Array 30 (2026) 100972

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/locate/array

Efficient and dynamic access control with end-to-end message security for MQTT

Liang Chen [a],*, James Wilson [b]

[a] School of Physics, Engineering and Computer Science, University of Hertfordshire, Hatfield, AL10 9AB, United Kingdom
[b] Warwick Manufacturing Group, University of Warwick, Coventry, CV4 7AL, United Kingdom

ARTICLE INFO

Keywords:
IoT
MQTT
Access control
Message security

ABSTRACT

The Internet of Things (IoT) has become deeply integrated into daily life, with devices now monitoring our health, managing homes, and controlling critical infrastructure. Ensuring the security of these interconnected systems is therefore essential. Among IoT communication protocols, MQTT has emerged as the most widely adopted lightweight messaging standard, enabling efficient publish–subscribe interactions between devices. However, existing solutions for authorisation and message-level security in MQTT are unnecessarily computationally expensive, making them unsuitable for constrained devices. In this paper, we introduce a novel access control policy model and an accompanying enforcement and message-security scheme designed specifically for MQTT. We are not aware of an existing MQTT policy model that can automatically assign and maintain access-control labels as new topics appear in the dynamic topic hierarchy. Our enforcement scheme uses lightweight symmetric cryptography to provide end-to-end payload confidentiality (brokers and mediators cannot read plaintext), rather than hop-by-hop protection as in MQTT over TLS. Our performance evaluation shows that our scheme significantly reduces computational and memory overhead compared to TLS, making it far more suitable for constrained IoT devices. This makes it practical for deployments where clients cannot afford a full TLS stack but still require confidentiality and policy enforcement.

1. Introduction

Internet of Things (IoT) devices are becoming increasingly popular and are now used for a wide range of critical applications [1]. In 2022, there were over 13 billion connected IoT devices worldwide, and this number is projected to rise to more than 29 billion by 2030, spanning industries from healthcare to manufacturing [2]. IoT devices connect nearly every aspect of our lives — from doorbells that allow us to monitor our homes remotely, and autonomous vehicles that communicate while driving, to heart rate sensors in hospitals and temperature sensors in industrial control systems [3].

There has been extensive development of IoT standards and protocols that enable devices to exchange data in a structured and meaningful way. One of the most widely adopted protocols is Message Queuing Telemetry Transport (MQTT) [4]. MQTT is a lightweight and scalable messaging protocol specifically designed for connecting a large number of resource-constrained devices. It operates on the publish–subscribe model, which decouples the message sender (publisher) from the message receiver (subscriber). A third component, known as the broker, filters incoming messages from publishers and distributes them appropriately to subscribers.

Given that IoT systems often handle sensitive data – including health information and control over critical infrastructure – it is essential to ensure their security. This involves two key aspects: (1) protecting message confidentiality and integrity (so that messages cannot be read or altered by unauthorised devices), and (2) enforcing access control, which defines what each device is permitted to do. Since MQTT is the de facto communication protocol for IoT systems, addressing these two aspects within MQTT is vital to achieving secure and trustworthy deployments.

While some access control mechanisms exist for MQTT [3,5], they typically suffer from several limitations:

- MQTT topics (categories of messages) evolve over time, yet no existing access control policy, to our knowledge, can adapt to these changes without manual intervention.
- Many existing access control models are complex and time-consuming to define and are not scalable to the large numbers of devices typically found in IoT environments.
- Existing schemes that enforce access control policies often introduce additional computational overhead to the MQTT architecture, making them impractical for real-world deployment on constrained devices.

* Corresponding author.
E-mail address: l.chen26@herts.ac.uk (L. Chen).

https://doi.org/10.1016/j.array.2026.100972
Received 17 December 2025; Received in revised form 20 April 2026; Accepted 1 June 2026
Available online 9 June 2026

Without effective and efficient access control, many MQTT deployments rely on minimal protections, leaving them vulnerable to compromise. A single compromised device can disrupt the entire system and expose sensitive data across the network. Therefore, it is crucial to develop a lightweight yet adaptive access control mechanism to strengthen defence-in-depth for MQTT-enabled IoT environments, where devices are particularly exposed to cyberattacks [6].

In terms of message security, Transport Layer Security (TLS) [7] remains the only widely adopted option. However, many IoT devices lack the computational capacity to perform TLS operations efficiently. As a result, message-level security is often neglected despite the sensitivity of the transmitted data. Moreover, TLS provides only hop-by-hop protection, allowing MQTT brokers to access message contents. Several studies [8–11] have explored end-to-end MQTT payload encryption schemes, but their experimental results do not convincingly demonstrate consistent performance improvements over TLS.

In this paper, we propose an access control and message security scheme for MQTT that is suitable for resource-constrained clients. More specifically, our contributions are as follows:

- We define a scalable access control policy model that evolves dynamically with MQTT topics. The model assigns each topic a security label on first publication and supports simple, low-maintenance policies that adapt automatically as the topic tree grows.
- We design an enforcement and message security scheme based on the proposed policy model, which provides end-to-end payload confidentiality and mitigates information leakage. Our scheme uses standard symmetric primitives (AEAD, MACs, and hash-based key derivation), and it combines dynamic topic labelling with mediator-based publish enforcement to keep brokers and mediators from accessing plaintext.
- We evaluate a prototype implementation and show that, in our measured setup, our scheme reduces CPU and memory overhead compared with TLS during connection/setup, while remaining competitive during steady-state messaging.

The rest of the paper is organised in the following way. Section 2 contains essential background materials on MQTT and information flow policy models. In Section 3 we define our policy model for authorisation in MQTT. We then present our design of policy enforcement and message security schemes in Section 4. In Section 5 we discuss our performance experiments and results. Finally, in Section 7, we conclude the paper with a summary of our contributions and ideas for future work.

2. Background

2.1. MQTT – the standard for IoT messaging

MQTT [4,12,13] is a widely used communication protocol designed for resource-constrained devices with poor network connections. It is an application layer protocol that usually runs over TCP. It organises messages into topics which are UTF-8 strings consisting of one or more topic levels separated by a forward slash like a file path. Therefore the set of topics in use at any one time can be thought of as a tree-like structure.[1] as illustrated in Fig. 1 The topic hierarchy reflects our running example where the factory has several machines on the assembly line. Each machine monitors both the temperature and the position of its arm, which can be adjusted to enable movement. The machines are equipped with low-powered sensors that communicate via MQTT. A control room monitors all the information to ensure that everything is running smoothly.

[1] In fact, topics form a forest of many trees because they may not have a common root.

MQTT is a *publish–subscribe protocol* with a central server called the 'broker'. A client can 'publish' a message to a topic by sending a PUBLISH packet to a broker. A client can also 'subscribe' to a topic (or multiple topics using wildcards) by sending a SUBSCRIBE packet to a broker. In this way the broker will forward any messages that are published to any subscribed topics to the client.

MQTT topics are *dynamic* in nature – a topic does not exist until it is published to for the first time, but that does not affect the ability to subscribe to it. For example, suppose the initial state of topics is as in Fig. 1 with two machines and suppose the control room is subscribed to the topic `machine/+/temperature`. When a third machine is added, the control room will automatically receive messages published to the topic `machine/3/temperature` even though it does not exist at subscription time. Existing access control policies for MQTT are static and unable to adapt to dynamically created topics without administrator intervention [3,5]. In contrast, our policy model is designed to do this.

To prevent secret messages from being intercepted and important messages from being changed, they must be encrypted and authenticated for confidentiality and integrity. MQTT does not provide any native way to do either of these so in sensitive scenarios, MQTT is usually tunnelled through TLS [7]. However, TLS is not a good solution for the following reasons:

- TLS (particularly the handshake) is too resource-intensive for many low-powered MQTT clients [8,11].
- TLS only provides hop-to-hop protection (from publisher to broker and separately from broker to subscriber) which means the broker has access to the unencrypted messages – it is not end-to-end encryption [5].

Therefore, an alternative authenticated-encryption method is needed that is less resource-intensive and provides end-to-end protection.

2.2. Information flow policies

One method of defining access control policies is with an *information flow policy* model. These are based on the flow of information between objects (what we want to protect) and subjects (that require access to particular objects) as well as between high and low confidentiality and integrity. In MQTT, we can consider subscribing (or rather, reading a message) as an information flow from an object (topic) to a subject (client), and publishing as an information flow from a subject (client) to an object (topic).

The Bell-LaPadula (BLP) Model [14] is an information flow policy model that ensures *confidentiality*, while the Biba Model [15] is an information flow policy model that ensures *integrity*. The BLP model assigns each subject a security label as its 'clearance' and each object a security label as its 'classification' using the function $\lambda : S \cup O \to L$, where S is the set of subjects and O is the set of objects. We define the set of security labels L as a partially ordered set (poset) $(L, \leqslant)$, where $\leqslant$, in other words, is a reflexive, anti-symmetric, transitive, binary relation defined on L.[2] The main goal of the BLP model is to prevent information flowing from an object with a high classification to a subject with a low clearance. To achieve this, there are two properties that are required to be enforced in the system:

- **Simple Security Property ('no read-up' rule):** A subject s can *read* object o if and only if $\lambda(s) \geqslant \lambda(o)$. This is the most intuitive way of enforcing confidentiality — subjects can only read objects with lower or equal classifications than their clearance.

[2] In the original BLP model, the security label L is defined to be $L \subseteq C \times 2^K$, where C is a totally ordered set of security classifications, K is set of need-to-know categories and $(c_1, K_1) \leqslant (c_2, K_2)$ if, and only if, $c_1 \leqslant c_2$ and $K_1 \subseteq K_2$. For the sake of simplicity, we define security labels as $(L, \leqslant)$ in this paper.

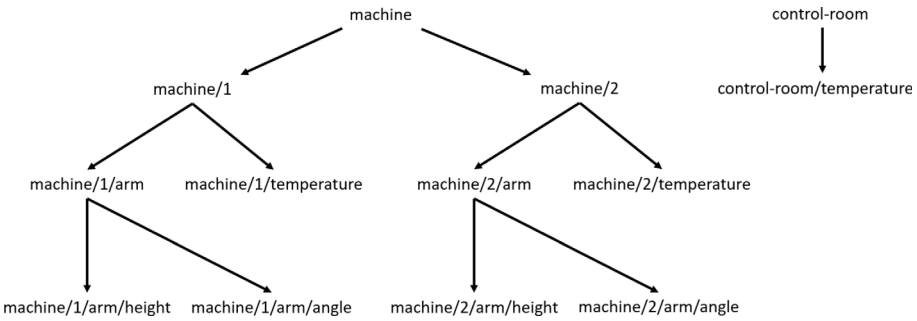

Fig. 1. An example of a topics forest that might appear in a factory.

- **Star Property ('no write-down' rule):** A subject s can *write* to object o if and only if $\lambda(s) \leqslant \lambda(o)$. This is to prevent sensitive information from flowing to a lower classification, which would enable subjects with a lower clearance to read it.

