WhisperMQTT: Lightweight Secure Communication Scheme for Subscription-Heavy MQTT Network

Youbin Kim and Man-Ki Yoon
Department of Computer Science
North Carolina State University
Raleigh, NC, USA

***Abstract*—The Message Queuing Telemetry Transport (MQTT) protocol is widely used for communication in machine-to-machine (M2M) and Internet of Things (IoT) systems due to its lightweight publish-subscribe model. Although MQTT can be secured with Transport Layer Security (TLS), this introduces significant computational overhead on the message broker. We demonstrate that this overhead becomes unnecessarily high in subscription-heavy networks, where a single message must be encrypted individually for each subscriber. To address this issue, we present WHISPERMQTT, a solution designed to optimize the encryption process in MQTT brokers. Our approach utilizes both TLS and non-TLS connections with subscribers: the TLS channel is used to securely distribute a per-topic decryption key, while the encrypted payload is transmitted over the non-TLS connection. This allows the broker to encrypt each topic message only once, irrespective of the number of subscribers, which reduces processing overhead and enhances both latency and throughput. Importantly, WHISPERMQTT integrates seamlessly with the MQTT protocol and applications without requiring any modifications.**

I. INTRODUCTION

Message Queuing Telemetry Transport (MQTT) [1] is a lightweight protocol for machine-to-machine (M2M) communication and Internet of Things (IoT). To support efficient communication, MQTT uses the publish-subscribe (pub-sub) model, which enhances both scalability and reliability. In this model, devices (clients) interact through a central *broker* that acts as a mediator, routing data transmissions based on specific *topics* of interest for clients. This topic-based approach enables the broker to distribute messages from the publisher to subscribers. Using this decoupled model allows devices to operate asynchronously and independently without needing to be aware of each other.

MQTT operates over the Transmission Control Protocol (TCP) transport by default, but TCP alone does not provide inherent security. To enhance security, both the broker and MQTT clients can use Transport Layer Security (TLS) [2], which is supported by most mainstream broker libraries [3]–[6]. TLS is a secure networking protocol that ensures data confidentiality, integrity, and authentication over networks. As a result, it has become the dominant secure networking protocol for a wide range of applications, including web services [7], messaging [1], media streaming [8], and cloud storage [9]. When integrated with MQTT, TLS establishes a secure connection between the broker and its clients. This approach is commonly used because TLS provides a variety of encryption algorithms [10], key exchange methods, and authentication mechanisms that are suitable for secure communication across different network environments.

Although TLS provides security benefits, it introduces significant computational overhead for the broker, as it must decrypt messages from publishers and encrypt messages for subscribers. The decryption process is a fixed-cost operation with limited optimization opportunities (aside from not using TLS), as each topic is published by a single publisher in a single message. However, encrypting messages for subscribers can be more costly for the broker because MQTT typically follows a *one-to-many* communication pattern: one publisher and multiple subscribers. This overhead becomes especially significant in *subscription-heavy* networks, where a large number of subscribers are associated with a single topic. In such cases, the broker must encrypt the *same* message *individually* for each subscriber.

Drawing on the insights from the above observations, we propose WHISPERMQTT, a solution that eliminates redundancy in the encryption operations for subscribers. It establishes both TLS and non-TLS connections between the broker and subscribers simultaneously. The TLS channel is used to distribute a per-topic encryption key to the subscribers, ensuring the same level of security as traditional TLS-based MQTT, while the actual encrypted data is transmitted through the non-TLS channel. Hence, the broker performs encryption only once per message, regardless of the number of subscribers. By encrypting just once, WHISPERMQTT significantly reduces the broker's computational overhead, leading to lower latency and higher throughput. Most importantly, WHISPERMQTT does not require any modification to the MQTT protocol and applications. We implement WHISPERMQTT in the popular Mosquitto MQTT broker [3] and the Paho MQTT client library [11]. We evaluate the performance of WHISPERMQTT through a set of experiments and demonstrate that it can reduce the latency overhead of the broker's publication process by 10% to 50% compared to standard TLS-based MQTT communication depending on the payload sizes and the number of subscribers.

In summary, we make the following contributions in this paper:

- We show that the MQTT communication in subscription-heavy networks can lead to significant computational

overhead on the broker when using standard TLS-based message distribution;
- We propose WHISPERMQTT, a novel approach that combines the benefits of TLS with non-TLS connections to optimize the encryption process in MQTT brokers for subscription-heavy networks; and
- We assess both the performance benefits and overhead of WHISPERMQTT compared to non-secure and standard TLS-based MQTT communication approaches by implementing it in open-source MQTT broker and client libraries.

II. PROBLEM DESCRIPTION

A. System and Threat Models

We consider a network of nodes that communicate using the MQTT protocol. We assume that nodes can join and leave the network at arbitrary times. A node can *publish* messages to one or more topics. Similarly, a node can *subscribe* to one or more topics it is interested in. In the MQTT protocol, a central broker is responsible for routing messages between publishers and subscribers. While this paper focuses on a single broker for simplicity, the proposed solution can be extended to support bridged brokers.

We also assume that (i) each node, including the broker, can generate a public and private key pair, (ii) the keys are accompanied by a verifiable certificate, and (iii) a standard security mechanism is in place to protect the private key in each node. The public-private key pairs and certificates are used to authenticate nodes to each other during the TLS handshake.

Let $e_{key}(\cdot)$ and $d_{key}(\cdot)$ denote the encryption and decryption functions, respectively, using a symmetric key key. Note that TLS uses symmetric key encryption to ensure the confidentiality and integrity of data transmitted between a client and a server during a session. Given a symmetric key used to encrypt an MQTT payload, anyone can decrypt it. We do not consider such a scenario where the symmetric key is exposed to nodes that are not subscribed to the topic of the encrypted data, as it represents an obvious and fundamental security breach. Hence, a node cannot decrypt an MQTT payload without subscribing to the topic through the broker. We also require a key revocation mechanism to prevent former subscribers from decrypting encrypted MQTT payload after their departure.

B. Motivational Scenario

Consider a camera positioned along a road that captures real-time footage of traffic for various purposes. Instead of operating a separate camera for each application, the system employs a pub-sub model like MQTT to share the real-time footage among multiple applications. In this scenario, the camera acts as a publisher, while subscribers include various vision-based applications. For example, the system could allow traffic control centers to monitor and manage traffic signals and flow, respond to accidents, detect speeding and traffic signal violations, and more. It could also enable law enforcement to identify and track suspects, detect suspicious activities, and conduct surveillance. However, the sensitive nature of these data necessitates encryption to ensure confidentiality and integrity, which poses a challenge due to the often large size of such data.

