out of 4 CPU cores were used for the MQTT broker. This is 81.3% usage of the CPU for 100 subscribers. Using linear interpretation, we can predict how much subscriber count is required to reach 400%.

$$\frac{y_1 - y_{10}}{x_1 - x_{10}} \tag{2}$$

First, we find the slope with the following equation taking the difference of the fraction of the first and the last points. With this, we get a slope of 1.32.

$$1.32x + c = y \tag{3}$$

after substituting x with 10 and y with 206.84, we get the c constant as 193.64. Finally, we have the following equation:

$$1.32x + 193.64 = 400 \tag{4}$$

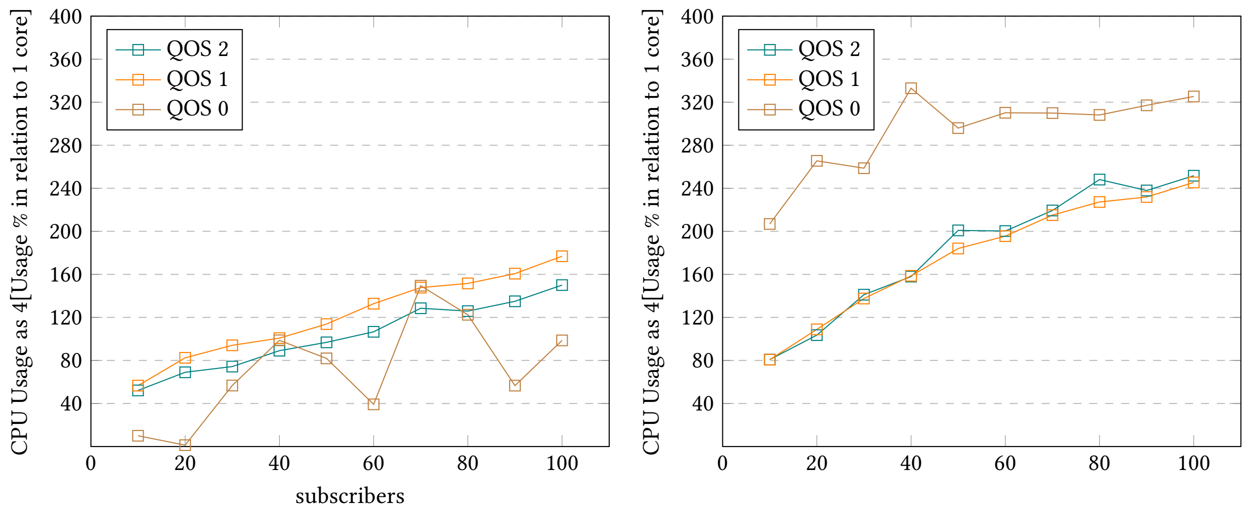

Fig. 2. TCP and TLS CPU utilization

After solving for x, we get 156.33. This means we would require 156 connections to the MQTT Broker to achieve 400% of the CPU usage, meaning all of the CPU usage would be towards the MQTT broker. This would negatively impact the availability of the MQTT Broker.

The figure in 3, shows memory usage in TCP connections remains stable when the subscriber amount is increased.

As can be seen, the memory usage of the EMQX broker is fairly stable. The only noticeable increase happens with "QOS 2", at 100 subscribers, where the memory usage peaks at 13%.

As can be seen by the figure 4, packet loss has a marginal impact on the performance of CoAP. Comparatively, when looking at the figure 4, we can see the biggest deviation from 0% packet loss to 2% packet loss. The median in 2% packet loss stays low at 10.48 ms, however, the 3rd quartile increases to 133.21 ms from 13.40 ms at 0% packet loss. This is a 993% increase, which suggests MQTT is not very good at dealing with packet loss. This is further proven when looking at the median rise in figure 4 as the packet loss % increases.