The Biba model [15] defines a partially ordered set of integrity levels $(I, \leqslant)$, which is analogous to the set of security levels in the BLP model. Let S be a set of subjects and O be a set of objects, we define a function $\lambda' : S \cup O \to I$ that assigns an integrity level to every subject and object, where $\lambda'(s)$ represents the *degree of trustworthiness* of subject s and $\lambda'(o)$ represents the *level of importance or quality* of information o. The Biba integrity model is to prevent information flowing from an untrusted subject to an object that must be accurate. It achieves this with two core properties that are the mathematical dual of the two properties in the BLP model:

- **'No write-up' rule:** A subject s can *write* to object o if and only if $\lambda'(s) \geqslant \lambda'(o)$. This is the most intuitive way of enforcing integrity — subjects must be trustworthy enough to write to objects that must be accurate.
- **'No read-down' rule:** A subject s can *read* object o if and only if $\lambda'(s) \leqslant \lambda'(o)$. This is to prevent a trusted subject from reading false information from a contaminated object and later writing it to an important object.

The Biba model has several policy variations, including the *Low-Water Mark Policy* and the *Ring Policy*. The Ring Policy is a relaxed version, where subjects can read any object (no restriction on reading down) but must not write up.

3. A policy model for MQTT

In many situations, it is undesirable to allow all clients to publish and subscribe to all topics. For example, only a temperature sensor should be able to publish to the topic `machine/1/temperature`, not a smart bulb. Thus some kind of access control policy is required to specify which clients can publish and subscribe to which topics. We believe that any way of defining access control policies for MQTT must maintain both the secrecy of sensitive information (confidentiality) and the trustworthiness of the data itself (integrity). In our running example, without confidentiality, designs being sent to the machines could be stolen by competitors. Without integrity, temperature readings could be spoofed so the control room believes everything is normal when there could be an emergency.

Since the topic hierarchy and clients in the MQTT are dynamically evolving, we also believe that fine-grained access control models, like RBAC or ABAC, lead to unnecessary administrative maintenance, which is not a pragmatic solution. We argue that information flow policies are ideal for MQTT, where security levels and categories that rarely change can be used to construct a partially ordered set to which it is straightforward to assign and reassign subjects and objects. In this section, we define a model of information flow policies for MQTT based on the security properties of both the BLP model and the Biba model with the following components:

- $(L, \leqslant)$: a partially ordered set of security labels.
- C: a set of clients. Clients can dynamically join and leave the system at all times, which is one of the defining features of MQTT. They are the subjects who seek to publish and subscribe to topics in the system.
- T: a set of topics. At first glance, individual messages of a topic are what needs protecting, so it would make sense for them to be the objects. However, there are several reasons why it is appropriate to protect topics instead. Firstly, in MQTT, each message is published to a single topic which naturally groups messages by content. Since it is the content that needs protecting, it is likely that all messages published to the same topic have the same security requirements. For example, all messages published to the topic `machine/1/temperature` are temperatures and if one temperature should only be visible to certain subjects then so should all temperatures on that topic. Secondly, in MQTT, clients subscribe to topics to receive all messages on those topics. They cannot choose to only receive certain messages, even if different messages have different protection needs. Hence treating individual messages as protected objects would contradict the very concept of MQTT grouping messages into topics.
- σ: a function $\sigma : C \cup T \to L$ that assigns a security label to each client and topic. When we assign security labels to topics that are published to, this means the *full* topic, e.g. `machine/1/temperature`. This has no bearing on the security levels of parent topics, e.g. `machine` and `machine/1` – these are not assigned a security label until they are published to separately. More specifically, we expect all topics that are published to (and thereby assigned a security label) to be leaves in the topic tree since the final topic level (e.g. `temperature`) describes the sorts of messages it contains, whereas higher topic levels (`machine` and `1`) here just organise them. Nevertheless, there is nothing preventing intermediate-node topics (e.g. `machine/1`) from being published to. This makes this policy model *non-hierarchical* in a sense that the security label of each node in the topic tree is independent and not related to the security labels of its parent or child topics. Since the parent and child nodes just may have different protection needs, there is no benefit of authorisation inheritance among the topic tree.
- $R = \{r_p, r_s\}$: two policy rules. The subscription rule r_s defines that a client $c \in C$ can subscribe to topic $t \in T$ if and only if $\sigma(c) \geqslant \sigma(t)$. The publishing rule r_p defines that a client $c \in C$ can publish messages to topic t if $t \notin T$ otherwise $\sigma(c) = \sigma(t)$. When a client c attempts to publish to topic t that does not exist in the topic tree T, the r_p always permit the request, resulting in the topic t being assigned the same security label as client c, that is $\sigma(t) \leftarrow \sigma(c)$. This assignment also enables the client c to subsequently publish messages to topic t which are allowed by rule r_p. It shows that our approach is flexible to cope with the dynamically changing topic tree without administrators' manual intervention.

3.1. Justifying our policy model

We can see that the two policy rules $R = \{r_p, r_s\}$ result from combining the rules of the Bell–LaPadula model with the Ring Policy (only the 'no write-up' rule) from the Biba model. The 'no read-down' rule in the Biba model is intended to prevent a subject from reading false information at a lower level, believing it to be true, and then writing it to an object at its own level. While this makes sense in an integrity-only policy model, in a combined model it is impractical to prevent a client from reading objects with lower labels than its own merely because it might read something false. The labels themselves provide an indication of *trustworthiness*, which clients can use to judge the reliability of messages they receive. This places some trust in clients, but the 'no write-up' rule still exists to limit how far untrusted data can propagate.

For integrity, we also make the publish rule for existing topics intentionally strict. Requiring $\sigma(c) = \sigma(t)$ prevents clients at other labels from injecting data into a topic that subscribers may rely on. This matches many IoT deployments where sensor topics are single-writer and control topics are published only by a small set of authorised controllers. If a deployment needs multiple publishers for the same topic, the rule can be relaxed in a controlled way (for example, by keeping a list of authorised publishers for that topic), which we leave for future work.

Our policy model is capable of evolving to control access to each new topic. When a topic t is published for the first time by client c, it is assigned the same security label l as c. This is the only possible label assignment for this topic that complies with the no 'write-up' and no 'write-down' rules. If a lower label were assigned to the topic, the client would effectively be writing to a lower-labelled topic and thus violate the no 'write-down' rule; conversely, assigning a higher label would violate the no 'write-up' rule. While the rest of the policy model (the assignment of labels to clients) follows a *mandatory* access control approach, since it is defined by the administrator, this mechanism introduces an element of *discretionary* access control. Although a client cannot choose the security label of the topics it publishes to (as it is fixed to the client's own label), its actions still influence the topic's label — if a client with a different label had published to the topic first, it would have been assigned a different label. Consequently, the set of topics T can initially be empty and will grow as topics are published. Any topics that the administrator assigns a label to during setup are fixed to that label — as if a client with that label had already published to them. In practice, MQTT topics can appear at any time, so our model labels topics as they are introduced and applies access control automatically as the topic tree grows.

3.2. Considering special labels

We may augment the poset $(L, \leqslant)$ to create a *least* element and a *greatest* element:

- Add a new element $\bot$ and define $\bot < l$ for all $l \in L$, which means $\bot$ becomes the least element.
- Add a new element $\top$ and define $\top > l$ for all $l \in L$, which means $\top$ becomes the greatest element.

This is useful for expressing a policy in which any client assigned $\top$ may subscribe to all topics, and any topic assigned $\bot$ may be subscribed to by all clients. Our enforcement scheme, presented in Section 4, highlights the convenience of these labels in supporting policy changes.

We also introduce a special label, $\ominus$, which can be added to the poset $(L, \leqslant)$ as an isolated element, serving to disable clients or reset topics.

- When a client is assigned $\ominus$, it cannot publish or subscribe to any topic (whether or not it exists yet) – the client itself does not exist as far as the system is concerned so all authorisation requests are denied.

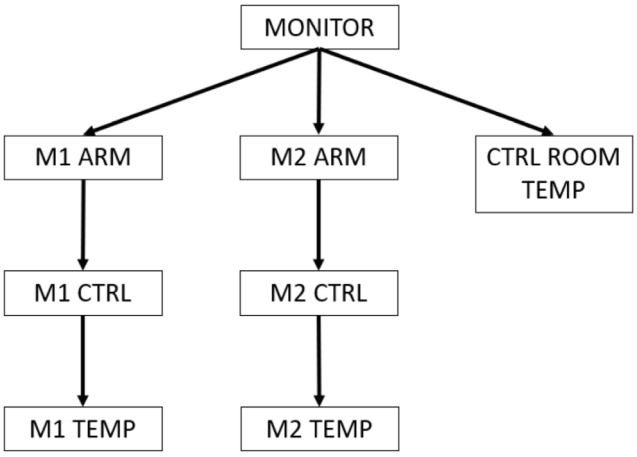

Fig. 2. The poset of security labels used for the factory example.

- When a topic is assigned $\ominus$, it has no label so is treated as if it has never been published to before and will be assigned the label of the next client to publish to it.

Effectively, clients and topics assigned $\ominus$ are not elements of C or T, but it can be conceptually helpful to think of adding or removing clients and topics from these sets as equivalent to setting their label to or from $\ominus$ instead. This means that every modification to the policy (except for changes to L itself) is simply an update to the assignment given by σ. This is used in Section 4 to simplify the necessary steps when changing the assignment.

3.3. Creating a policy for the running example

With the policy model established, we can define a policy for the factory example shown in Fig. 1. We first examine the MQTT clients and the topics to which they need to publish and subscribe. Ideally, each client would be restricted to these topics only; however, fine-grained control requires a more detailed case study.

- Each machine (number `x`) has a temperature sensor which is a small, low-powered MQTT client. It should be able to publish to the topic `machine/x/temperature` and not subscribe to any topics.
- Each machine has another small, low-powered MQTT client that operates the motors in the arm to move it. It should not be able to publish, and should only be able to subscribe to the topics `machine/x/arm/#`, from which it receives its instructions.
- Each machine also has a control panel through which a human operator or computer program can control the machine's arm and monitor its temperature. It should be able to publish to `machine/x/arm/angle` and `machine/x/arm/height`, and subscribe to `machine/x/temperature`.
- The control room monitoring station is another MQTT client that should be able to subscribe to `#`, but not publish to any topics.
- The control room temperature sensor is an MQTT client that should be able to publish to `control-room/temperature`, but not subscribe to any topics.

With our policy model, it is not possible to completely disallow publishing or subscribing to all topics. However, it is possible to restrict this to a single label that is not assigned to any existing topics, so that only new topics can be accessed.

To enforce this policy, we create the poset of security labels shown in Fig. 2 and assign them to the clients and topics as in Table 1.

Table 1
The σ function assigning a label to each client and topic.

The set of clients C	$\sigma(c)$ for $c \in C$
Machine x's temperature sensor	`Mx TEMP`
Machine x's arm operator	`Mx ARM`
Machine x's control panel	`Mx CTRL`
Control room's temperature sensor	`CTRL ROOM TEMP`
Control room's monitoring station	`MONITOR`
The set of topics T	**$\sigma(t)$ for $t \in T$**
`machine/x/temperature`	`Mx TEMP`
`machine/x/arm/height`	`Mx CTRL`
`machine/x/arm/angle`	`Mx CTRL`
`control-room/temperature`	`CTRL ROOM TEMP`

Each machine's temperature sensor is assigned the security label `Mx TEMP`, and the control room's temperature sensor is assigned `CTRL ROOM TEMP`. These sensors publish to the topics `machine/x/temperature` and `control-room/temperature`, respectively, so these topics are assigned the same labels. As leaf nodes in the security label hierarchy, these sensors can only publish to and subscribe to topics that share the same security label.

