# End-to-End Protection of IoT Communications Through Cryptographic Enforcement of Access Control Policies (Extended)

Stefano Berlato$^{1,3[0000-0002-1700-672X]}$, Umberto Morelli$^{3[0000-0003-2899-2227]}$, Roberto Carbone$^{3[0000-0003-2853-4269]}$, and Silvio Ranise$^{2,3[0000-0001-7269-9285]}$

[1] Department of Informatics, Bioengineering, Robotics and Systems Engineering,
University of Genoa, Genoa, Italy
[2] Department of Mathematics, University of Trento, Trento, Italy
[3] Security and Trust Research Unit, Fondazione Bruno Kessler, Trento, Italy
{sberlato,umorelli,carbone,ranise}@fbk.eu

**Abstract.** It is crucial to ensure the security and privacy of communications in Internet of Things (IoT) scenarios that process an increasingly large amount of sensitive data. In this context, we propose a cryptographic enforcement mechanism of access control policies to guarantee the confidentiality and integrity of messages exchanged with the MQTT protocol in presence of external attackers, malicious insiders and "honest-but-curious" service providers. A preliminary performance evaluation with a prototype implementation in an open-source tool shows the overhead is acceptable in relevant use case scenarios and provides a higher level of security with respect to other approaches.

**Keywords:** Cryptographic Access Control · Internet of Things · End-to-end Protection · MQTT

## 1 Introduction

The capillary diffusion of Internet of Things (IoT) devices holds the potential to improve the well-being of society in several scenarios, like eHealth and smart cities. The undeniable benefits offered by IoT-based scenarios should be coupled with their security, though. Indeed, the environments in which these scenarios are deployed are traditionally assumed to be hostile due to the presence of external attackers. Moreover, being often unattended and equipped with limited computational resources, IoT devices are intrinsically vulnerable and exposed to high levels of risk. Hence, suitable security mechanisms should be adopted to ensure the protection of sensitive data (e.g., personal or confidential information) throughout their life cycle, i.e., when in-transit, in-use and at-rest. In particular, we note that IoT-based scenarios are especially focused on the transmission of data, which is one of the fundamental layers of their architecture [20]. In this context, communication security and data encryption are the two top concerns for IoT scenarios, as clearly shown by the 2021 Eclipse IoT survey.[4] However, since the traditional client-server network paradigm does not properly fit the needs and peculiarities of IoT (e.g., limited computational and communication capabilities, unreliable channels, latency requirements), these scenarios usually employ more lightweight and efficient *publish-subscribe* protocols such as Message Queue Telemetry Transport (MQTT) [18].

Furthermore, we note that the complexity and dynamicity of IoT-based scenarios (considering also the latest trends in security such as Zero Trust) make it almost impossible to assume full trust on any agent involved. Instead, the agents operating in these scenarios are usually assumed to be untrusted or partially-trusted, where "partially-trusted" (or "honest-but-curious") denotes an agent which faithfully performs the assigned tasks but, at the same time, tries to access sensitive data, usually for profit [12, 7]. In other words, besides being threatened by a plethora of external attackers, sensitive data in IoT-based scenarios must be secured from malicious insiders (e.g., disgruntled employees, harmful tenants) and honest-but-curious service providers as well (e.g., Cloud, Edge).

When no fully-trusted central entity is present, a decentralized approach has to be adopted to protect sensitive data. In this regard, the use of cryptography is fundamental to mitigate or prevent possible attacks on the confidentiality and integrity of data. Indeed, these two security properties are of the utmost importance, especially when considering scenarios involving personal information (e.g., users' health data) or providing vital services in which the integrity of the data is crucial (e.g., smoke sensors). A popular cryptographic-based solution to protect communications by guaranteeing confidentiality and integrity is the use of Transport Layer Security (TLS). However, the adoption of TLS may be difficult in presence of constrained IoT devices that cannot support cumbersome handshakes and computationally expensive key derivation algorithms [18, 13]. Moreover, TLS offers hop-to-hop protection (i.e., information is only encrypted when travelling through the network), thus it cannot protect sensitive data against partially-trusted agents.