To calculate throughput, the following equation is used:

$$THROUGHPUT = \frac{MEGABYTES}{LATENCY(min)} \tag{5}$$

We calculate the throughput of MQTT and CoAP by taking the number of bytes in the payload and dividing it by the latency as described in section 4.3. The median latency of 1000 trials will be taken to calculate throughput. Below are the equations used to convert my data from bytes into megabytes in 6, as well as convert the latency from milliseconds to minutes in 7, which were both used to figure out the throughput with equation 5.

$$MEGABYTES = BYTES/1000000 \tag{6}$$

$$LATENCY(min) = LATENCY(ms)/1000/60 \tag{7}$$

The throughput data is then presented in table 4 using megabytes per minute as the unit. From the results, we can see a reverse linear relationship between MQTT's throughput as packet loss increases. A 94.23% decrease in throughput can be seen between 0% and 10% packet loss. Alternatively, the throughput of CoAP remains stable, reaching 6.25 MB/minute at 8%. These results suggest that CoAP is more suitable in unstable networks with high packet loss percentage.

Table 4. Throughput comparison between MQTT and CoAP

Packet Loss	0%	2%	4%	6%	8%	10%
MQTT (MB/min)	6.07	5.73	2.81	0.59	0.54	0.35
CoAP (MB/min)	6.00	6.03	6.03	6.15	6.25	6.09

Similarly, cryptographic communication requires additional bytes for cryptography-related information, such as public keys or certificates. Additionally, a lot more bytes need to be exchanged to achieve a certain state when using TLS. To highlight the packet size difference between a TCP and a TLS connection, an experiment analyzing captured packets by wireshark will be held. The goal of the experiment is to analyze the total bytes exchanged between the client and server to achieve a certain state. For example, to achieve the connected state with a TCP connection, 194 bytes are required. To make this experiment fair, the following variables between TCP and TLS connections were controlled: QOS level, 0 re-transmissions, equal payload length of 0 bytes, equal topic length of 4 bytes, and MQTT version 5.0 for both connections. The states: CON, SUB, PUB, and DISC represent "connected", "subscribed", "published", and "disconnected" respectively. States SUB, PUP, and DISC, are all achieved from the CON state, as in MQTT, the client needs to be connected to the server before any other action can be made. Each cell represents the number of bytes that need to be exchanged between client and server to establish that state for a given protocol. The "established state" in this experiment is defined in terms of fully achieving an action within the MQTT protocol including the size of the last acknowledgment, assuming no re-transmissions. This means the very first byte sent from the client to the very last acknowledgment