Each machine's control panel is assigned the security label `Mx CTRL`, which is the parent of `Mx TEMP`, allowing it to subscribe to `machine/x/temperature`. To control the movement of the machine's arm, it publishes to `machine/x/arm/angle` and `machine/x/arm/height`, so these topics are also assigned the label `Mx CTRL`.

Each machine's client that operates the motors must be able to subscribe to `machine/x/arm/#`, but not publish to these topics. Therefore, it should be assigned a label that is the parent of `Mx CTRL`, `Mx ARM`. Note that this also allows it to subscribe to `machine/x/temperature`, as discussed below.

Finally, the control room monitoring station is assigned the label `MONITOR`, which is an ancestor of all other security labels. This allows it to subscribe to all topics; however, since no topics are assigned this label, it cannot publish to any existing topic.

Note that only the temperature sensors can publish to the temperature topics, even though other clients can read them, due to the "no write-down" and "no write-up" rules. Therefore, these values can be trusted. Also note that the clients of one machine cannot publish to or subscribe to topics related to other machines. The only shared entity is the control room monitoring system.

Any policy in this model must concede that each machine's arm operator can read its temperature through transitivity. This occurs because the machine's control panel must read the temperature and publish the arm position, and, in turn, the machine's operator must read the arm position. This issue could potentially be resolved by having the control panel act as two logical clients – one subscribing to the temperature and the other publishing the arm position – however this approach requires further investigation.

4. Design and enforcement of a secure MQTT scheme

In this section we present the design, operation, and enforcement rationale of a secure scheme that enforces the policy model introduced in Section 3 by embedding lightweight cryptographic controls into the standard MQTT publish–subscribe architecture.

4.1. Architectural overview

Traditional MQTT deployments rely on TLS to protect communication channels between clients and brokers. However, TLS secures only transport endpoints rather than message payloads, offering no post-delivery control over which subscribers can decrypt brokered messages. Moreover, TLS introduces asymmetric cryptographic overhead, making it impractical for low-power IoT devices [9,10]. Our scheme decouples *policy enforcement* from *payload confidentiality* by introducing *mediators* that manage publishing rights and by using symmetric encryption to control subscriber access.

Fig. 3 illustrates the entities and their interactions as an extension of the standard MQTT architecture:

- Clients: MQTT publishers and subscribers that encrypt and decrypt payloads using symmetric primitives.
- Mediator: An enforcement entity that validates publisher authorisation and re-encrypts messages before forwarding them to brokers. A single physical device may act as both a mediator and broker, or these may reside on separate devices.
- Broker: A standard MQTT broker that routes PUBLISH packets between subscribers but never accesses plaintext messages.
- Key Generator (KG): A trusted authority that generates symmetric keys, maintains label hierarchies, and publishes public derivation values.

The complete journey of a PUBLISH packet, also illustrated in Fig. 3, consists of three principal legs:

1. Client-to-Mediator: the client sends a publish request to a mediator, which performs checks and prepares the message for delivery.
2. Mediator-to-Broker: the mediator forwards the message to the broker for normal MQTT routing.
3. Broker-to-Subscriber: the broker delivers the message to subscribers, who verify and decrypt locally if authorised.

This structure separates enforcement from routing, reducing performance overhead while providing cryptographic guarantees of message authenticity, confidentiality and integrity.

4.2. Threat model and security goals

We assume a standard network attacker who can eavesdrop, inject, drop, delay, and replay packets. In addition, some clients may be compromised and behave maliciously.

Our trust assumptions are as follows:

- **Key Generator (KG):** trusted. The KG correctly generates keys, publishes derivation values, and provisions clients and mediators.
- **Broker:** not trusted for confidentiality. A broker may be honest-but-curious (attempting to learn payload contents) or compromised. We therefore do not rely on the broker for payload secrecy or access control.
- **Mediator:** trusted for enforcement but not for confidentiality. A mediator is expected to mediate publish requests and authenticate client packets, but it is not authorised to learn plaintext payloads. We therefore design the cryptography so that a mediator cannot decrypt payloads even if it is honest-but-curious. If a mediator is compromised and behaves maliciously, payload confidentiality should still hold, but availability and correct enforcement may be affected.

We aim to achieve the following properties, under standard cryptographic assumptions for the authenticated-encryption and MAC primitives used in the scheme. Appendix restates these goals in a game-based form and provides short proof sketches.

- **End-to-end payload confidentiality:** brokers and mediators should not learn plaintext payloads. Only clients authorised to read a topic (i.e. whose labels satisfy $\sigma(t) \leqslant \sigma(c)$) and that hold the required keys should be able to decrypt.
- **Payload integrity and authenticity:** a subscriber should detect any modification of a payload in transit. Integrity here means tamper detection and data-origin authentication at the topic/label level (not non-repudiation).

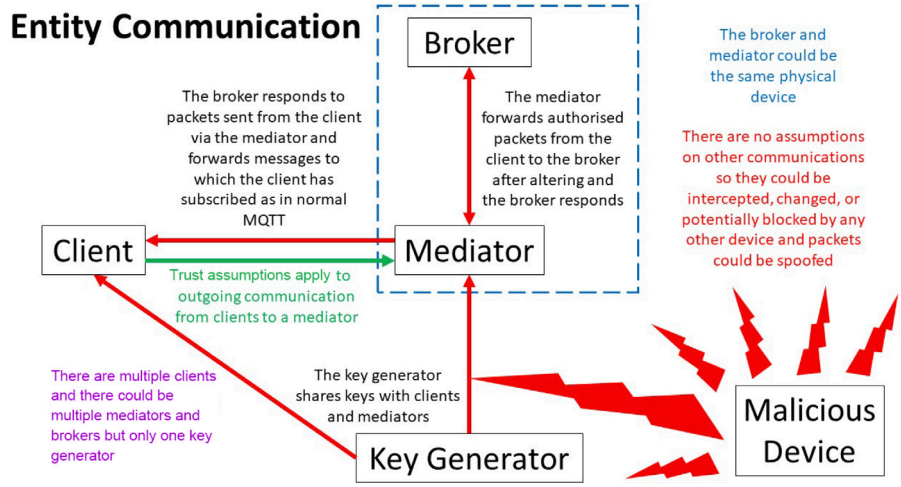

Fig. 3. Entities in the proposed enforcement scheme and their communication relationships.

- **Policy enforcement for publishing:** unauthorised publish attempts should be rejected by mediators according to rule r_p, including topic creation and label assignment for new topics.

Our scheme is not designed to guarantee availability against denial-of-service attacks (a compromised mediator can always drop packets), and it does not provide non-repudiation because we intentionally avoid asymmetric signatures for constrained devices. We also do not attempt to hide metadata such as topic names, message timing, or traffic volume, so traffic analysis remains possible. We discuss practical deployment assumptions and mediator-bypass hardening in Section 4.6.

4.3. Design rationale

In our architecture, the mediator is responsible for enforcing publishing rules, but payload confidentiality must not depend on trusting the mediator. We therefore design the message protection so that a mediator can validate and process messages without learning plaintext. Mediators operate under a *semi-trusted* assumption: they reliably enforce access control but are not authorised to inspect plaintext messages. This departs from fully trusted gateways and reflects real-world IoT deployments, where perfect isolation cannot be guaranteed. Clients therefore operate under one of three conditions: (1) all outbound traffic passes through a trusted mediator, (2) the client is trusted to apply encryption correctly, or (3) all subscribing clients are trusted not to leak decrypted content.

Embedding access control logic within brokers reduces modularity and interoperability. Instead, mediators act as *reference monitors*, guaranteeing complete mediation of publish operations. They verify publisher authorisation and prevent unauthorised topic creation. While mediators enforce publishing rights, subscription rights are enforced cryptographically: any client may subscribe, but only those holding valid decryption keys can access plaintext. Thus, confidentiality is maintained even if brokers or mediators behave honestly-but-curiously.

We employ ChaCha20-Poly1305 [16] as the Authenticated Encryption with Associated Data (AEAD) primitive for all encryption and authentication operations described in Section 4.4. This symmetric AEAD algorithm provides both confidentiality and integrity in a single operation, with far lower computational cost than asymmetric protocols such as TLS. Its performance characteristics make it particularly suitable for constrained IoT nodes, reducing latency and energy consumption while ensuring message authenticity.

Our scheme uses symmetric keys because they are practical for constrained devices. As a result, the integrity we provide is *tamper detection and data-origin authentication at the topic/label level*, rather than non-repudiation. If multiple publishers are authorised to publish at the same label (and therefore share the same topic/label key), then any of them can produce a valid authentication tag for that topic. This is sufficient for many IoT deployments where trust is managed at the level of device classes or labels, but it does not let a subscriber cryptographically distinguish which authorised publisher created a message. If per-sender authenticity is required, the mediator can additionally bind a publisher identifier to each message (for example as associated data) and verify a per-sender MAC under k_c^M (or under a per-sender topic MAC key), which we leave for future work.

To prevent mediators from viewing message content, we use a *label-scoped anti-mediator key hierarchy*. In particular, for each label $l \in L$ we define an anti-mediator key $\bar{k}_l$ that is given only to clients (never to mediators or brokers). When a client publishes, it encrypts the plaintext payload using the anti-mediator key for the payload's label. The mediator therefore sees only ciphertext, but can still check authorisation and forward messages by re-encrypting for topic-level distribution. This also limits the damage if a client is compromised: an attacker can only decrypt payloads for topics that the compromised client is allowed to access, rather than for the entire system.

We use a Direct Key Encrypting Key Assignment Scheme (DKEKAS) [17] so that clients can derive the keys they need locally, rather than requiring the KG to send every key explicitly. Each label $l \in L$ is associated with a symmetric key k_l. For every pair (n, l) such that $n \leqslant l$, the KG publishes public derivation data

$$z_{n,l} = k_n \oplus H(n||k_l).$$

Using this public data, a client holding k_l can compute subordinate keys k_n through lightweight hash and XOR operations. This allows higher-privilege clients to access lower-level topics while preventing reverse derivation.

We apply the same derivation approach to anti-mediator keys. Each label l has an anti-mediator key $\bar{k}_l$, and for every pair (n, l) such that $n \leqslant l$, the KG publishes

$$\bar{z}_{n,l} = \bar{k}_n \oplus H(n||\bar{k}_l),$$

so that a client holding $\bar{k}_l$ can derive $\bar{k}_n$ locally, while a mediator (which never receives any $\bar{k}$ key) cannot derive these keys.

Each client is provisioned with a unique pre-shared secret pss_c, enabling both the KG and the client to derive a temporary session key k_c for secure provisioning. This symmetric provisioning avoids the complexity of asymmetric key exchanges while ensuring confidentiality and authenticity during key delivery. The KG uses k_c to encrypt and transmit the client's label key k_l, mediator link key k_c^M, and the client's anti-mediator root key $\bar{k}_{\sigma(c)}$ (from which it can derive lower anti-mediator keys using the published $\bar{z}$ values). Because pss_c is unique per

client, compromise of one device does not affect the security of others. This model supports scalable device onboarding in IoT ecosystems without requiring a full public-key infrastructure.

4.4. System operation

Initial setup. System configuration begins by defining the poset of security labels L and the σ function that associates each client and topic with a label. The KG creates an anti-mediator key $\bar{k}_l$ for each label $l \in L$; generates a symmetric key k_l for each $l \in L$; and, for every client c, derives its provisioning key k_c (from pss_c) and a mediator link key k_c^M. The KG then publishes public derivation data $z_{n,l}$ and $\bar{z}_{n,l}$ for all $n \leqslant l$, enabling authorised clients to derive both subordinate topic/label keys and subordinate anti-mediator keys locally.