In this paper, we address these issues by proposing a solution for the end-to-end protection of IoT communications through the cryptographic enforcement of Access Control (AC) policies. In detail, our contributions are as follows:

- we design a Cryptographic Access Control (CAC) scheme enforcing AC policies in IoT scenarios to prevent external attackers, malicious insiders and partially-trusted agents from breaching the confidentiality and the integrity of sensitive data;
- we implement the proposed CAC scheme in a modular and portable tool, which we make open-source and freely available;[5]
- we conduct a preliminary experimental evaluation to analyze the performance of our tool and investigate the (possible) overhead with respect to both the approach proposed in [9] and a traditional TLS-based solution.

[4] https://outreach.eclipse.foundation/iot-edge-developer-2021

[5] https://github.com/stfbk/CryptoAC

As a final remark, we acknowledge that the use of cryptography alone to enforce AC policies makes the evaluation of permissions depending on dynamic and contextual (e.g., time-based) conditions difficult, if possible at all. The limited expressiveness of CAC can however be mitigated through the combination with more traditional (e.g., centralized) AC enforcement mechanisms, at the cost of addressing possible collusions between users and the (agents managing the) enforcement mechanisms. In other words, rather than supplanting existing approaches to AC like [9], CAC can complement and synergize with them to provide an even more complete and thorough protection of sensitive data.

The paper is structured as follows. In Section 2 we compare our approach with related work, while in Section 3 we introduce the background. In Section 4 we give an overview of our approach, while providing a more detailed description in Section 5. We briefly describe the implementation of our CAC scheme and present the performance evaluation in Section 6.[6] We conclude the paper with final remarks and future work in Section 7.

## 2 Related Work

During the analysis of the large number of papers devoted to secure data in IoT scenarios, we have collected the key properties discussed and present them in Table 1. For lack of space, we only provide a discussion of the most closely related works.

In [9], the authors propose to plug in into MQTT-based IoT scenarios a logically centralized entity for enforcing Attribute-Based Access Control (ABAC) policies. While a traditional AC mechanism allows for context awareness and (horizontal) scalability, the proposal requires full trust on the central agent and does not employ cryptography to guarantee integrity and confidentiality.

Table 1: Comparison with Related Work

| | [9] | [18] | [8] | [19] | [15] | [11] | Our work |
|---|---|---|---|---|---|---|---|
| Channel encryption | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| End-to-end encryption | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ |
| Integrity guarantee | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| AC policy enforcement | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Scalable w.r.t. #subscribers | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ |
| Context Awareness | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓* |
| Suit constrained IoT devices | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |

*as mentioned in Section 1 and discussed at the end of Section 5.2, we can easily complement our CAC scheme with traditional AC enforcement mechanisms for context awareness

[6] An extended version of this work containing more details about the implementation is available at ....

As constrained IoT devices can hardly support TLS, in [18] the authors propose an alternative lightweight security mechanism for MQTT. Each MQTT client is equipped with a smart card containing the public key of the broker which is used to agree upon a session key (unique for each client). Relying on a smart card relieve resource consumption, while symmetric cryptography ensures confidentiality (but not end-to-end encryption). On the other hand, the broker needs to encrypt messages for each client separately, yielding a non-negligible overhead, and AC policy enforcement, as data integrity, is not discussed.

In [8], the authors design a secure communication scheme for MQTT based on the Augmented Password-Only Authentication and Key Exchange (AugPAKE) protocol.[7] Each client establishes a symmetric key with the broker to encrypt communications, while topics are associated with authorization tokens. As in [18], the per-client re-encryption of MQTT messages makes the solution hardly scalable, and the broker has access to the plaintext MQTT messages. Finally, the authors do not discuss mechanisms to provide data integrity.