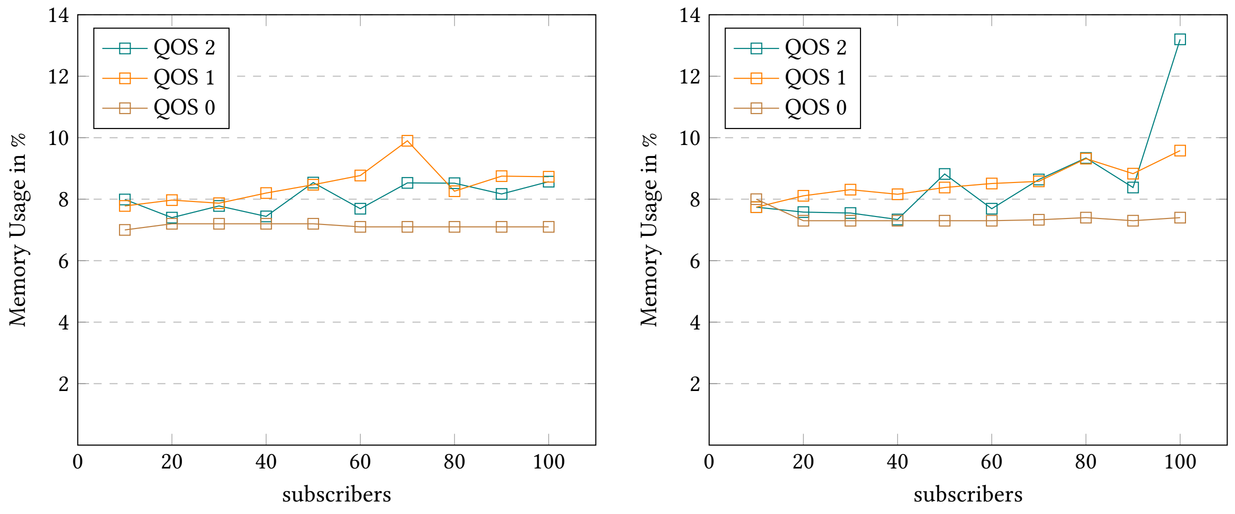

Fig. 3. TCP and TLS memory usage comparison

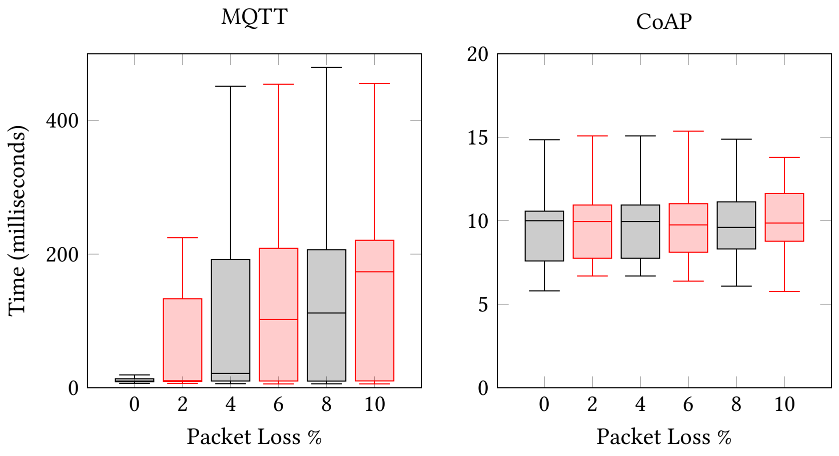

Fig. 4. MQTT and CoAP latency comparison

sent by either the client or the server. The largest difference can be seen for the "CON" state, which takes 3382 bytes to establish the TLS connection. This is 17.43 times larger than that of the TCP connection. This is an issue for IoT projects, which are subject to low bandwidth.

6 DISCUSSION

The research in [6], established a novel denial of service attack towards the MQTT protocol, called 'SlowITe'. This attack made it impossible to establish new legitimate connections with the broker, which compromised the availability of the server. With such a crucial vulnerability it is valuable to test it out on several versions of MQTT and find ways to mitigate this attack. When attempting to re-create the SlowITe attack, the broker offered 'unlimited' connections allowed by default. As the SlowITe attack relied on abusing the limits on several connections set by the broker, this attack was not possible on the EMQX broker. However, allowing 'unlimited' connections introduces other problems, such as the loss of control of server resources. From the graph 2, the impact of additional subscribers to a topic can be seen. This suggests that either the broker would be susceptible to a SlowITe attack, or a traditional DoS attack, where one publisher publishes to many subscribers.

In [12], payload throughput was measured and compared between TCP and TLS connections. The study concluded that the time taken to establish a TLS connection was way greater than the TCP connection. This can be explained using results from 3, where it took 17.43 times more bytes to achieve the connected state for TLS connections than the TCP connections. Additionally, TCP connections achieved a greater payload throughput than TLS connections and achieved a greater energy efficiency [12]. A similar result can be seen by comparing CPU usage in 2, where TLS connections use significantly more CPU than TCP connections.

performance was compared between MQTT and CoAP The experiment for research question 3 compared the performance between MQTT and CoAP with varying packet loss. The Authors had done the same experiment, however, the results of the experiment were different. The results in figure 4 from the experiment suggest that CoAP is more performant in terms of latency when the network has packet loss. However, the results in [13] show that the mean latency of MQTT packets was lower than those of CoAP packets.

This paper answers all of the three research questions. A thorough security analysis was conducted in 5. A performance analysis was conducted, where stress tests were conducted on an MQTT broker running on the RPi 4. The results in 2 show the CPU usage when using TCP and TLS respectively. Additionally, a comparison of memory usage between TCP and TLS can be seen in 3. The last research question of comparing latency and throughput between MQTT and CoAP was answered with an analysis of the gathered data in figure 4 and table 4.