Algorithm 1 Key Generator Setup

(Anti-mediator hierarchy) For each label $l \in L$, generate and store anti-mediator key $\bar{k}_l$ (never shared with mediators).
(Topic/label hierarchy) For each label $l \in L$, generate and store symmetric topic key k_l.
for each label $l \in L$ **do**
  for each $n \leqslant l$, $n \neq l$ **do**
    Publish $z_{n,l} = k_n \oplus H(n \| k_l)$.
    Publish $\bar{z}_{n,l} = \bar{k}_n \oplus H(n \| \bar{k}_l)$.
  end for
  for each client c with $\sigma(c) = l$ **do**
    Derive k_c from pss_c.
    Generate mediator link key k_c^M.
    Send $\{k_l, \bar{k}_l, k_c^M, \sigma(c)\}$ to c, encrypted under k_c.
  end for
end for
Send $\{L, C, T, \sigma(), \{k_c^M\}\}$ to mediators over an authenticated channel (e.g. TLS).

Clients receive only the keys relevant to their label (including $\bar{k}_{\sigma(c)}$), and mediators learn which clients they serve but never receive any anti-mediator keys.

Publishing messages. When a client c (acting as publisher) sends a message m to topic t, it first encrypts m using the anti-mediator key associated with its label (which equals the label of t under rule r_p), producing $m_1 = \mathrm{Enc}(\bar{k}_{\sigma(c)}, m)$. It then encrypts m_1 with k_c^M to produce m_2, and computes a MAC μ_1 over (m_2, t, s_1) using k_c^M, where s_1 is a timestamp. The resulting PUBLISH packet contains (m_2, s_1, μ_1), as detailed in Algorithm 2.

Algorithm 2 Mediator Publish Authorisation

Require: Client c, topic t, payload (m_2, s_1, μ_1).
Verify s_1 is recent; if stale, discard.
Decrypt m_2 with k_c^M to recover m_1 (still ciphertext under $\bar{k}_{\sigma(c)}$) and verify μ_1; if verification fails, discard.
if $t \in T$ **then**
  Allow only if $\sigma(c) = \sigma(t)$.
else
  Add t to T and assign $\sigma(t) \leftarrow \sigma(c)$; synchronise across mediators.
end if
Derive $k_{\sigma(t)}$.
Encrypt m_1 with $k_{\sigma(t)}$ to obtain m_3.
Generate timestamp s_2 and compute $\mu_2 = \mathrm{MAC}(k_{\sigma(t)}, m_3 \| s_2 \| \sigma(t) \| t)$.
Replace payload with $(m_3, s_2, \sigma(t), \mu_2)$ and forward securely to broker.

The mediator validates publication requests, ensures label consistency, and performs re-encryption without accessing plaintext. If a topic is encountered for the first time, it inherits the label of the publisher.

Rule r_p allows the first publication of a topic to create it and assign it the publisher's label, which matches MQTT's dynamic topic hierarchy. In some deployments, administrators may still want extra control over *which* clients are allowed to create topics in sensitive namespaces (for example, control topics such as `machine/x/arm/#`). To support this, we allow the mediator to apply simple topic-creation checks before accepting the "new topic" case in Algorithm 2. One way to do this is to reserve critical topic prefixes during setup, by pre-populating T with administrator-labelled topics so that those names cannot be claimed by accident. Another way is to use simple pattern-based rules (based on topic filters) that restrict which labels are allowed to create topics under certain prefixes. For the most sensitive namespaces, the mediator can also require explicit authorisation for topic creation, such as an allowlist by client identity or label. These checks prevent low-privilege or misconfigured clients from claiming important topic names, while still allowing automatic topic growth in non-critical parts of the topic tree. We plan to explore these governance rules in more detail in future work.

Subscription and decryption. Upon receiving $(m_3, s_2, \sigma(t), \mu_2)$ for topic t, a subscriber c' verifies integrity and decrypts in two stages, as shown in Algorithm 3.

Algorithm 3 Subscriber Decryption Procedure

Require: Message $(m_3, s_2, \sigma(t), \mu_2)$ on topic t.
Verify s_2 is recent; if stale, discard.
Derive $k_{\sigma(t)}$ if $\sigma(t) \leqslant \sigma(c')$: $k_{\sigma(t)} \leftarrow z_{\sigma(t),\sigma(c')} \oplus H(\sigma(t) \| k_{\sigma(c')})$.
Verify μ_2 using $k_{\sigma(t)}$; if invalid, discard.
Decrypt m_3 under $k_{\sigma(t)}$ to recover m_1.
Derive $\bar{k}_{\sigma(t)}$ if $\sigma(t) \leqslant \sigma(c')$: $\bar{k}_{\sigma(t)} \leftarrow \bar{z}_{\sigma(t),\sigma(c')} \oplus H(\sigma(t) \| \bar{k}_{\sigma(c')})$.
Decrypt m_1 under $\bar{k}_{\sigma(t)}$ to obtain plaintext m.

Successful decryption requires that the subscriber's label dominates the topic label ($\sigma(t) \leqslant \sigma(c')$), and that the subscriber can obtain the required keys $k_{\sigma(t)}$ and $\bar{k}_{\sigma(t)}$. This label condition is what allows the subscriber to derive $\bar{k}_{\sigma(t)}$ (and $k_{\sigma(t)}$) from the published derivation values, so unauthorised subscribers cannot decrypt even if they receive the ciphertext.

Non-PUBLISH processing. Other MQTT packets (CONNECT, SUBSCRIBE, PING, etc.) do not carry confidential payloads but still require integrity and replay protection. The client and mediator therefore attach and verify timestamps and MACs as shown in Algorithm 4.

Algorithm 4 Non-PUBLISH Packet Authentication

Require: Packet P from client c with timestamp s.
Client computes $\mu = \mathrm{MAC}(k_c^M, P \| s)$ and sends (P, s, μ) to mediator.
Mediator checks s is recent and verifies μ using k_c^M.
if valid **then**
  Forward P securely to broker.
else
  Discard and log violation.
end if

This lightweight MAC-based authentication ensures integrity for non-payload traffic while maintaining compatibility with standard MQTT brokers.

4.5. Dynamic policy and key management

In operational environments, access control policies evolve as clients join, leave, or change permission levels. The scheme supports such evolution through controlled updates to $\sigma()$, selective key regeneration, and publication of updated derivation data. Special labels $\top$, $\bot$, and $\ominus$ (introduced in Section 3.2) simplify representation of *universal access*, *maximum restriction*, and *deactivation*, respectively.

Updating $\sigma()$: Client and topic relabelling. Policy evolution is expressed as updates to $\sigma()$, which assigns each entity a label in $L \cup \{\bot, \top, \ominus\}$. A client's label defines its highest accessible confidentiality level; a topic's label defines the minimum clearance needed to read it. Setting $\sigma(t) = \bot$ makes a topic public, while $\sigma(t) = \top$ restricts access to top-level clients. Assigning $\sigma(t) = \ominus$ disables a topic until republished. Changes to client labels influence both publishing and decryption rights. If the new label n satisfies $\sigma(c) \leqslant n$, it represents an upgrade, requiring only distribution of k_n. Downgrades ($n < \sigma(c)$) or $\sigma(c) = \ominus$ trigger revocation: all keys k_q such that $q \leqslant \sigma(c)$ but $q \not\leqslant n$ must be replaced. Assigning $\sigma(c) = \ominus$ disables a client entirely, blocking all future interactions.

Adding and removing clients and topics. Adding a client assigns $\sigma(c) \in L$, generates pss_c, and provisions keys $\{k_{\sigma(c)}, \bar{k}_{\sigma(c)}, k_c^M\}$ as during setup. Removing a client sets $\sigma(c) \leftarrow \ominus$; the KG may rekey the affected anti-mediator subtree (i.e., the relevant $\bar{k}_l$ values) to restore confidentiality after compromise. New topics are created implicitly upon first publication, inheriting the publisher's label $\sigma(t) \leftarrow \sigma(c)$. Topics can be retired by setting $\sigma(t) \leftarrow \ominus$, and those labelled $\bot$ remain globally readable.

Evolving the label hierarchy. The poset $(L, \leqslant)$ may change over time, for example, by adding or merging levels. Minor changes are handled incrementally: the KG generates new keys, publishes $z_{n,l}$, and updates $\sigma()$. Major restructures require reinitialisation of affected subsets. Special labels $\bot$, $\top$, and $\ominus$ act as fixed boundaries, ensuring that partial updates remain coherent during reconfiguration.

Compromise and rekeying. Because anti-mediator keys are scoped to labels, a compromised client does not automatically expose every message in the system. In particular, if a client at label l is compromised, the attacker can only derive anti-mediator keys for labels $n \leqslant l$, and therefore can only decrypt payloads for topics at or below that clearance. To restore confidentiality after compromise, the KG generates fresh anti-mediator keys $\bar{k}_q$ for the affected labels, publishes updated derivation values $\bar{z}_{p,q}$, and provides updated root keys to clients that remain authorised. This gives a practical rotation mechanism while keeping the recovery work limited to the labels that are actually affected.

Rekeying procedures and overhead. In our scheme, most policy changes are handled by updating $\sigma()$ and then updating only the keys that are affected. If a client is *upgraded* to a higher label, the KG simply provides the client with the new root keys k_n and $\bar{k}_n$ (encrypted under the provisioning key k_c), and the client can derive any lower keys it is allowed to use. If a client is *downgraded* to a lower label, or disabled by setting $\sigma(c) = \ominus$, then the client must no longer be able to decrypt data at labels it previously had access to. In this case, the KG regenerates the relevant label keys k_q and anti-mediator keys $\bar{k}_q$ for each affected label q, publishes fresh derivation values $z_{p,q}$ and $\bar{z}_{p,q}$, and provides updated root keys to the remaining authorised clients. *Topic reset* is handled by setting $\sigma(t) = \ominus$, which removes t from T so that the next authorised first publisher re-creates it and assigns its label as usual. In terms of overhead, the main public state is the derivation data: in the worst case, publishing all $z_{n,l}$ and $\bar{z}_{n,l}$ values is $O(|L|^2)$, but each client stores only its root keys $(k_{\sigma(c)}, \bar{k}_{\sigma(c)})$ and derives other keys on demand. Key updates after downgrade or compromise therefore touch only the affected subset of labels, rather than the entire lattice.

Quantitative overhead of published derivation data. The worst-case $O(|L|^2)$ bound occurs when the label order is close to a total order, so that most label pairs are comparable and therefore require published derivation data. To make this concrete, Table 2 estimates the size of the public derivation data $\{z_{n,l}, \bar{z}_{n,l}\}$ under a worst-case chain of $|L|$ labels. We assume 256-bit values (32 bytes) for each published z or $\bar{z}$, so each comparable pair (n, l) with $n < l$ contributes 64 bytes of public data. In practice, real deployments typically use a much smaller number of labels and a sparser partial order, so the published state is often substantially smaller than this worst-case estimate.

In practice, rekeying after a downgrade or compromise republishes derivation values only for the affected labels, so the update size is typically much smaller than the full worst-case shown in Table 2.