C. Problem Statement

While TLS ensures data confidentiality and integrity, it comes at the expense of additional computational load due to the handshake process, message encryption/decryption, and computation of the message authentication code. As a result, using TLS can lead to increased latency compared to unencrypted communication. From the broker's perspective, the major overhead lies in encrypting the *same* MQTT payload with *different* TLS sessions keys for each subscriber of the same topic. To illustrate the impact of this overhead on the broker, we consider a scenario where a large payload is published to multiple subscribers and measure the time taken by the broker to handle a single publication, starting from the moment the data is received from the publisher until it is sent out to all subscribers. The experimental setup is detailed in Section IV-A.

Fig. 1 compares the average latency incurred by the broker when sending 3 MB of data to all subscribers using TLS versus non-TLS. The x-axis represents the number of subscribers, while the y-axis shows the average latency in milliseconds (ms). As depicted, the results show a significant increase in the broker's overhead with TLS compared to non-TLS, with the overhead growing more pronounced as the number of subscribers to the same topic increases; e.g., the average time for 10 subscribers with TLS is 33.29 ms, more than twice the 12.87 ms observed with non-TLS.

Building on the above observation, we aim to design a communication scheme that ensures the broker's overhead for publishing to multiple subscribers of the same topic scales moderately with the number of subscribers, as seen with non-TLS, while preserving data confidentiality, integrity, and authenticity equivalent to TLS.

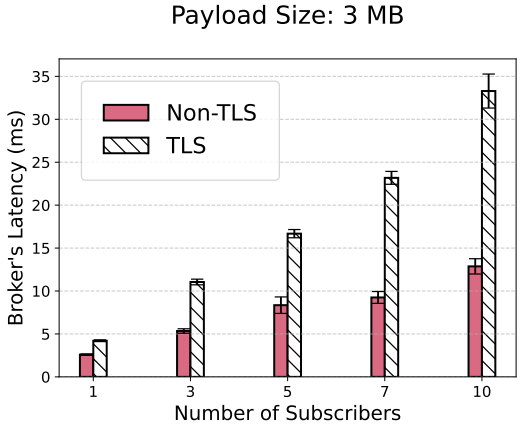
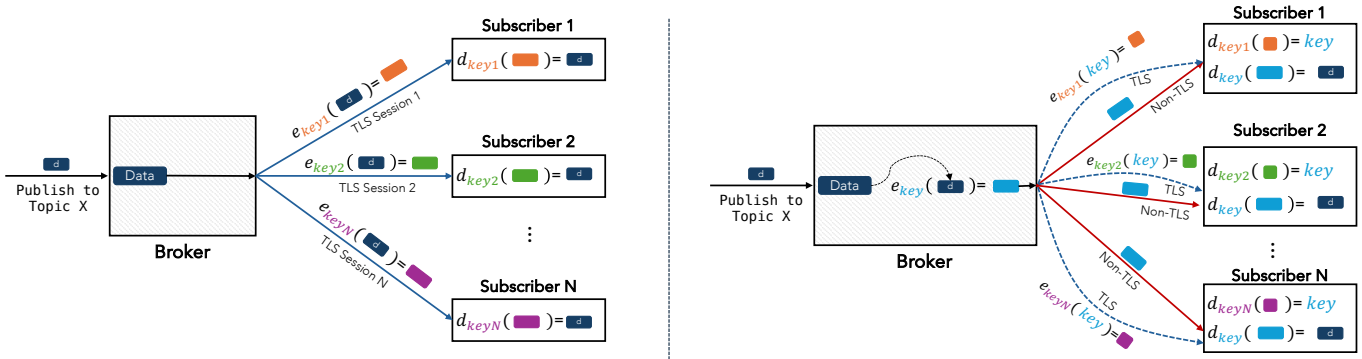


Fig. 1: The average time taken by the broker to send out 3 MB of data to all subscribers.



(a) Standard TLS-based Publication

(b) Proposed Solution (TLS for key, non-TLS for encrypted data)

Fig. 2: Secure MQTT communication using (a) TLS to encrypt messages for each channel, and (b) our proposed solution, WHISPERMQTT, that encrypts the message only once using a single key and distributes the key over TLS channels.

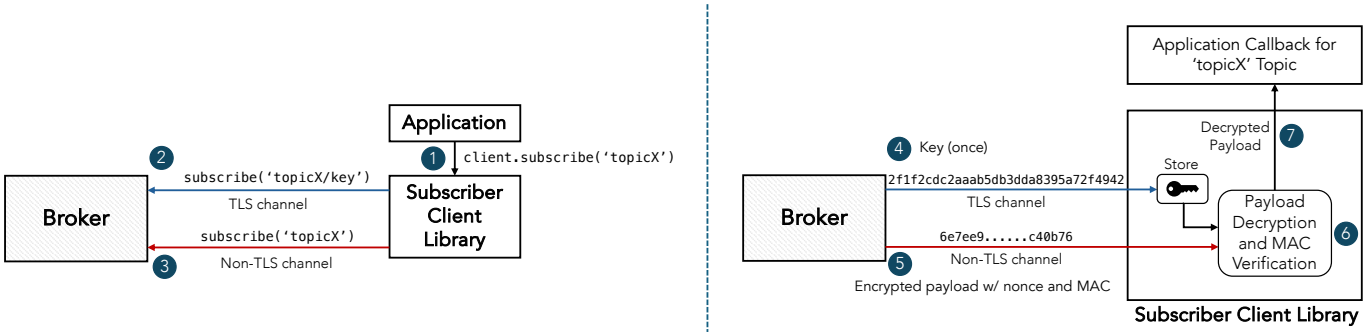


Fig. 3: **(Left)** Subscriber creates a subscription to the key topic and the data topic. **(Right)** Subscriber decrypts the encrypted payload using the decryption key that was sent over the TLS channel.

III. WHISPERMQTT DESIGN AND IMPLEMENTATION

A. High-level Idea

Fig. 2 illustrates the high-level idea of WHISPERMQTT, as shown in (b), and how it contrasts with the standard TLS-based subscription depicted in (a). As briefly discussed in Section I, the overhead imposed by TLS on the MQTT broker arises from encrypting the same message separately for each subscriber. Specifically, each subscriber connects through a different TLS session, and the message transmitted over each session is encrypted individually using a session-specific key. Consequently, the same message is encrypted N times for the N subscribers. As the number of subscribers grows, the need to repeatedly encrypt the same data leads to substantial overhead for the broker.