Even when the client-broker link is encrypted (e.g., via TLS), processing data in clear at the broker constitutes a privacy and security risk. As such, in [19] the authors propose the use of Trusted Execution Environments (TEEs) at the broker: whenever a client publishes a message to a topic, the message is encrypted with a symmetric key previously shared with the TEE and then sent to the broker over TLS. While achieving end-to-end encryption and integrity, this approach suffers from an overhead that can be up to 8× in some scenarios.

In [15], the authors propose a framework for protecting MQTT-based IoT scenarios with 3 increasing security levels: the first provides data integrity, authenticity and accountability, the second adds confidentiality while the third offers long-term security. While having different security levels allows adapting the solution to the requirements of different IoT scenarios (e.g., latency, scalability), the proposed solution neither preserves the confidentiality of data from the MQTT broker nor considers the enforcement of AC policies.

In [11], the authors discuss the protection of data in an eHealth scenario where several wearable devices (e.g., smartwatches, pacemakers) communicate with a single predetermined entity (i.e. the doctor assigned to the patient) through symmetric cryptography. Similarly to our approach, this solution provides end-to-end encryption and integrity, and it is suitable for constrained IoT devices. However, it supports many-to-1 communication scenarios only.

In conclusion, the main difference between the work presented in this paper and the above works is that the latter do not provide end-to-end encryption while enforcing AC policies and supporting many-to-many communications. This is a novel contribution of our approach as shown in Table 1.

## 3 Background

We describe some concepts of AC and Role-Based Access Control (RBAC). We also overview the MQTT protocol and the Mosquitto MQTT broker.

[7] https://datatracker.ietf.org/doc/draft-irtf-cfrg-augpake/

### 3.1 Access Control

Samarati and De Capitani di Vimercati [17] defined AC as “the process of mediating every request to resources maintained by a system and determining whether the request should be granted or denied”. A resource usually consists of data such as messages or files. In the following, we assume an AC policy $\mathbf{P}$ to be compiled into a RBAC model rather than an a ABAC model (as in [9]), since support for the enforcement of RBAC policies is readily available in several MQTT broker implementations, thus simplifying the experimental validation of our approach. In this work, the state of $\mathbf{P}$ can be described as a tuple $(\mathbf{U}, \mathbf{R}, \mathbf{F}, \mathbf{UR}, \mathbf{PA})$, where $\mathbf{U}$ is the set of users, $\mathbf{R}$ is the set of roles, $\mathbf{F}$ is the set of resources, $\mathbf{UR} \subseteq \mathbf{U} \times \mathbf{R}$ is the set of user-role assignments and $\mathbf{PA} \subseteq \mathbf{R} \times \mathbf{PR}$ is the set of role-permission assignments, being $\mathbf{PR} \subseteq \mathbf{F} \times \mathbf{OP}$ a derivative set of $\mathbf{F}$ combined with a fixed set of operations $\mathbf{OP}$ (both $\mathbf{PR}$ and $\mathbf{OP}$ are not included in the state of the AC policy as they remain constant over time). A user $u$ can use a permission $\langle f, op \rangle$ if $\exists r \in \mathbf{R} : (u, r) \in \mathbf{UR} \wedge (r, \langle f, op \rangle) \in \mathbf{PA}$. Role hierarchies can always be compiled away by adding suitable pairs to $\mathbf{UR}$.

### 3.2 MQTT

MQTT is a lightweight *publish-subscribe* messaging protocol,[8] widely employed in scenarios involving (computationally constrained) IoT devices. MQTT expects a message to be published to a *topic*, which can be seen as a temporary communication channel, grouping messages logically related to each other (e.g., concerning a specific location, event or action). An IoT device (in this context called “MQTT client”) can subscribe to a topic, thus expressing the will to receive messages published to that topic. Whenever a client wants to publish a message to a topic, it sends the message (and the name of the topic) to a server called “MQTT broker”, which can be seen as the central node of a star network topology. When the broker receives the message and the name of the topic, it broadcasts the message to all MQTT clients that previously subscribed to that topic. Each topic can have one “retained” message, i.e., a message stored by the broker and sent to each client that subscribes to the topic.