7 CONCLUSION

This research investigated the security and performance aspects of the MQTT protocol in the Internet of Things (IoT) context. Our analysis revealed that MQTT offers lightweight and efficient messaging transport, making it suitable for resource-constrained IoT devices. However, MQTT inherently lacks built-in security features. This exposes data transmissions to eavesdropping, tampering, and unauthorized access. The research explored using Transport Layer Security (TLS) alongside MQTT. TLS implementation significantly enhances the security of MQTT communication by encrypting data and authenticating connections. However, TLS introduces some performance overhead. Therefore, the trade-off between security and performance should be managed based on the specific needs of the IoT application. Future research directions could explore lightweight security mechanisms specifically designed for resource-constrained devices used in conjunction with MQTT.

REFERENCES

[1] 2014. *MQTT Version 3.1.1.* https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[2] Shaun Behrens. 2019. *From Oil Pipelines to the IoT: A Brief History of MQTT.* https://blog.paessler.com/a-brief-history-of-mqtt

[3] Ian Craggs. 2022. *Understanding the Differences Between MQTT and HTTP.* https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/

[4] Laurenz Dallinger. 2023. Mastering Mqtt Packet: A usage example guide. (2023). https://cedalo.com/blog/mqtt-packet-guide/

[5] Dmitrii Dikii, Sergey Arustamov, and Aleksey Grishentsev. 2021. DoS attacks detection in MQTT networks. *Indonesian Journal of Electrical Engineering and Computer Science* 21 (01 2021), 601. https://doi.org/10.11591/ijeecs.v21.i1.pp601-608

[6] Enrico Cambiaso Ivan Vaccari, Maurizio Aiello. 2020. SlowITe, a Novel Denial of Service Attack Affecting MQTT. (May 2020). https://doi.org/10.3390/s20102932

[7] Daniel Kant, Andreas Johannsen, and Reiner Creutzburg. 2021. Analysis of IoT Security Risks based on the exposure of the MQTT Protocol. *Electronic Imaging* 33, 3 (2021), 96–1–96–1. https://doi.org/10.2352/ISSN.2470-1173.2021.3.MOBMU-096

[8] Puneet Kumar and Behnam Dezfouli. 2019. Implementation and analysis of QUIC for MQTT. *Computer Networks* 150 (2019), 28–45. https://doi.org/10.1016/j.comnet.2018.12.012

[9] Diana Bezerra Correia Lima, Rubens Matheus Brasil da Silva Lima, Douglas de Farias Medeiros, Renata Imaculada Soares Pereira, Cleonilson Protasio de Souza, and Orlando Baiocchi. 2019. A Performance Evaluation of Raspberry Pi Zero W Based Gateway Running MQTT Broker for IoT. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON).* 0076–0081. https://doi.org/10.1109/IEMCON.2019.8936206

[10] Maroun Chamoun Mohammed El-Hajj, Ahmad Fadlallah and Ahmed Serhrouchni. 2019. A Survey of Internet of Things (IoT) Authentication Schemes. (2019). https://doi.org/10.3390/s19051141

[11] Avram Piltch. 2020. . https://www.tomshardware.com/reviews/raspberry-pi-4

[12] Thomas Prantl, Lukas Iffländer, Stefan Herrnleben, Simon Engel, Samuel Kounev, and Christian Krupitzer. 2021. Performance Impact Analysis of Securing MQTT Using TLS. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Virtual Event, France) *(ICPE '21).* Association for Computing Machinery, New York, NY, USA, 241–248. https://doi.org/10.1145/3427921.3450253

[13] Victor Seoane, Carlos Garcia-Rubio, Florina Almenares, and Celeste Campo. 2021. Performance evaluation of CoAP and MQTT with security support for IoT environments. *Computer Networks* 197 (2021), 108338. https://doi.org/10.1016/j.comnet.2021.108338