4.6. Deployment considerations

Our scheme is intended to be deployed in the common IoT setting where clients are the constrained devices, while the network edge (gateway) has enough resources to host mediators and a broker. In this deployment, the broker can remain unmodified and focus on routing, while mediators perform publish-time enforcement and payload re-wrapping. To prevent clients from bypassing enforcement, the broker should be configured to accept publish traffic only from authorised mediators (for example using network controls that restrict broker access, or by requiring mutual TLS where the broker accepts PUBLISH connections only from mediator identities). Clients then connect to mediators, and mediators connect onward to the broker. This makes publish enforcement a deployment property, rather than an assumption about client behaviour. The KG can run as a management component on the local edge or in a trusted backend, depending on operational constraints. Client onboarding requires only a per-device pre-shared secret and a one-time secure provisioning step to deliver the client's root keys; afterwards, clients derive additional keys locally from the published derivation values. For resilience, mediators and the KG can be replicated: multiple mediators can share topic/label state (e.g., through a small shared datastore or synchronisation channel), and KG key updates can be rolled out by re-provisioning only the affected labels. Finally, our design does not attempt to hide metadata such as topic names or traffic patterns; if a deployment requires stronger privacy against traffic analysis, additional techniques such as topic-name encryption, padding, or batching can be layered on top. Evaluating these deployment trade-offs (including multi-broker environments and cross-domain routing) is an important direction for future work.

5. Performance experiments

In this section we compare our enforcement scheme to a plain MQTT client with no security and an MQTT client using TLS. Our scheme will naturally be less performant than plain MQTT, but the goal is to provide end-to-end payload protection and policy enforcement with overhead that is competitive with TLS in the same software stack.

5.1. Theoretical performance comparison

Table 3 summarises the dominant cryptographic work performed at the client; it is intended as an operation-level comparison rather than a cycle-accurate cost model. Plain MQTT performs no cryptography. TLS incurs a handshake during connection establishment and then applies record-layer authenticated encryption to each packet. Our scheme avoids client-side public-key handshake work by using symmetric key setup, but it applies two layers of authenticated payload protection (inner anti-mediator layer and outer topic/label layer). As a result, relative performance depends on the workload: connection-heavy scenarios are dominated by handshake/setup costs, while message-heavy scenarios are dominated by how many payload bytes are processed under AEAD.

A TLS client performs the handshake upon initial connection, whenever session keys are refreshed, and potentially at periodic intervals. This process involves resource-intensive asymmetric cryptographic operations that are challenging for constrained devices. For each MQTT packet sent or received, TLS encrypts or decrypts the entire packet, whereas our scheme applies encryption only to the message payload and a MAC over selected headers.

In our design, the key-derivation step on receive is lightweight (hash/XOR) and can be cached: after the first message for a given label, subsequent messages at the same label do not require repeating derivation unless cache entries are evicted.

Table 2
Illustrative size of published derivation data in the worst case (chain order), assuming 32-byte z values and 32-byte $\bar{z}$ values.

$\lvert L\rvert$	Comparable pairs ($n < l$)	Published values ($z, \bar{z}$)	Approx. size
10	45	90	~2.8 KiB
50	1225	2450	~76.6 KiB
100	4950	9900	~309 KiB
500	124,750	249,500	~7.6 MiB
1000	499,500	999,000	~30.5 MiB

Table 3
Comparison of the theoretical processing required by each type of client.

Action	Plain MQTT	MQTT over TLS	Our Scheme
Connect & Setup	No crypto	Handshake & record setup	Symmetric key setup
Receive	No crypto	Decrypt	Decrypt × 2 & key derivation
Publish	No crypto	Encrypt	Encrypt × 2
Other MQTT	No crypto	En/decrypt	MAC Create/Verify

5.2. TLS baselines and fairness

TLS can be configured in several ways that affect performance, especially during connection setup. In our experiments, we use TLS 1.3 with certificate-based authentication as a baseline, since this is widely deployed in MQTT stacks. In constrained deployments, the handshake cost can often be reduced by avoiding certificates (for example by using pre-shared keys) or by reusing a previous session when reconnecting. These options mainly reduce the cost of establishing a connection. They do not remove the need to encrypt and authenticate data packets once the connection is established, so clients still perform symmetric record protection on each MQTT packet and incur TLS record framing overhead. We therefore interpret our results as follows: the connect-stage difference is largest when certificate handshakes are required or occur frequently, while in deployments that avoid certificates or frequently reuse sessions the connect-stage gap will narrow, but the per-message costs remain dominated by symmetric AEAD processing as summarised in Table 3.

To reduce implementation bias, we use the same AEAD primitive for both TLS and our scheme, so differences mainly reflect protocol structure (e.g., handshake and record framing) rather than cipher choice. The specific AEAD we use is stated in Section 5.3.

5.3. Configuration

We built three client configurations for testing: a plain MQTT client, an MQTT-over-TLS client, and a client using our scheme. To focus on client-side overhead, we implemented an *emulated infrastructure* component that plays the role of the mediator, the KG, and the peer endpoints required to drive the test workload. This lets us compare the clients directly under identical conditions, without mixing the client measurements with the performance of the broker or mediator.

Our evaluation focuses on clients because they are the resource-constrained devices in typical IoT deployments and they perform the cryptographic work that dominates the costs we care about (for example, TLS handshake work and per-message encryption/decryption). The experiments therefore cover the main client-side parts of our design in Section 4, including initial key setup (the client receiving its keys from the KG), deriving topic keys from public values, and the two-layer payload protection used when publishing and receiving messages. We analyse mediator and KG overhead separately in Section 5.4.4.

We use consistent software and configuration to make the experiment as fair as possible. We implement the clients and the emulated infrastructure in Java using the HiveMQ client library,[3] chosen for its extensibility and clear documentation. Although Java adds overhead, this affects all client variants in the same way, so our results are best interpreted comparatively. We use Gradle[4] to compile and run the code, as it manages HiveMQ dependencies automatically. The Mosquitto [18] broker[5] is used for its simplicity and ease of configuration.

We run all tests simultaneously inside Docker containers, following the synchronised execution approach of Michaelides et al. [19], to reduce distortion from background processes. Each client type (plain MQTT, TLS, and our scheme) runs in its own network containing three containers: the client, the broker, and the emulated infrastructure. We use Eclipse Temurin[6] Docker images for clients and emulated infrastructure components, and Ubuntu for brokers.

We use the `docker stats`[7] command to record CPU, memory, and network usage for each client container, and then process the data using custom Python scripts with Matplotlib[8] to generate graphs. We limit the maximum CPU available to each container to simulate constrained clients and amplify performance differences.

We use MQTT v5.0 with QoS level 1 throughout the tests. Although we recommend ChaCha20-Poly1305 in practice, the HiveMQ client does not support this cipher suite, so we use AES-256-GCM as the AEAD algorithm for both the TLS client and our client. For TLS, we use TLS 1.3 with standard certificate-based authentication as our baseline configuration. We discuss alternative TLS configurations commonly used in constrained deployments (PSK and session resumption) and how they would affect the connect-stage overhead in Section 5.2.

We use a 1 MiB payload as a stress test to make cryptographic costs visible in the measurements. Many IoT deployments use much smaller MQTT payloads than 1 MiB. Since both TLS and our scheme are dominated by symmetric AEAD for per-message payload protection, the cryptographic cost scales approximately linearly with payload size. We discuss how this affects the interpretation of our results later in this section. Repeating the measurements across a payload-size sweep is left for future work.

Table 4 summarises the key parameters of our experimental setup. We use a minimal label lattice with two levels ($l_1 > l_2$) so that the client must derive a lower-level key when receiving messages on topic t (i.e., $\sigma(c) = l_1$ and $\sigma(t) = l_2$). We perform the following actions: (1) the client connects to the broker; (2) it subscribes to t; (3) it receives a message on t (and in our scheme derives the key for l_2 from public information if needed); and (4) it publishes to a different topic for the first time.

[3] HiveMQ client library: https://hivemq.github.io/hivemq-mqtt-client/

[4] Gradle build tool: https://gradle.org/

[5] Eclipse Mosquitto: https://mosquitto.org/

[6] Eclipse Temurin: https://hub.docker.com/_/eclipse-temurin/

[7] Docker container stats: https://docs.docker.com/reference/cli/docker/container/stats/

[8] Matplotlib: https://matplotlib.org/

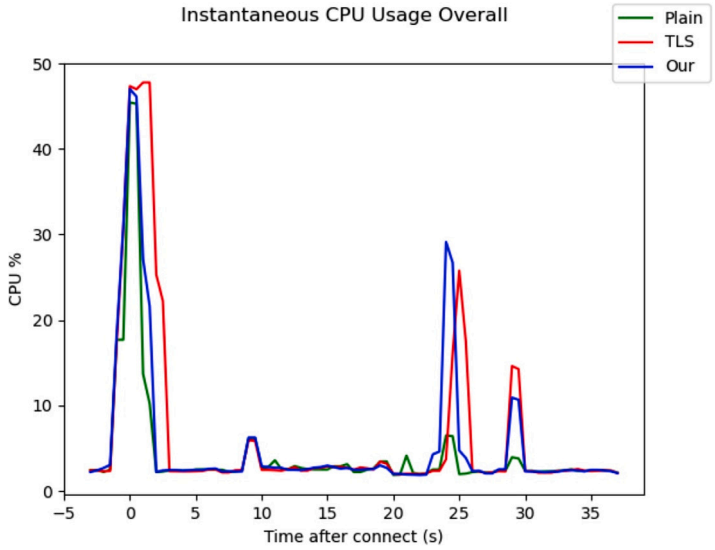

Fig. 4. Instantaneous CPU usage over the full experiment (median of 10 runs). The connection stage includes setup work: TLS performs a handshake, while our scheme performs symmetric key provisioning. The large spikes correspond to (i) connection/setup and (ii) payload processing during receive and publish.

Table 4
Summary of the experimental configuration.

Parameter	Value
MQTT version/QoS	v5.0/QoS 1
Payload size	1 MiB (stress test)
Clients compared	plain MQTT, MQTT over TLS, our scheme
Label set L	$\{l_1, l_2\}$ with $l_1 > l_2$
Client/topic labels	$\sigma(c) = l_1$, $\sigma(t) = l_2$
Initial topic set T	$\{t\}$

Because this evaluation is implemented in Java and executed in Docker containers, the absolute CPU and memory values include JVM and container overhead. We therefore interpret the results as a controlled, comparative evaluation within a single software stack, rather than as absolute benchmarks for microcontroller-class devices.

5.4. Results

We observed some run-to-run variation in CPU traces, including occasional unexplained spikes that are likely caused by host-level background activity and JVM/container effects. To reduce the impact of such noise, we repeated the experiment ten times and report the median value at each time step in Fig. 4–Fig. 7. We also summarise per-stage CPU totals in Table 5. Since our measurements are obtained via `docker stats` at a coarse sampling rate, we do not report fine-grained confidence intervals for the instantaneous traces.

We divided the experiment timeline into distinct stages so that each action could be analysed individually. Time is measured in seconds from the moment the clients simultaneously connect to the broker. The clients subscribe at 10 s, prepare to receive a message at 20 s, receive a message at 25 s, and publish at 30 s. Since time 0 corresponds to the peak connection activity, each stage begins slightly before its nominal start time. For example, Fig. 4 shows CPU usage over the entire test, while Fig. 5 presents the same results separated into individual plots for each stage with adjusted y-axes for clarity.