Among MQTT broker implementations, it is common to find extensions supporting security mechanisms such as TLS and centralized enforcement of AC policies based on, e.g., roles and access control lists. For instance, Mosquitto[9] is an open-source (EPL/EDL licensed) message broker maintained by Eclipse that implements the MQTT protocol (versions 5.0, 3.1.1 and 3.1). Mosquitto provides a variety of functionalities including the DYNamic SECurity (DYNSEC) plugin, which enforces dynamic RBAC policies via a centralized enforcement point.

[8] https://www.iso.org/standard/69466.html

[9] https://mosquitto.org/

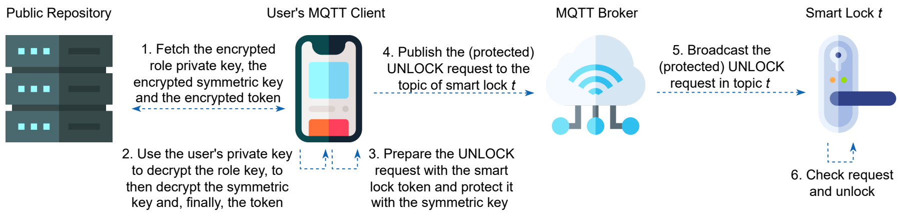
Public Repository
User's MQTT Client
MQTT Broker
Smart Lock *t*
1. Fetch the encrypted role private key, the encrypted symmetric key and the encrypted token
2. Use the user's private key to decrypt the role key, to then decrypt the symmetric key and, finally, the token
3. Prepare the UNLOCK request with the smart lock token and protect it with the symmetric key
4. Publish the (protected) UNLOCK request to the topic of smart lock *t*
5. Broadcast the (protected) UNLOCK request in topic *t*
6. Check request and unlock

Fig. 1: Instance of an UNLOCK Request to a Smart Lock

# 4 Overview

First, we discuss a smart building scenario (as in [9]) focusing on an IoT service commonly considered in the literature, i.e., Smart Lock (Section 4.1). Then, we give an overview of our CAC-based approach for the end-to-end protection of sensitive data exchanged by IoT devices through the MQTT protocol (Section 4.2) in the context of the Smart Lock service previously described. Below, we keep the discussion at a high level to allow the reader to get a general understanding of the approach before delving into (complex) details in Section 5.

## 4.1 Smart Lock Service

Organizations operating in large buildings (e.g., government structures, hospitals, research centres) have to manage access to several locals, some of which might contain confidential documents, delicate equipment or health hazards. In this context, smart locks may be used to regulate and restrict access to rooms, laboratories and closets more efficiently than traditional locks [2, 1, 3] by enforcing RBAC policies that are administered mandatorily (this means that delegations are not relevant in this scenario). A smart lock can be seen as a cyber-physical device made of a smart cylinder and a microcontroller with limited computational, storage and communication capabilities.

A smart lock usually requires the use of a token to be unlocked, which is distributed to (authorized) users according to their qualifications. Generalizing, we can say that users are assigned to one or more roles by the system administrator, where the roles reflect the internal hierarchy of an organization (e.g., employee, canteen staff, human resources). For instance, in a research centre, the AC policy may assign to members of the cleaning service the permission to access all rooms in a building except for laboratories, while members of a research unit may have the permission to access their laboratory only. After having chosen a role to assume, a user can interact with the smart lock through a dedicated MQTT topic. For instance, a smart lock “LOCK_ID” located on the first floor of a building may be subscribed to the topic “building/firstfloor/$LOCK_ID”. In this way, a user (who belongs to an authorized role) can publish to that topic UNLOCK and LOCK requests by presenting the related token.