To address this problem, WHISPERMQTT removes the redundancy by shifting the encryption process to the broker layer, as illustrated in Fig. 2(b). Since the broker knows that all subscribers of a given topic should receive the same message, it can encrypt the message *once* with a single key rather than using separate keys for each subscriber. This single key is then distributed to the subscribers through their individual TLS channels (i.e., the dashed blue lines in (b)). The overhead of key distribution is minimal, as the key is typically very small (e.g., 32 bytes) and requires infrequent updates. Thus, the encryption overhead for publishing a message to subscribers becomes independent of the number of subscribers; only the total transmission time increases with the number of subscribers, which is unavoidable.

B. Protocol Detail

In what follows, we describe the WHISPERMQTT protocol in detail for each network participant and explain how a subscription-heavy MQTT network can benefit from this protocol.

1) Publisher: No modifications are required for publishers; they can publish messages to the broker as usual. This is due to the decoupling between publishers and subscribers in the pub-sub communication model.

2) Subscriber: As explained earlier, the key idea of WHISPERMQTT is to distribute the decryption key via a TLS channel and the encrypted data via a non-TLS channel. This approach allows the broker to encrypt the data *only once*, no matter how many subscribers are subscribed to the topic, and distribute the encrypted data to all subscribers. For this,

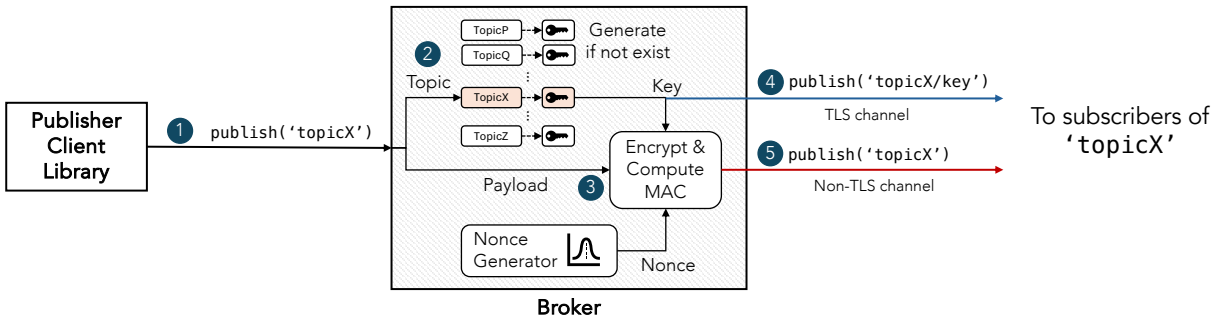


Fig. 4: WHISPERMQTT broker's operations. For each incoming message, the broker encrypts it using a per-topic key and sends the encrypted data over non-TLS channels, while the key is distributed (when created or updated) over TLS channels.

the subscriber subscribes to the *key topic* and the *data topic*. Fig. 3 (left) illustrates the subscription process for a subscriber. The subscriber application is not aware of the dual channel communication; it simply requests the client library to subscribe to a topic, e.g., `'topicX'`. Then the client library sends *two* subscription requests to the broker: one for the key topic, `'topicX/key'`, and the other for the data itself, `'topicX'`.

Then, as shown on the right-hand side of Fig. 3, the broker sends the decryption key to the subscriber via the key topic, which is encrypted using the TLS session key (this process is transparent to both the broker and the client library). The key is not sent again for the same topic unless it needs to be revoked and regenerated (e.g., when one of the subscribers for the topic leaves the network). The subscriber's client library stores the key internally and uses it to decrypt all future encrypted payloads received from the broker. For each encrypted data received over the non-TLS channel, the subscriber's client library verifies the data's integrity using the message authentication code (MAC) sent along with the encrypted payload and then decrypts it using the stored key. Upon successful verification and decryption, the decrypted payload is delivered to the subscriber application's callback function as if it were a regular MQTT message.

3) Broker: Fig. 4 and Algorithm 1 illustrate the operations of the WHISPERMQTT broker:

1) A publisher sends a message to the broker.
2) Lines 2–7: The broker first looks up the topic to find the corresponding encryption key. If the key does not exist (i.e., the topic is new), the broker generates a new key for the topic, distributes it to all subscribers over TLS channels (`TLSPublish`), and stores it in a key-value store.
3) Lines 9–10: Whether the key is pre-existing or newly generated, the broker encrypts the original payload using the key and a random nonce (e.g., an Initialization Vector in the case of AES-GCM) and computes the message authentication code.
4) Lines 11–12: The broker then replaces the original payload with the encrypted payload, nonce, and message authentication code, as shown in Fig. 5. Finally, the broker sends this data to all subscribers of the topic over non-TLS channels (`NonTLSPublish`).

Algorithm 1 `WhisperPublish` $(topic, payload)$ (See Fig. 4 and Fig. 5

1: **Input:** Topic and unencrypted payload
2: $key \leftarrow$ `FindKey`$(topic)$ {See Algorithm 2}
3: **if** key is NULL **then**
4: {Key does not exist → generate and publish one}
5: $key \leftarrow$ `RAND(KEY_SIZE)`
6: Insert the $topic \rightarrow key$ mapping to hash table
7: `TLSPublish`(topic+'/key', key)
8: **end if**
9: $nonce \leftarrow$ `RAND(NONCE_SIZE)`
10: $(encData, MAC) \leftarrow$ `Encrypt`$(key, payload, nonce)$
11: $newPayload \leftarrow encData|nonce|MAC$
12: `NonTLSPublish`$(topic, newPayload)$

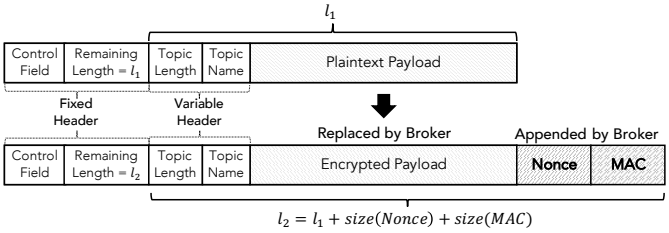


Fig. 5: WHISPERMQTT broker's payload encryption.