Note that, in addition to sending a CONNECT packet and receiving a CONNACK packet, the connection stage also includes setup operations: the TLS client performs the handshake, while our client receives its encrypted keys. Consequently, this stage results in significantly higher resource usage by the TLS client compared to the others.

Table 5
Percentage difference in CPU usage during each stage and overall.

Comparison	Connect	Subscribe	Receive	Publish	Overall
TLS > Plain	68%	−6%	66%	41%	54%
Our > Plain	22%	−2%	86%	28%	29%
TLS > Our	38%	−4%	−11%	10%	20%

5.4.1. CPU usage

Figs. 4 and 5 show that TLS has the highest CPU cost during the connect stage, where the TLS handshake dominates processing. In our measured setup, TLS remains elevated for longer during connection establishment, while plain MQTT and our scheme return more quickly to the background level.

During the subscribe stage, all three clients show similar CPU usage, which is expected because subscribing does not involve heavy payload processing. During the receive and publish stages, the dominant cost is symmetric cryptography over the large payload. Our scheme performs two layers of authenticated encryption/decryption on the payload, while TLS applies record protection to the packet. As a result, our client shows the larger spike during receive in this configuration, while TLS shows the larger spike during publish. Table 5 summarises these differences over each stage.

If MQTT payloads are smaller, the time spent on fixed work such as packet handling and the Java/Docker runtime becomes a larger part of the total cost. The cryptographic work in both TLS and our scheme still grows roughly in proportion to the number of payload bytes processed, so the CPU differences between the schemes are expected to shrink for very small messages. In that case, the connect stage (TLS handshake versus our key setup) is likely to matter more relative to the per-message cost.

5.4.2. Memory usage

Fig. 6 shows that the TLS client consistently uses the most memory throughout the experiment. After the connection stage, TLS stabilises at roughly the high-430 MiB range, while the plain client stabilises around the low- to mid-400 MiB range. Our scheme remains closer to the plain client: it sits a few MiB above plain for most of the run and shows a similar step increase around the receive stage (approximately 24–25 s), where both plain and our scheme rise by around 10–15 MiB.

This behaviour is consistent with the fact that TLS maintains additional connection and record-layer state, while our scheme relies

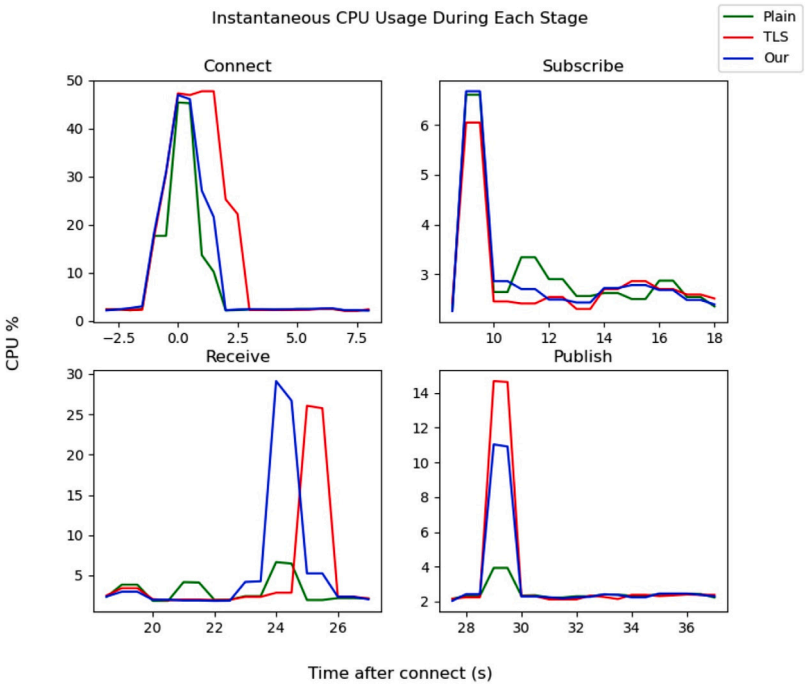

Fig. 5. Instantaneous CPU usage separated by stage (median of 10 runs). TLS dominates during connect due to the handshake. During steady-state messaging, our scheme shows a higher receive spike (outer decrypt + key derivation if uncached + inner decrypt) while TLS shows a higher publish spike in this configuration.

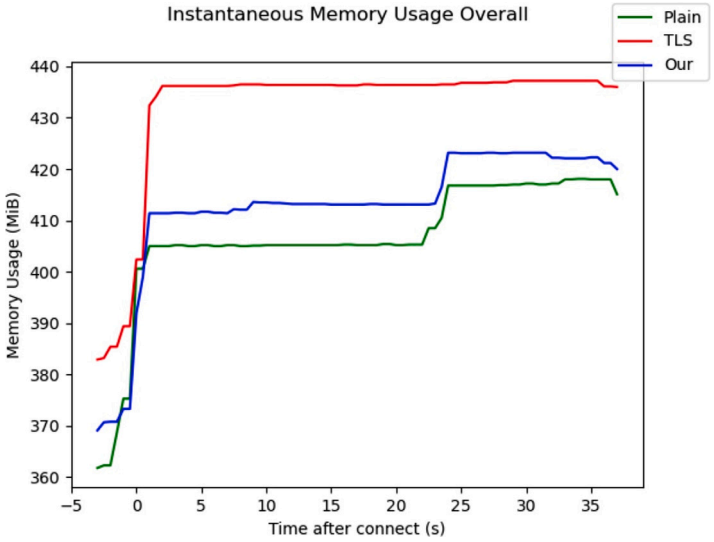

Fig. 6. Instantaneous memory usage over the full experiment (median of 10 runs). The absolute values are influenced by the Java runtime and container environment, so the figure is best interpreted comparatively: TLS consistently uses more memory than plain MQTT, while our scheme remains closer to plain MQTT.

mainly on symmetric keys and payload processing. In our design, the main extra memory cost at the client is key material (root keys and any cached derived keys) and buffers for encrypting/decrypting payloads, which is smaller than the additional state maintained by TLS in this software stack. Because these measurements are taken in a Java/Docker environment, the absolute values are not representative of embedded device footprints; the key observation is the relative ordering: TLS uses noticeably more memory than the other two clients, and our scheme remains much closer to plain MQTT.

5.4.3. Network usage

Fig. 7 shows that the publish stage dominates total transmitted data, as expected for a 1 MiB payload: all three clients send approximately 1200 kB and the curves are almost indistinguishable at that scale. Differences are clearer in stages where payloads are small. During connect, TLS sends the most data (close to 1 kB), our scheme is intermediate (around 0.7 kB), and plain MQTT sends the least (around 0.3 kB), which is consistent with TLS handshake traffic and with our scheme adding additional metadata during setup. Similar small constant differences appear during subscribe.

During receive, TLS again shows the largest overhead (close to 5 kB in this run), while our scheme and plain MQTT remain closer (around 3 kB and 2.7 kB respectively). Overall, these results suggest that both TLS and our scheme add mostly fixed-size protocol overhead for small messages, whereas during large publish operations the payload

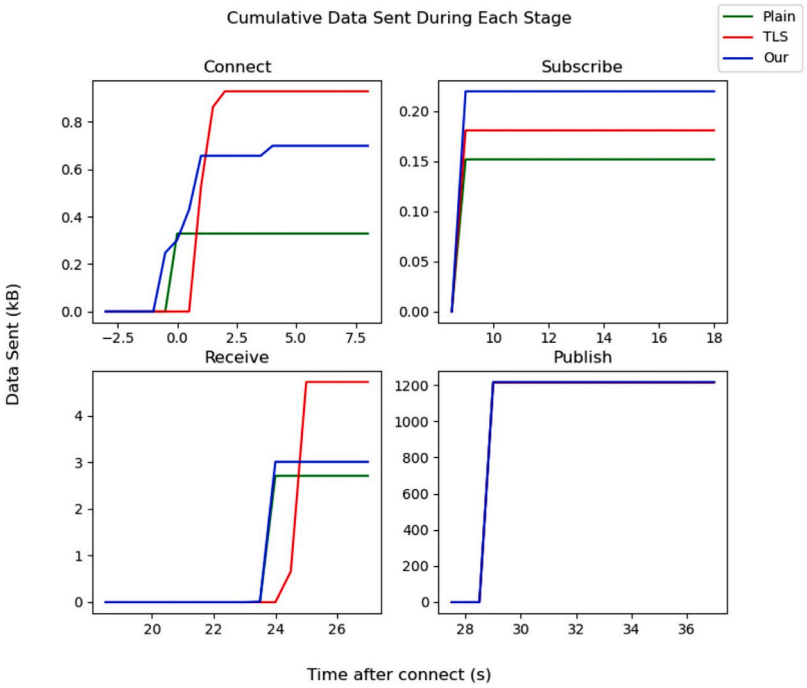

Fig. 7. Cumulative data sent by the client during each stage. Differences during connect and subscribe reflect protocol overheads, but they are small compared to the publish stage, where the 1 MiB payload dominates overall network usage.

dominates and network usage is essentially the same across all three clients.

5.4.4. Discussion

Overall, the results suggest that our scheme can reduce connect-stage overhead relative to TLS in this measured setup, while remaining comparable in steady-state messaging under this stress-test payload. Naturally, our scheme incurs slightly higher overhead than plain MQTT, but the difference is not significant. The only stage in which our client demands more processing than the TLS client is during message reception. However, since constrained clients typically send far more messages than they receive, this trade-off is acceptable.

Although we made every effort to ensure fair comparisons, certain limitations remain. For instance, TLS can be configured in various ways using different cipher suites, some of which employ pre-shared keys instead of X.509 certificates, potentially yielding different results. We selected configurations that best reflect common real-world deployments while using the same symmetric authenticated encryption algorithm as our scheme for consistency.

Performance data collected via Docker stats may not be perfectly accurate or precise, as it records values roughly twice per second, potentially leading to spikes being missed, particularly of CPU usage. However, repeating the experiment and using the median at each time step helps mitigate this limitation.

Future work will extend this evaluation to a payload-size sweep representative of typical IoT messaging (tens of bytes to a few kilobytes) and to explicit TLS 1.3 baselines using PSK and session resumption, so that the connect-stage comparison can be quantified under those configurations.

Our experiments focus on clients because they are usually the constrained devices in MQTT deployments, and most of the cryptographic cost in our scheme is paid at the client. For completeness, we also analyse the work performed by mediators and the KG. A mediator verifies each client packet using k_c^M and, for PUBLISH packets, performs one additional payload AEAD operation to add the outer topic layer; this work grows linearly with the number of forwarded messages and can be spread across multiple mediators. The KG performs setup-time key creation and publishes derivation values; in the worst case, publishing $z_{n,l}$ and $\bar{z}_{n,l}$ for all pairs with $n \leqslant l$ requires $O(|L|^2)$ public values, while each client stores only its assigned root keys (constant in $|L|$) and derives other keys on demand. After compromise or policy change, rekeying touches only the part of the label poset that must be refreshed, rather than the entire lattice. We will include a full system-level benchmark of mediator throughput and KG update cost in a future implementation.