### 4.2 Securing the Smart Lock Service

CAC involves the use of cryptography to enforce AC policies while guaranteeing the confidentiality and the integrity of sensitive data. In the Smart Lock service described in Section 4.1, the AC policy corresponds to the assignments between users and roles (e.g., cleaning service, research unit) and roles to permissions (e.g., which locals of a building the members of a role can un/lock), while the data to protect correspond to the tokens of the smart locks. Cryptography is then employed to implement role memberships, distribute the tokens according to the AC policy and secure them when transmitted among IoT devices.

More specifically, each user (i.e., MQTT client) and each role, where roles are defined by the administrator according to the internal hierarchy of the organization, is provided with a pair of asymmetric cryptographic keys. Instead, each smart lock is provided with a dedicated symmetric key and assigned to an MQTT topic (i.e., one key and topic for smart lock). The token of each smart lock is encrypted with the related symmetric key, which is in turn (separately) encrypted with the public keys of all roles that are authorized to interact with that smart lock. Similarly, the private keys of all roles to which a user belongs are (separately) encrypted with the user's public key. All encrypted information (i.e., tokens, symmetric keys and roles' private keys) are made available through a public repository. In addition, this information is digitally signed by the administrator of the policy to guarantee its integrity, and the digital signatures are stored together with the information.

Whenever a user wants to send an UNLOCK request to a smart lock, she first chooses one role which is authorized to open the lock. As shown in Figure 1, she uses her private key to decrypt the private key of the role, which is in turn used to decrypt the symmetric key. Then, the user can decrypt the token of the smart lock with the symmetric key. Finally, the user exploits the knowledge of the token to interact with the smart lock (e.g., by engaging in a challenge-response protocol), using the same symmetric key to secure MQTT messages published to the topic of the smart lock. Neither the (partially trusted service provider hosting the) MQTT broker (e.g., the Cloud or Edge) nor possible Man-in-the-Middle (MitM) attackers nor malicious insiders can access the (encrypted) token, while digital signatures make tampering attempts obvious.

Unfortunately, the symmetric keys cannot be hard-coded into the smart locks or the users' MQTT clients, and must instead be dynamically distributed. Intuitively, a new key has to be created whenever a new smart lock is added to the building. Besides, whenever a permission is revoked, the involved symmetric key (as well as the token) must be renewed. Otherwise, revoked users could use cached keys to still be able to decrypt MQTT messages or collude with the service provider hosting the MQTT broker. The use of TEEs can potentially relieve this issue (e.g., see the Cloud-based CAC schemes proposed in [14] and [10]). However, constrained IoT devices are not likely to be equipped with a TEE. To renew a symmetric key, the administrator has to distribute new public or private information (e.g., the new symmetric key) to all users, along with version numbers to differentiate between old and new information. The new information and the version numbers are stored in the public repository as well. Asymmetric cryptography such as Identity-Based Encryption (IBE), Public Key Infrastructure (PKI)-based and Attribute-Based Encryption (ABE) is usually employed to regulate access to the new information (i.e., ensure only authorized users can decrypt the new symmetric key) and provide accountability (i.e., ensure that the new symmetric key was indeed created by the administrator).

## 5 Cryptographic Access Control for MQTT

We present the design of a role-based CAC scheme for the end-to-end encryption of sensitive data exchanged through MQTT in IoT-based scenarios. We choose MQTT since it is one of the most employed publish-subscribe protocols in IoT [16]. However, our scheme can adapt to other publish-subscribe protocols as well (e.g., AMQP[10]). The design of our CAC scheme is inspired to the work in [12] with several technical variations and two notable differences, namely the context of use (i.e., Cloud in [12] vs. IoT in this work) and the protection of data (i.e., at-rest in [12] vs. in-transit in this work). Below, we first discuss how to map RBAC elements to MQTT concepts (Section 5.1). Then, we present our CAC scheme (Section 5.2). Finally, we provide some considerations on the security of the scheme (Section 5.3). The symbols used in this Section are in Table 2.