Note that the topic key is distributed (i.e., published) to all subscribers only when it is (re)generated. Hence, the broker retains the key for future subscribers who join the network. This can be done by setting the `RETAIN` bit [12] of the key topic message to true. Conversely, when an existing subscriber leaves the network or simply unsubscribes, the broker revokes the key and generates a new one for the topic, as shown in Fig. 6. The new key is then distributed to all remaining subscribers of the topic over their TLS channels. This process ensures that former subscribers cannot decrypt the encrypted data after unsubscribing, even if they gain access to it after

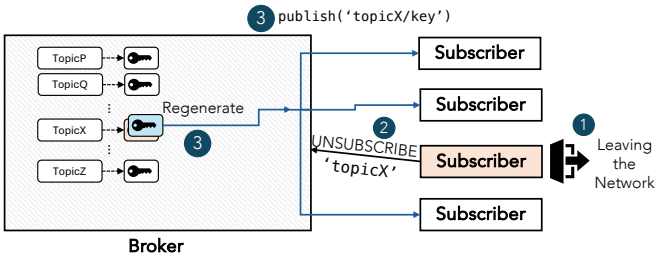


Fig. 6: Decryption key is regenerated and distributed to subscribers when a subscriber leaves the network or unsubscribes.

leaving the network. Note that, although small, there are overheads involved in key generation and distribution. Hence, a malicious node may attempt to repeatedly join and leave the network to launch a denial-of-service attack against the broker. To mitigate this, the broker can impose a rate limit on the number of key updates per topic or blacklist the node if it exhibits such behavior.

C. Implementation-Specific Details

Cryptographic Algorithm. For the implementation of WHISPERMQTT, we use AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) [13] with a 256-bit key (which is the highest key length supported by AES) for the cryptographic algorithm. The choice is based on the fact that `AES_256_GCM_SHA384` is one of the default cipher suites in the TLS protocol [14]. Hence, for a fair comparison with the TLS-based MQTT communication, we use the `AES_256_GCM_SHA384` cipher suite. AES significantly enhances encryption and decryption performances due to its wide support across modern processor architectures that provide hardware-accelerated AES instructions. On x86 processors, AES-NI (Advanced Encryption Standard New Instructions) [15] provides hardware-accelerated encryption and decryption through instructions. Similarly, ARM processors integrate AES support via the ARMv8 Cryptography Extensions [16]. These hardware-based optimizations enable faster and more efficient AES encryption/decryption operations.

AES is a symmetric key encryption algorithm, and hence the same key is used for both encryption and decryption. In WHISPERMQTT protocol, the broker generates a new key for each topic when a new topic is published. The GCM generates an authentication tag (or MAC) during encryption, which is appended to the encrypted payload to ensure both data integrity and authentication, as shown in Fig. 5. This tag ensures that the cipher text has not been tampered with. As a result, an attacker cannot alter data transmitted by the broker over a non-TLS channel, as they cannot generate the correct tag without the key. If the attacker attempts to modify the encrypted payload, the subscriber will detect this manipulation during decryption.

Also included is a unique nonce (called an Initialization Vector, IV) that is required for each encryption operation to ensure security. The nonce is generated for each new message randomly by the broker and is sent along with the encrypted payload and the tag to the subscribers. The nonce is used to ensure that the same plain text data encrypted with the same key does not produce the same cipher text. This is important for security as it prevents attackers from deducing the key by observing the cipher texts. In our implementation, the nonce size is 96 bits, and the tag size is 128 bits.

Algorithm 2 `FindKey` $(topic)$

Input: Topic name
Output: Encryption/Decryption key for the topic

$index \leftarrow \texttt{hash}(topic)$
$current \leftarrow hahstable[index]$
while $current! =$ NULL **do**
  {Iterate through chain of topic-to-key map}
  if $current.topic == topic$ **then**
    return $current.key$
  else
    $current \leftarrow current.next$
  end if
end while
{Key does not exist if reaching here}
return NULL

Topic to Key Mapping. As explained in the previous section, the broker needs to store the encryption/decryption key (i.e., 256-bit AES key) for each topic. To achieve this, we use a hash map data structure to map topics to their corresponding keys. Specifically, given a new `PUBLISH` packet from the publisher, the broker first extracts the topic name from its variable header. Then, it applies a simple hash algorithm to compute the hash value of the topic name, which is then used as the index in the hash map. Because multiple topic names can have the same hash value, our implementation resolves conflicts using separate chaining for the values (i.e., the keys). Algorithm 2 summarizes the key lookup operation.

Memory Optimization. The payload encryption process shown in Fig. 5 involves allocating memory for the cipher text and copying it to a new payload buffer. In our implementation, we avoid allocating a new memory buffer for the cipher text. Instead, we try to encrypt the payload *in-place* by using `realloc` in the C stdlib library; this function tries to resize the memory block pointed to by the original payload buffer. If the buffer is large enough to accommodate the additional fields, i.e., nonce and tag, the encryption is performed in-place. Otherwise, a new buffer is allocated. This optimization reduces the overhead associated with new memory allocation, copying, and deallocation and improves the overall latency of the broker.

D. Security Analysis

As explained above, in the WHISPERMQTT protocol, the encryption/decryption key for a topic, such as

'topicX', is published to a corresponding key topic, e.g., 'topicX/key'. The security of the key is ensured by the TLS layer, which provides end-to-end encryption between the broker and each of the subscribers. During the TLS handshake, the broker authenticates the subscribers before establishing the session for the key topic. If a subscriber fails authentication, the key topic is not published to it. Without the key, the encrypted payload of the data topic is meaningless to the subscriber. Hence, WHISPERMQTT offers the same level of security as standard TLS-based MQTT communication.

A concern may arise from the fact that any client node could subscribe to the key topic. Suppose an illegitimate node subscribes to the key topic 'topicX/key'. The node's subscription to the key topic is not a security concern, because the node would need to have a legitimate subscription to the (encrypted) data topic 'topicX'. That is, if the node was a legitimate subscriber, it already has access to the key topic. If the node was *not* a legitimate subscriber, it would not have access to the encrypted data topic, and hence the key would be useless. Therefore, the security of the key is not compromised by the public nature of the key topic.

Then, what if a legitimate subscriber leaks the key 'topicX/key' to an unauthorized client? This concern is not specific to WHISPERMQTT. A legitimate subscriber already has access to plain text data, which is decrypted from the cipher text 'topicX' using the key provided by the broker. As a result, even in standard TLS-based MQTT communication, they could potentially leak the decrypted data to an unauthorized client. Therefore, such malicious behavior is not directly tied to the security of the key itself. Nevertheless, one could revoke and regenerate the key at random intervals to further enhance security.

IV. EVALUATION

A. Setup

We implemented WHISPERMQTT in the Eclipse Mosquitto broker (version 2.0.18) [3] and the Eclipse Paho MQTT Python Client (version 2.1.0) [11]. For the TLS and AES-GCM encryption/decryption, we used OpenSSL version 3.0.13 [14]. Our experiments were conducted on the Google Cloud Platform [9]. We used several `e2-standard-16` virtual machines (VMs), which are based on the x86/64 architecture and run Debian GNU/Linux 12. Each VM is equipped with 16 virtual CPUs and 64 GB of memory. This setup was selected to ensure that the VMs have adequate computational resources to meet the demands of both the broker and the clients. We varied the number of subscribers – 1, 3, 5, 7, and 10 – distributing them evenly across up to four VMs (e.g., 3, 3, 2, and 2 for the case of 10 subscribers).