In addition to CPU and memory, energy consumption is often the key constraint for battery-powered clients. While we do not instrument power directly in this prototype, the dominant energy cost of cryptography on constrained nodes typically follows the number of symmetric AEAD operations and the amount of data processed. Our scheme performs two payload AEAD operations at the endpoints (inner anti-mediator layer and outer topic layer), whereas TLS performs one record-layer AEAD operation plus record framing on each packet and may also incur additional handshake energy during connection setup. The analytic message-size discussion in this section therefore also serves as an energy proxy: if CPU time scales roughly linearly with bytes processed, relative energy cost will follow the same trend.

Finally, while the overhead of our Java-based implementation does not obscure the overall trends, it limits the precision of percentage-based comparisons. A lower-level implementation (e.g., in C or Rust) would provide more accurate insight into the scheme's true efficiency.

6. Related work

There has been substantial work on IoT security in general [20], including architectures and policy models that emphasise strong device identity and least-privilege authorisation. For example, Ameer et al. [21] propose a Zero-Trust IoT architecture (ZTA-IoT) together with an access-control framework that enumerates the different interactions that must be governed across an IoT ecosystem, and they develop a usage-control policy model (UCONIoT) with a proof-of-concept implementation. Similarly, recent work on privacy-preserving data management for IoT considers lightweight access control in encrypted settings; for instance, Spidey [22] supports secure queries and updates over encrypted IoT property graphs and adds lightweight

multiuser access control (including role hierarchy) without relying on heavy attribute-based encryption. These lines of work provide important general guidance and building blocks, but they do not address MQTT-specific challenges such as dynamic topic creation and brokered message confidentiality, which are the focus of this paper.

There has also been substantial work specifically on the security of MQTT [23], including authentication enhancements, access-control enforcement, and end-to-end payload protection schemes. Our contribution is scoped to MQTT: we focus on (i) access control that evolves automatically with MQTT's dynamic topic tree, and (ii) lightweight end-to-end payload protection that does not rely on trusting the broker (or mediator) with plaintext. In the remainder of this section, we therefore review the closest MQTT-focused schemes that influence our design and provide the most direct comparison points.

Berlato et al. [5] use a payload encryption scheme to enforce a Role-Based Access Control (RBAC) policy model for MQTT. Like our information flow policy model, RBAC is scalable to many types of clients and flexible in defining policies; however, its policy model cannot naturally adapt to a changing topic hierarchy, which is one of the key features of the MQTT protocol. The RBAC enforcement scheme uses asymmetric encryption for digital signatures and for accessing symmetric keys, and it requires clients to authenticate with the MQTT broker. This introduces significant additional computational overhead compared with our scheme. The performance experiments measure only transmission time, indicating that their scheme is slower than TLS for sending messages (excluding setup and the TLS handshake), making it unsuitable for constrained clients.

Colombo et al. [3] present an Attribute-Based Access Control (ABAC) policy model and its enforcement scheme for a distributed MQTT environment, where clients and brokers are connected across different domains and networks. ABAC is chosen for its flexibility and ability to define fine-grained policies. However, as the number of clients and topics grows, the "unlimited" number of attributes associated with each client, topic, and environmental context becomes unmanageable. In contrast, by assigning only a single label to each client and topic, our policy model scales much better as the system grows and supports the dynamic nature of MQTT topics, which their approach cannot. On the policy enforcement side, they introduce an entity called *monitors*, which serve a role similar to our mediators but perform significantly more tasks. For example, to obtain attribute values, monitors must parse each packet, extract relevant information, and add additional data to the payload so it can be retrieved by other monitors. Moreover, user-defined policies must be synchronised to enable consistent enforcement across all packets. This introduces unnecessary complexity for monitors to enforce policies across a network of brokers. Their performance evaluation, similar to [5], focuses on transmission time and throughput. In contrast, our experiments focus on the processing effort required by constrained clients, specifically to assess the suitability of our scheme for practical IoT environments.

Chen et al. [24] present a trust management approach for authentication and authorisation in a distributed network of MQTT brokers. They argued that policy-based approaches, such as RBAC and ABAC, are unlikely to scale well in a multi-broker network. Instead, they proposed a trust-based approach for controlling message flow across the networked brokers. However, their work remains preliminary at a conceptual level, with no explicit scheme or experiments to demonstrate its practical use. In contrast, our policy model has been shown to be simple and flexible, but it is worth exploring how it can be extended to a network of multiple brokers in the near future.

Hamad et al. [25] propose SEEMQTT, an end-to-end secure MQTT framework for mobile IoT in which the publisher encrypts messages with a symmetric key and uses secret sharing across multiple honest-but-curious KeyStores plus trust delegation to control which subscribers can reconstruct the key and decrypt messages. Compared with our approach, SEEMQTT achieves E2E confidentiality and publisher-controlled authorisation but relies on external KeyStores and IBE-based links, and it does not address automatic policy evolution with a dynamic MQTT topic hierarchy.

Al-Fayoumi and Al-Haija [26] study MQTT security from an availability perspective and propose a lightweight machine-learning IDS for detecting low-rate DDoS attacks in a software-defined IoT setting. This line of work is complementary to ours: it targets attack detection and resilience, rather than publish/subscribe authorisation or end-to-end payload confidentiality.

Aljund et al. [27] propose an authentication framework for MQTT-IoT that combines PUF-style challenge–response, AI-driven anomaly detection, and multi-factor authentication during connection establishment, and they report formal verification using AVISPA. Their approach strengthens client identity assurance but relies on enhanced broker-side verification and does not address topic-level access control or end-to-end payload confidentiality in the brokered path.

Several studies on encryption schemes for MQTT [9–11] exist, but they do not consider access control policies or their enforcement, thereby allowing all clients to publish and subscribe to all topics. Instead, they focus on developing lightweight end-to-end payload encryption schemes for MQTT, based on the observation that TLS is too resource-intensive for such environments.

Fan et al. [9] use asymmetric cryptography to distribute symmetric keys used for message encryption. The publisher must generate and encrypt a unique symmetric key for each message before encrypting the message itself, which places a heavy computational burden on publishers. Although their performance experiments show that the scheme's setup stage is over 800 times faster than the TLS handshake, decryption is almost 300 times slower and encryption is more than 20 times slower. Since encryption and decryption occur far more frequently than setup, the scheme would perform significantly worse than TLS in realistic scenarios.

Like our scheme, Bashir and Mir [10] use a key generator and pre-shared secrets to avoid the asymmetric cryptography. Their description of encryption is very vague (only suggesting XOR) and does not mention integrity, although MACs could trivially be used to verify message integrity. However, their scheme faces the same problem as ours: if it relies solely on symmetric cryptography, any device capable of receiving a message can modify that message before forwarding it to others, thereby distorting what they receive. On the other hand, if asymmetric cryptography is used, the scheme is likely to be too resource-intensive for constrained clients. They also conduct a performance analysis, but only in terms of energy consumption and storage requirements, not processor or memory usage. Moreover, their results are compared only to plain MQTT, making it difficult to determine whether the performance is acceptable.

Sadio et al. [11] present performance experiments demonstrating that the ChaCha20-Poly1305 AEAD is suitable for constrained MQTT devices. Therefore, we recommend using this in our scheme (although it is primitive-independent). However, they do not provide any mechanism for distributing keys between clients.

Table 6 summarises how closely existing approaches match the combination of properties targeted in this paper. The encryption-only schemes [9–11] provide end-to-end payload protection but do not enforce access control. Policy-based approaches such as RBAC/ABAC [3, 5] and SEEMQTT [25] enforce authorisation, but they typically rely on heavier mechanisms (e.g., asymmetric operations, external infrastructure, or more complex monitoring) and/or do not provide end-to-end payload confidentiality in the brokered path. The remaining MQTT security work is complementary: Al-Fayoumi and Al-Haija [26] addresses low-rate DDoS detection and Aljund et al. [27] strengthens client–broker authentication, but neither provides topic-level authorisation or automatic policy evolution for dynamic topics. In addition, existing MQTT access-control schemes typically assume a largely static topic set or require manual policy updates when new topics appear, whereas our policy model assigns a label when a topic is created so that access control evolves automatically as the MQTT topic tree grows.

Table 6
Comparison of the features of existing schemes with ours.

Author(s)	Resource intensity	Broker unchanged	E2E encryption	Access control	Dynamic topics
Our Work	Good	Yes	Yes	Yes	Yes
Berlato et al. [5]	Medium	No	Yes	Yes	No
Colombo et al. [3]	Poor	Yes	No	Yes	No
Hamad et al. [25]	Medium	Yes	Yes	Yes	No
Fan et al. [9]	Medium	No	Yes	No	No
Bashir and Mir [10]	Good	Yes	Yes	No	No
Sadio et al. [11]	Good	Yes	Yes	No	No
Al-Fayoumi and Al-Haija [26]	Medium	Yes	N/A	No	N/A
Aljund et al. [27]	Medium	No	N/A	No	N/A

7. Conclusion

In this paper, we have presented a novel access control policy model and enforcement scheme for MQTT that address the security and performance challenges faced by constrained IoT clients. To the best of our knowledge, existing MQTT access-control approaches do not automatically evolve with the topic hierarchy in the way our model does. Our model is designed to be easy to define, scalable to large numbers of clients, and expressive enough to represent common MQTT use cases.

We also proposed an enforcement scheme that simultaneously enforces the policy and protects message payloads end-to-end, while remaining lightweight enough for constrained devices in our target deployment setting. Our scheme ensures confidentiality and integrity through end-to-end encryption, operates with unmodified MQTT brokers, and is cryptographic primitive agnostic. In practice, our scheme provides a clear improvement over having no message protection for constrained clients that cannot support TLS. While TLS remains preferable when feasible, our solution offers a practical balance between security and efficiency in resource-limited IoT environments.

There are several promising directions for future work. First, we could investigate the ramifications of allowing a single physical client to act as multiple logical clients and examine whether such behaviour (i.e., collusion) could violate the policy model. Second, we aim to adapt and evaluate our approach for other common IoT communication protocols with similar characteristics. We also plan to extend and evaluate our approach in multi-broker MQTT deployments, where messages and policy information may need to be enforced consistently across brokers and domains. Finally, we intend to fully implement our scheme, including measuring the performance of mediators and the key generator, to enable integration with commercial (e.g., HiveMQ) or open-source projects (e.g., Mosquitto) for refinement and evaluation in real-world IoT environments.

CRediT authorship contribution statement

Liang Chen: Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **James Wilson:** Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Security model and proof sketches

This appendix states the security goals from Section 4.2 in a more formal, game-based way, and gives short proof sketches under standard assumptions for the primitives used in our scheme. The goal is not to introduce new mechanisms, but to make the scope of our guarantees precise.

Notation and assumptions. For a topic t with label $\sigma(t)$, the payload delivered by the broker has two protection layers: an outer layer under the topic/label key $k_{\sigma(t)}$ and an inner "anti-mediator" layer under $\bar{k}_{\sigma(t)}$ (Section 4.3–4.4). We assume the outer and inner payload layers use an AEAD scheme $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$ with standard confidentiality and integrity (IND-CPA/IND-CCA style confidentiality and INT-CTXT ciphertext integrity). We also assume the MAC used on headers is unforgeable under chosen-message attack. Finally, we assume the hash function H used in the key-assignment derivation values is preimage-resistant, so that it is infeasible to recover k_l (or $\bar{k}_l$) from public values without already holding an authorised ancestor key.