B. Results

In what follows, we evaluate both the performance benefits and overhead of WHISPERMQTT compared to unencrypted and standard TLS-based MQTT communication approaches. We focus on the following metrics: latency, protocol overhead, and operational costs.

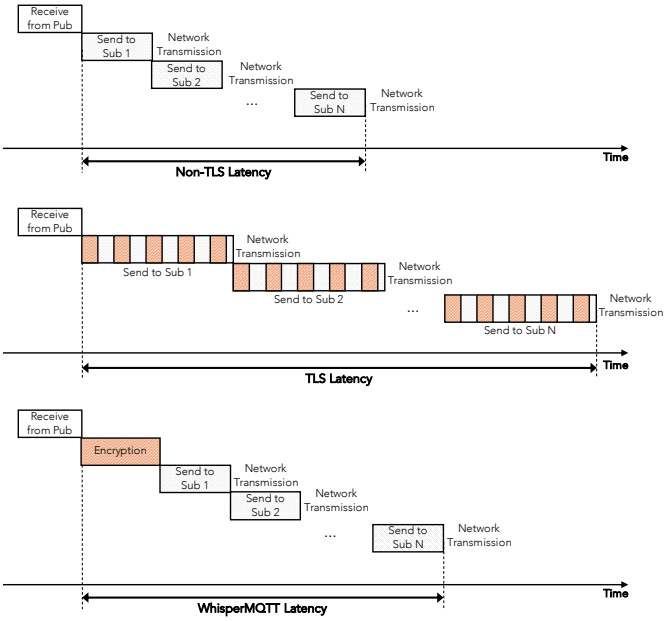


Fig. 7: Timeline of the broker's publication to subscribers when using non-TLS (top), TLS (middle), and WHISPERMQTT (bottom). The latency is measured from the moment the broker receives a publication from the publisher until it is sent out to all subscribers. The latencies do not include network transmission times.

1) Broker latency: We first evaluate the latency in the broker when publishing to multiple subscribers. We measure the latency from the moment an MQTT packet is identified as a `PUBLISH` packet until it is sent out to all subscribers of the topic. Fig. 7 illustrates the timeline of the publication from the broker to the subscribers when using non-TLS (top), TLS (middle), and WHISPERMQTT (bottom). Note that the time taken for the messages to travel from the broker to the subscribers, i.e., transmission time, is not included in the latency measurement.

Fig. 8 shows the average latency for different payload sizes (10 KB, 100 KB, 1 MB, and 3 MB) and numbers of subscribers (1, 3, 5, 7, and 10). The x-axis represents the number of subscribers, while the y-axis represents the average latency in milliseconds that are based on 500 samples. Note the difference in the y-axis scales across the plots. The results highlight the following: (i) when the data size remains constant, the average latency increases as the number of subscribers grows no matter the communication method; (ii) the non-TLS case shows the lowest latency and the moderate growth rate, which is expected as it does not involve encryption; (iii) the TLS-based publication shows the highest latency and the steepest growth; (iv) WHISPERMQTT demonstrates latency growth comparable to non-TLS communication, while still supporting encrypted data transmission, as TLS does.

With a payload size of 1 MB being published to 10 subscribers, the latency increase for WHISPERMQTT compared to non-TLS is about 28.74%, while TLS shows an increase

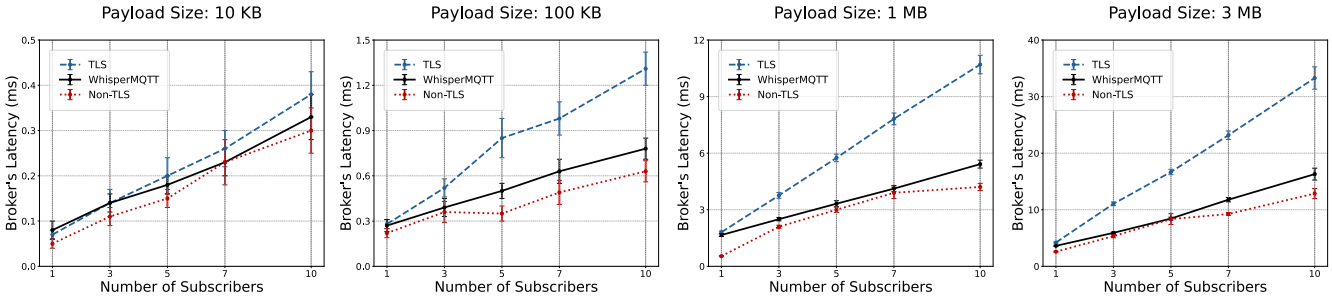


Fig. 8: Broker's latency to publish different payload sizes to multiple subscribers.

TABLE I: Average latency (ms) and percentage increase due to TLS and WHISPERMQTT compared to non-TLS for different payload sizes when publishing to 10 subscribers.

Payload Size	**10 KB**	**100 KB**	**1 MB**	**3 MB**
(A) Non-TLS Latency	0.30	0.63	4.21	12.87
(B) TLS Latency	0.38	1.31	10.70	33.29
(C) WHISPER Latency	0.33	0.78	5.42	16.30
((B-A)/A) TLS Overhead	26.67%	107.94%	154.16%	158.67%
((C-A)/A) WHISPER Overhead	10.00%	23.81%	28.74%	26.66%
((B-C)/B) Latency Reduction	13.16%	40.46%	49.35%	51.04%

of approximately 154.16% relative to non-TLS, as shown in Table I. This trend is also observed with other payload sizes; the percentage increase for WHISPERMQTT compared to non-TLS are approximately 10.00% for 10 KB, 23.81% for 100 KB, and 26.66% for 3 MB, respectively. In contrast, TLS shows percentage growth of about 26.67% for 10 KB, 107.94% for 100 KB, and 158.67% for 3 MB. As can be seen, the overhead of TLS increases rapidly with the payload size and the number of subscribers. The last row in Table I further indicates that WHISPERMQTT reduces the broker's latency by over 50% compared to TLS for payloads of 3 MB.

To demonstrate that WHISPERMQTT operates *independently* of specific processor architectures and is compatible with various environments, we conducted an additional experiment. For this purpose, we configured a virtual machine for the broker on an ARM processor-based instance (Ampere Altra processor [17], `t2a-standard-16`), maintaining the same conditions as in the previous experiments. Fig. 9 show similar trends to those observed in the previous experiments; WHISPERMQTT aligns with the growth rate of the non-TLS-based publication shown in Fig. 8. This result shows WHISPERMQTT's ability to maintain consistent performance across diverse processor architectures and environments.

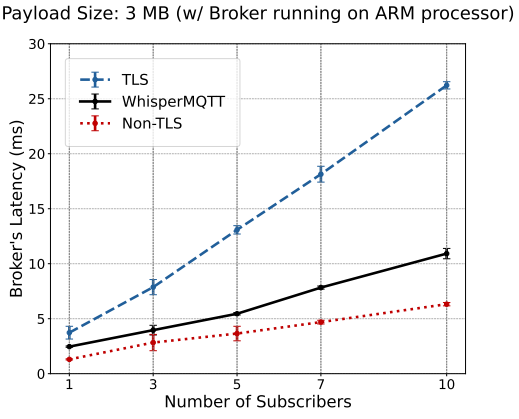


Fig. 9: ARM processor-based broker's latency to publish a 3 MB payload.

The results above highlight that WHISPERMQTT effectively reduces latency overhead while ensuring secure communication comparable to the TLS-based publication.

2) Protocol overhead: Encryption introduces additional metadata, which leads to overhead in data transmission. Therefore, we evaluate the overhead of WHISPERMQTT by comparing the message size with the TLS-based communication. Note that MQTT packets and TLS records (which will be discussed later) are encapsulated in TCP packets. Hence, we focus on the payloads transmitted by TCP packets, excluding the TCP and IP headers.

An MQTT packet consists of a fixed header, a variable header, and a payload [12] as shown in Fig. 5. The fixed header contains the control field (1 byte) and the remaining length field, which is 2–4 bytes long, depending on the payload size. The variable header includes the topic name length (2 bytes) and the topic name itself (4 bytes). Hence, as shown in the row labeled (A) in Table II, the MQTT header size ranges between 9 and 11 bytes. For a payload of 10,000 bytes, the total MQTT packet size is 10,009 bytes, and for a payload of 3,000,000 bytes, it is 3,000,011 bytes, as shown in Row (B).[1] These are the sizes of the messages that the broker sends to each subscriber over a non-TLS connection. Note, however, that we do not consider the MQTT packet header as overhead, since it remains constant across all communication methods. Instead, we treat it as part of the payload that the underlying layer must transmit.

When using TLS, the data is encapsulated in *TLS records* [2] as shown in Fig. 10. A TLS record is the basic unit of the

[1]Note that we use the message sizes that are multiples of 1,000 bytes, not 1,024 bytes, for this experiment for simplicity in the overhead analysis.

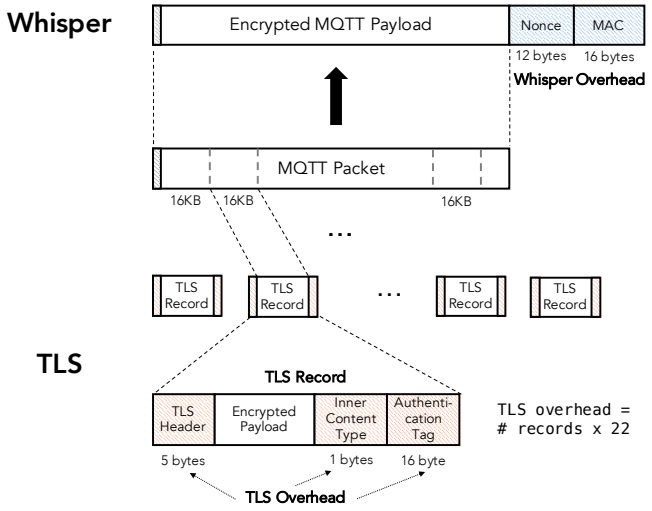


Fig. 10: Protocol overhead of WHISPERMQTT (top) and TLS (bottom). WHISPERMQTT adds a single pair of nonce and MAC to the payload, regardless of its size, while TLS introduces overhead that increases with the payload size.

TABLE II: Protocol overhead comparison between TLS and WHISPERMQTT for different MQTT payload sizes.

Payload Size (bytes)	**10,000**	**100,000**	**1,000,000**	**3,000,000**
(A) MQTT header size	9	10	10	11
(B) MQTT packet size	10,009	100,010	1,000,010	3,000,011
(C) # TLS records	1	7	62	184
(D=C×22) TLS overhead	22	154	1,364	4,048
(E) WHISPER overhead	28	28	28	28
(D/E) TLS / WHISPER %	79%	550%	4,871%	14,457%

communication in the TLS protocol. A TLS record consists of a header and a payload. The TLS header is a fixed-length structure composed of content type (1 byte), version (2 bytes), and the length of the encrypted payload (2 bytes). When sending data over TLS, we need to account for the overhead introduced by the TLS record structure, including the header and the authentication tag (which is generated by an Authenticated Encryption with Associated Data (AEAD) cipher, like AES-GCM). Note that there is no padding because AES-GCM is a block cipher mode that does not require padding.

The maximum size of the encrypted payload (application data) in a single TLS record is typically 16 KB (16,384 bytes). A data size of 10,000 bytes (plus the MQTT header, which is 9 bytes) can be sent in a single TLS record. Hence, the additional overhead is (see Fig. 10):

- 5 bytes for the (single) TLS record header,
- 1 byte for the inner content type [2], and
- 16 bytes (=128 bits) for the authentication tag.

Therefore, the total overhead per record is 22 bytes.

For large data sizes, the data is divided into *multiple* TLS records, each of which is encrypted *individually*. Assuming each TLS record is filled to its maximum capacity (thus minimizing TLS overhead), an MQTT payload of 100,000 bytes (plus the 10-byte MQTT header) is divided into 7 TLS records because $\lceil 100,010/16,384 \rceil = 7$. In our experiments, we captured 100,164 bytes of TCP payload: six full TLS records, each consuming 16,384 + 22 bytes, and a partial record consuming 1,706 + 22 bytes. Hence, the total TLS overhead for sending 100,010 bytes of an MQTT packet is 154 bytes. Similarly, an MQTT packet of 3,000,011 bytes was divided into 184 TLS records, as $\lceil 3,000,011/16,384 \rceil = 184$, resulting in a total overhead of 4,048 bytes. Generalizing this, the overhead introduced by TLS for a given MQTT packet size m can be calculated by $\lceil m/R \rceil \times (17 + 5)$, where R is the maximum record size (e.g., 16 KB). The row labeled (D) in Table II shows the overhead introduced by TLS for different MQTT payload sizes.