Security goals as games. We model confidentiality for a topic t by the following indistinguishability game. An adversary $\mathcal{A}$ chooses two equal-length payloads (m_0, m_1) for t. A challenger samples a bit $b \in \{0,1\}$ and returns a broker-delivered ciphertext constructed exactly as in Algorithm 2 (including both payload layers and any associated data). The adversary outputs a guess b'. We say the scheme provides end-to-end confidentiality if for any efficient $\mathcal{A}$ that does *not* hold an authorised key for $\sigma(t)$ (i.e., it cannot obtain $k_{\sigma(t)}$ and $\bar{k}_{\sigma(t)}$ via provisioning/derivation), the advantage $\left|\Pr[b' = b] - \frac{1}{2}\right|$ is negligible. This captures the intended threat model where brokers and mediators may be honest-but-curious or compromised, but are not provisioned with anti-mediator keys. We model integrity as ciphertext authenticity for the payload layers and unforgeability for the header MACs. Informally, an adversary that can inject/modify/replay packets should not be able to produce a broker-delivered message for topic t that a subscriber accepts (passes timestamp checks and verifies μ_2) unless it was produced by some authorised publisher and processed by a mediator according to Algorithm 2. As discussed in Section 4.3, this notion is *topic/label level* authenticity (not non-repudiation): if multiple publishers are authorised at the same label and share $k_{\sigma(t)}$, then any of them can produce a valid tag for that topic. If a client with label l is compromised, the attacker learns that client's provisioned roots $(k_l, \bar{k}_l)$ and may derive keys only for labels $n \leqslant l$ using the published derivation values. The attacker should not be able to derive any key for a label h with $h \not\leqslant l$ (in particular, any higher confidentiality label), so decryption capability is limited to the part of the label poset dominated by l.

Proof sketches. Consider any broker/mediator/network adversary that does not hold $\bar{k}_{\sigma(t)}$. Even if it obtains (or guesses) the outer key $k_{\sigma(t)}$, decrypting the outer layer yields only the inner ciphertext $m_1 = \mathsf{Enc}(\bar{k}_{\sigma(t)}, m)$. By AEAD confidentiality, m_1 reveals no information about m without $\bar{k}_{\sigma(t)}$. Therefore, the plaintext remains hidden from brokers and mediators, matching the intended "end-to-end payload confidentiality" goal in Section 4.2. A subscriber accepts a message for topic t only if it can (i) obtain $k_{\sigma(t)}$ to verify μ_2 and decrypt m_3, and (ii) obtain $\bar{k}_{\sigma(t)}$ to decrypt m_1 (Algorithm 3). Because both $k_{\sigma(t)}$ and $\bar{k}_{\sigma(t)}$ are derivable only when $\sigma(t) \leqslant \sigma(c')$ and the subscriber holds the corresponding root keys, an unauthorised subscriber cannot decrypt even if it subscribes to the topic and receives ciphertext from the broker. Any modification to m_3, $\sigma(t)$, or t causes μ_2 verification to fail except with negligible probability (MAC unforgeability). Any modification to the inner ciphertext m_1 is detected when AEAD decryption under $\bar{k}_{\sigma(t)}$ fails.

Table A.7
Summary of compromise impact. Confidentiality is designed not to rely on brokers or mediators, but the KG remains a trusted root.

Compromised component	What the attacker learns	Payload confidentiality	Tamper detection	Enforcement and availability
Broker	Ciphertexts and metadata (topics, timing, volume)	Holds (ciphertext only); cannot decrypt payloads	Unaffected at subscribers	May drop/delay messages
Mediator	k_c^M for served clients; policy state; ciphertexts	Holds if $\bar{k}$ keys are not leaked	May inject/drop; subscribers detect payload tampering	Can violate enforcement or DoS
Client at label l	$(k_l, \bar{k}_l)$ and derived keys for $n \leqslant l$	Breaks confidentiality for topics with labels $\leqslant l$	Attacker can forge at that label	Client can misbehave at its label
Key Generator (KG)	All keys and labels	Not protected	Not protected	Not protected

Timestamp checks prevent straightforward replay of old packets outside the freshness window. Together, this provides the "tamper detection and data-origin authentication at the topic/label level" described in Section 4.2–4.3. If an attacker compromises a client at label l, it learns $(k_l, \bar{k}_l)$ and can derive subordinate keys for $n \leqslant l$ using the public values $(z_{n,l}, \bar{z}_{n,l})$. Under the assumption that H prevents reverse derivation, the attacker cannot derive any key for a label h that it was not authorised to read (i.e., $h \not\leqslant l$). Therefore compromise impact is limited to the compromised client's clearance rather than the entire system.

Compromise summary. Table A.7 summarises the compromise cases and the resulting security impact. The table highlights the intended trust boundary: brokers and mediators are not trusted for payload secrecy, while the KG is trusted. Mediator compromise can affect enforcement and availability, but it does not reveal plaintext unless anti-mediator keys are also leaked.

Data availability

No data was used for the research described in the article.

References

[1] Ravidas S, Lekidis A, Paci F, Zannone N. Access control in internet-of-things: A survey. J Netw Comput Appl 2019;144:79–101. http://dx.doi.org/10.1016/J.JNCA.2019.06.017.

[2] Vailshery LS. Number of internet of things (IoT) connected devices worldwide from 2019 to 2034, by vertical. Technical repor, Statista Ltd.; 2025, Available from: https://www.statista.com/statistics/1194682/iot-connected-devices-vertically/.

[3] Colombo P, Ferrari E, Tümer ED. Regulating data sharing across MQTT environments. J Netw Comput Appl 2021;174:102907. http://dx.doi.org/10.1016/J.JNCA.2020.102907.

[4] OASIS Standard. MQTT Version 5.0. 2019, http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[5] Berlato S, Morelli U, Carbone R, Ranise S. End-to-end protection of IoT communications through cryptographic enforcement of access control policies. In: Proceedings of the 36th annual IFIP WG 11.3 conference on data and applications security and privacy. vol. 13383, 2022, p. 236–55. http://dx.doi.org/10.1007/978-3-031-10684-2_14.

[6] Sasi T, Lashkari AH, Lu R, Xiong P, Iqbal S. A comprehensive survey on IoT attacks: Taxonomy, detection mechanisms and challenges. J Inf Intell 2024;2(6):455–513. http://dx.doi.org/10.1016/j.jiixd.2023.12.001.

[7] Rescorla E. The Transport Layer Security (TLS) protocol version 1.3. 2018, http://dx.doi.org/10.17487/RFC8446, RFC 8446.

[8] Heer T, Morchon OG, Hummen R, Keoh SL, Kumar SS, Wehrle K. Security challenges in the IP-based internet of things. Wirel Pers Commun 2011;61(3):527–42. http://dx.doi.org/10.1007/S11277-011-0385-5.

[9] Fan C, Shie C, Tseng Y, Huang H. An efficient data protection scheme based on hierarchical ID-based encryption for MQTT. ACM Trans Sens Netw 2023;19(3):61:1–21. http://dx.doi.org/10.1145/3570506.

[10] Bashir A, Mir AH. Lightweight secure-MQTT for internet of things. In: Proceedings of the international conference on optical and wireless technologies. 2020, p. 57–66. http://dx.doi.org/10.1007/978-981-15-2926-9_7.

[11] Sadio O, Ngom I, Lishou C. Lightweight security scheme for MQTT/MQTT-SN protocol. In: Proceedings of the sixth international conference on internet of things: systems, management and security. 2019, p. 119–23. http://dx.doi.org/10.1109/IOTSMS48152.2019.8939177.

[12] HiveMQ. MQTT Essentials: The Ultimate Guide to the MQTT Protocol for IoT Messaging. 2020, https://www.hivemq.com/download-mqtt-ebook/.

[13] Eclipse Foundation. 2021 IoT & Edge Developer Survey Report. 2021, https://outreach.eclipse.foundation/iot-edge-developer-2021.

[14] Bell D, LaPadula L. Secure computer system: Unified exposition and multics interpretation. Technical Report, MITRE Corporation; 1976, https://apps.dtic.mil/sti/citations/tr/ADA023588.

[15] Biba K. Integrity considerations for secure computer systems. Technical Report, MITRE Corporation; 1975, https://seclab.cs.ucdavis.edu/projects/history/papers/biba75.pdf.

[16] Nir Y, Langley A. ChaCha20 and Poly1305 for IETF Protocols. 2018, http://dx.doi.org/10.17487/RFC8446, RFC 8439.

[17] Crampton J, Martin KM, Wild PR. On key assignment for hierarchical access control. In: Proceedings of the 19th IEEE computer security foundations workshop. 2006, p. 98–111. http://dx.doi.org/10.1109/CSFW.2006.20.

[18] Light RA. Mosquitto: server and client implementation of the MQTT protocol. J Open Source Softw 2017;2(13):265. http://dx.doi.org/10.21105/joss.00265.

[19] Michaelides M, Sengul C, Patras P. An Experimental Evaluation of MQTT Authentication and Authorization in IoT. In: Proceedings of the 15th ACM workshop on wireless network testbeds, experimental evaluation & cHaracterization. 2021, p. 69–76. http://dx.doi.org/10.1145/3477086.3480838.

[20] Mosenia A, Jha NK. A comprehensive study of security of internet-of-things. IEEE Trans Emerg Top Comput 2017;5(4):586–602. http://dx.doi.org/10.1109/TETC.2016.2606384.

[21] Ameer S, Praharaj L, Sandhu R, Bhatt S, Gupta M. ZTA-IoT: A novel architecture for zero-trust in IoT systems and an ensuing usage control model. ACM Trans Priv Secur 2024;27(3):22:1–36. http://dx.doi.org/10.1145/3671147.

[22] Wu Y, Wang J, Xu D, Zhou Y. Spidey: Secure dynamic encrypted property graph search with lightweight access control. IEEE Internet Things J 2025;12(7):8095–109. http://dx.doi.org/10.1109/JIOT.2024.3502220.

[23] Laghari S, Li W, Manickam S, Nanda P, Al-Ani AKI, Karuppayah S. Securing MQTT ecosystem: Exploring vulnerabilities, mitigations, and future trajectories. IEEE Access 2024;12:139273–89. http://dx.doi.org/10.1109/ACCESS.2024.3412030.

[24] Chen L, Vidalis S, Yang S. A trust-based approach for data sharing in the MQTT environment. In: Proceedings of the 20th annual international conference on privacy, security and trust. 2023, p. 1–5. http://dx.doi.org/10.1109/PST58708.2023.10320147.

[25] Hamad M, Finkenzeller A, Liu H, Lauinger J, Prevelakis V, Steinhorst S. SEEMQTT: secure end-to-end MQTT-based communication for mobile IoT systems using secret sharing and trust delegation. IEEE Internet Things J 2023;10(4):3384–406. http://dx.doi.org/10.1109/JIOT.2022.3221857.

[26] Al-Fayoumi M, Al-Haija QA. Capturing low-rate DDoS attack based on MQTT protocol in software defined-IoT environment. Array 2023;19:100316. http://dx.doi.org/10.1016/J.ARRAY.2023.100316.

[27] Aljund IM, Al-Haija QA, Al-Dalaien M. AI-Driven PUF-Based Authentication for Secure MQTT-IoT Systems. In: Proceedings of the 7th international conference on intelligent autonomous systems. 2025, p. 169–74. http://dx.doi.org/10.1109/ICoIAS69065.2025.00032.