While the protocol overhead of TLS increases with data size as analyzed above, the overhead of WHISPERMQTT remains *constant*. This is because its per-topic-based encryption adds only a single pair of metadata, nonce and MAC, to the MQTT packet as part of the payload, regardless of the data size, as depicted in Fig. 10. In contrast, TLS's record-based encryption divides the data into multiple records and adds metadata to each one. As can be seen from the row labeled (E) in Table II, the protocol overhead of WHISPERMQTT is 28 bytes for all data sizes: 96 bits (12 bytes) for the nonce and 128 bits (16 bytes) for the MAC.

WHISPERMQTT also incurs a negligibly small overhead due to key distribution. We use a 256-bit (32-byte) key for AES-GCM encryption, which is distributed to subscribers by the broker whenever a key is generated or updated. A single key update through the TLS channel requires a single TLS record, which incurs an overhead of 54 bytes: 32 bytes for the key itself and 22 bytes for TLS overhead. Assuming the key is updated on average every p publications, the overhead due to the key update is $54/(p \times s)$, where s is the unencrypted payload size. For example, if the key is updated every 100 publications, the overhead is $54/(100 \times 10,000) = 0.0054\%$ for a 10,000-byte message. This overhead is effectively amortized over larger messages and/or less frequent key updates. For instance, when sending a 3,000,000-byte message and updating the key every 1,000 publications, the overhead due to the key distribution becomes 0.0000018%.

The last row in Table II shows the percentage increase in the number of bytes for protocol overhead when using TLS compared to WHISPERMQTT (ignoring the overhead due to key distribution, which is negligible). Only when the MQTT payload is small enough to fit into a single TLS record, the overhead of TLS is less than that of WHISPERMQTT. However, as the payload size increases, the overhead of TLS increases significantly compared to WHISPERMQTT. For example, when sending a 3,000,000-byte message, the overhead increases by 14,457% when using TLS compared to WHISPERMQTT. This highlights the efficiency of WHISPERMQTT

in reducing the protocol overhead compared to TLS-based publication. Note that the benefits of using WHISPERMQTT are more pronounced in a subscription-heavy MQTT network, where the broker needs to send the same data to multiple subscribers.

3) Operational costs: We also assess the WHISPERMQTT-specific operational costs, which include (a) the time taken to encrypt a payload using AES-GCM and (b) the time needed to look up the encryption/decryption key given the topic.

AES-GCM encryption time: We measured the average time taken to encrypt a payload using AES-GCM (with a 256-bit key). For this, we randomly generated 10,000 payloads of sizes 10 KB, 100 KB, 1 MB, and 3 MB.

TABLE III: Average AES-GCM encryption time.

Data Size	10 KB	100 KB	1 MB	3 MB
Average	2.98 us	26.06 us	265.74 us	803.63 us
Std. Deviation	0.45 us	1.48 us	7.78 us	23.11 us

Table III shows the average encryption time for different data sizes. The average encryption time increases linearly with the data size, as expected. Although the encryption time is relatively small, it is important to consider the overhead when encrypting each payload for a large number of subscribers in the standard TLS-based communication. On the other hand, in WHISPERMQTT, the encryption is done only once for each publication, which reduces the burden on the broker significantly; e.g., approximately 0.8 milliseconds for encrypting a 3 MB payload no matter how many subscribers are there.

Key lookup time: As explained in Section III-C, the broker maintains a hash table to map topics to encryption/decryption keys. A poor design and implementation of this hash table could lead to a performance overhead as the lookup operation is performed for each publication. Hence, we measured the average time taken by the broker to look up a key in this hash table (i.e., Algorithm 2). In this experiment, we gradually generated random topic-to-key mappings, starting from an empty table and increasing to 10,000 topics (with lengths randomly chosen between 10 and 20 characters), measured the lookup time for each, and then calculated the average. This process was repeated 100 times for each data point.

Fig. 11 shows the average lookup times for two different hash table sizes (i.e., the number of slots). For the hash table with 100 slots, the lookup time increases gradually as the number of mappings reaches 1,000. Beyond this point, the lookup time rises more rapidly due to the increased number of collisions (i.e., multiple topic-to-key mappings being hashed to the same slot). For example, when the number of mappings reaches 5,000, each slot would contain, on average, 50 mappings (i.e., a linked list of 50 mappings), assuming the outputs of the hash function are uniformly distributed. Therefore, on average, 25 comparisons (each involving multiple memory reads) would be needed to complete a single query. In contrast, the hash table with 1,000 slots shows a more stable lookup

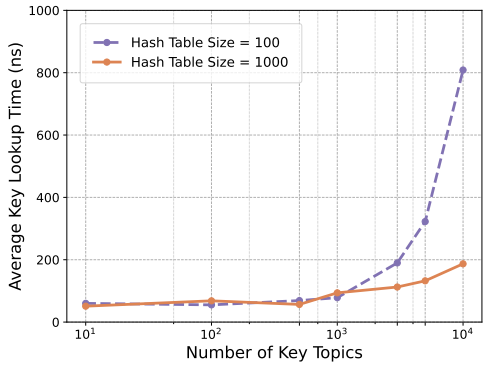


Fig. 11: Average time to look up a key in the topic-to-key hash table. The x-axis represents the number of topic-to-key mappings that the broker maintains. Notice that the x-axis is in logarithmic scale and the y-axis is in nanoseconds.

time, even with 10,000 mappings. This is due to the significantly lower probability of collisions, which results in shorter chains of mappings within each slot. Overall, the lookup time remains very small (tens or hundreds of *nanoseconds*), even with a small hash table size, which indicates that the overhead of key lookup is negligible in WHISPERMQTT.

V. RELATED WORK

Recent research in MQTT for IoT emphasizes optimizing protocol performance under resource constraints and exploring lightweight encryption methods [18]. Yassein *et al.* [19] provide an overview of MQTT and its architecture, message format, scope, and Quality of Service (QoS) levels, while emphasizing its efficiency in low-bandwidth, publish-subscribe communication. In addition to offering a taxonomy of MQTT implementations, Jutadhamakorn *et al.* [20] and Mishra and Kertesz [21] describe the development of a low-cost, scalable MQTT broker utilizing open-source software and M2M protocols, with a focus on clustering for effective message exchange in IoT networks.

In the realm of MQTT with TLS, trends include enhancing TLS efficiency and developing configurations that balance robust security with minimal latency. In [22], Prantl *et al.* provide a comprehensive analysis of the performance degradation associated with the use of MQTT over TLS. The study quantifies the trade-offs inherent in employing TLS with MQTT, highlighting the notable drawbacks in terms of performance. Specifically, the work delves into a detailed examination of the broker connection times and energy efficiency under different conditions, comparing scenarios where TLS is implemented versus those where it is not. This analysis provides valuable insights into the practical implications of integrating TLS into MQTT systems, especially in contexts where performance optimization is critical.

In addition to TLS, various encryption methods can enhance the security of MQTT communications in resource-constrained

IoT environments. Attribute-Based Encryption (ABE) performs particularly efficiently for communication protocols that require fine-grained access control [23]. Liao *et al*. [24] also propose a new approach that combines chaos synchronization with Pairing-Free Cipher text-Policy-ABE (PF-CP-ABE) to improve MQTT security for devices with limited resources. This allows for effective encryption with less computational load. Furthermore, they show using Advanced Encryption Standard (AES) and hybrid encryption approaches can provide strong security while ensuring efficient performance in resource-constrained environments.

Application-level encryption (ALE), also known as at-rest encryption, secures data at the application layer prior to transmission or storage, as WHISPERMQTT does, to protect sensitive data from physical threats [25]. Orobosade *et al.* [26] address security issues in cloud computing caused by a rise of confidential data and malicious behavior. They emphasize the importance of various encryption techniques in cloud security by comparing Elliptic Curve Cryptography (ECC) and Advanced Encryption Standard. The study evaluates encryption/decryption times, throughput, and cipher-text to plaintext ratios while proposing a hybrid model that combines symmetric and asymmetric keys to enhance data secrecy. They showed that ECC improves key management efficiency, AES offers quick and reliable encryption, and the hybrid approach successfully balances speed and security.

Melvin *et al*. [27] explore various MQTT broker protocols, comparing Mosquitto [3], VerneMQ [4], EMQX [6], and HiveMQ [5] to determine the most efficient protocol. They find that Mosquitto stands out due to its low CPU and memory usage, making it highly efficient. However, Mosquitto's performance significantly suffers in environments with packet loss and high client volumes.

VI. CONCLUSION

In this paper, we presented a novel approach to securing MQTT networking in a lightweight manner without modifying the MQTT protocol. The proposed scheme, WHISPERMQTT, reduces the computational overhead on the broker when publishing messages to a group of subscribers by shifting encryption operations to the broker layer. By utilizing both TLS and non-TLS connections, it is able to reduce the number of encryption operations performed by the broker, leading to decreased latency and increased throughput. Our evaluation results show that WHISPERMQTT can reduce the latency overhead of the broker's publication process by 10% to 50% compared to standard TLS-based MQTT communication.

ACKNOWLEDGMENT

This work is supported in part by NCSU Faculty Research and Professional Development fund and the Google Cloud Research Credits program with the award GCP19980904. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

REFERENCES

[1] Mqtt: The standard for iot messaging. https://mqtt.org/.
[2] Internet Engineering Task Force (IETF). Rfc8446: The transport layer security (tls) protocol version 1.3. https://datatracker.ietf.org/doc/html/rfc8446.
[3] Eclipse mosquitto, an open source mqtt broker. https://mosquitto.org/.
[4] Vernemq mqtt broker. https://vernemq.com/intro/mqtt-primer/.
[5] Hivemq mqtt broker. https://www.hivemq.com/products/mqtt-broker/.
[6] Emqx mqtt broker. https://www.emqx.com/en/blog/the-ultimate-guide-to-mqtt-broker-comparison.
[7] Google. Https encryption on the web. https://transparencyreport.google.com/https/overview?hl=en.
[8] Mqtt streaming. https://doc.akka.io/docs/alpakka/current/mqtt-streaming.html.
[9] Google cloud platform overview. https://cloud.google.com/docs/overview.
[10] D. Dinculeană and X. Cheng, "Vulnerabilities and limitations of mqtt protocol used between iot devices," *Applied Sciences*, vol. 9, no. 5, p. 848, 2019.
[11] Eclipse paho client python. https://projects.eclipse.org/projects/iot.paho.
[12] Mqtt version 3.1.1 – oasis standard. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.
[13] M. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," National Institute of Standards and Technology, 2007.
[14] OpenSSL. Tls1.3. https://wiki.openssl.org/index.php/TLS1.3.
[15] Intel advanced encryption standard instructions (aes-ni). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html.
[16] Arm v8-a cryptographic extension. https://developer.arm.com/documentation/101432/r1p2/Functional-description/About-the-Cryptographic-Extension.
[17] Ampere altra family product brief. https://amperecomputing.com/briefs/ampere-altra-family-product-brief.
[18] G. C. Hillar, *MQTT Essentials-A lightweight IoT protocol*. Packt Publishing Ltd, 2017.
[19] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi, "Internet of things: Survey and open issues of mqtt protocol," in *Proceedings of the international conference on engineering & MIS (ICEMIS)*, 2017, pp. 1–6.
[20] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, and D. Kobayashi, "A scalable and low-cost mqtt broker clustering system," in *Proceedings of the 2nd International Conference on Information Technology (INCIT)*, 2017, pp. 1–5.
[21] B. Mishra and A. Kertesz, "The use of mqtt in m2m and iot systems: A survey," *Ieee Access*, vol. 8, pp. 201 071–201 086, 2020.
[22] T. Prantl, L. Iffländer, S. Herrnleben, S. Engel, S. Kounev, and C. Krupitzer, "Performance impact analysis of securing mqtt using tls," in *Proceedings of the ACM/SPEC international conference on performance engineering*, 2021, pp. 241–248.
[23] V. Gupta, S. Khera, and N. Turk, "Mqtt protocol employing iot based home safety system with abe encryption," *Multimedia Tools and Applications*, vol. 80, no. 2, pp. 2931–2949, 2021.
[24] T.-L. Liao, H.-R. Lin, P.-Y. Wan, and J.-J. Yan, "Improved attribute-based encryption using chaos synchronization and its application to mqtt security," *Applied Sciences*, vol. 9, no. 20, p. 4454, 2019.
[25] Y. Ding and K. Klein, "Model-driven application-level encryption for the privacy of e-health data," in *Proceedings of the International Conference on Availability, Reliability and Security*, 2010, pp. 341–346.
[26] A. Orobosade, T. A. Favour-Bethy, A. B. Kayode, and A. J. Gabriel, "Cloud application security using hybrid encryption," *Communications on Applied Electronics*, vol. 7, no. 33, pp. 25–31, 2020.
[27] M. Bender, E. Kirdan, M.-O. Pahl, and G. Carle, "Open-source mqtt evaluation," in *Proceedings of the 18th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2021, pp. 1–4.