### 5.1 Role-based Access Control to MQTT

We map MQTT clients and MQTT topics to the set of users $\mathbf{U}$ and resources $\mathbf{F}$ of the RBAC policy, respectively. The set of roles $\mathbf{R}$ is instead defined by the administrator, as described in Section 4.1, thus roles are not mapped to any MQTT concept. The set of operations $\mathbf{OP}$ is composed by publish ($\mathsf{Pub}$) and subscribe ($\mathsf{Sub}$). Each user $u$ and role $r$ is provided with a pair of asymmetric keys $(\mathbf{k}^{\mathbf{enc}}, \mathbf{k}^{\mathbf{dec}})$ for en/decryption. Besides this key pair, the administrator $A$ is provided with an additional pair of asymmetric keys $(\mathbf{k}^{\mathbf{ver}}_A, \mathbf{k}^{\mathbf{sig}}_A)$ for verification/creation of digital signatures. Each topic $f$ is assigned to a symmetric key $\mathbf{k}^{\mathbf{sym}}_f$, used to encrypt each message $m$ in $f$, resulting in $\mathbf{Enc}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_f}(m)$.

To assign a user $u$ to a role $r$, $r$'s decryption key $\mathbf{k}^{\mathbf{dec}}_r$ is encrypted with $u$'s encryption public key $\mathbf{k}^{\mathbf{enc}}_u$, resulting in $\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_u}\left(\mathbf{k}^{\mathbf{dec}}_r\right)$. To give permission to a role $r$ over a topic $f$ (e.g., to allow the users assigned to the role $r$ to lock and unlock the smart lock corresponding to the topic $f$), $f$'s symmetric key $\mathbf{k}^{\mathbf{sym}}_f$ is encrypted with $r$'s encryption public key $\mathbf{k}^{\mathbf{enc}}_r$, resulting in $\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_r}\left(\mathbf{k}^{\mathbf{sym}}_f\right)$.

To handle revocations, we associate version numbers to (the keys of) roles and topics. The administrator only can create a new topic $f$ by generating a new symmetric key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}$ and publishing a retained message to $f$ containing the version number $v_f$, which is initially equal to 1. Whenever the key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}$ needs to be updated (due to, e.g., user's revocation), the administrator generates

[10] https://www.amqp.org/

a new symmetric key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)}$ and replaces the retained message with a new one, containing the (updated) version number $v_f+1$, i.e., the old version number plus 1. In this way, users are notified of the key renewal and can update their key

Table 2: Symbols

| Symbol | Description |
|---|---|
| $e$ | A generic entity (either a user, a role or a topic) |
| $u$ | A generic user |
| $A$ | The administrator user |
| $r$ | A generic role |
| $f$ | A generic topic |
| $v_e$ | A generic version number for the entity $e$ |
| $op$ | Either $\{\mathsf{Sub}\}$, $\{\mathsf{Pub}\}$ or $\{\mathsf{Sub}, \mathsf{Pub}\}$ |
| $\mathbf{N}$ | Null (i.e., empty) value |
| $m$ | A generic plaintext |
| $c$ | A generic ciphertext |
| $-$ | Wildcard |
| $\mathbf{Gen}^{\mathbf{Pub}}$ | Generation of key pair for en/decryption |
| $\mathbf{Gen}^{\mathbf{Sig}}$ | Generation of key pair for signatures |
| $\mathbf{Gen}^{\mathbf{Sym}}$ | Generation of symmetric key |
| $\mathbf{k}^{\mathbf{enc}}_{(e,v_e)}$ | Public encryption key of $(e, v_e)$ |
| $\mathbf{k}^{\mathbf{dec}}_{(e,v_e)}$ | Private decryption key of $(e, v_e)$ |
| $\mathbf{k}^{\mathbf{ver}}_{(e,v_e)}$ | Public verification key of $(e, v_e)$ |
| $\mathbf{k}^{\mathbf{sig}}_{(e,v_e)}$ | Private signing key of $(e, v_e)$ |
| $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}$ | Symmetric key of topic $(f, v_f)$ |
| $\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(e,v_e)}}(-)$ | Encryption with public key $\mathbf{k}^{\mathbf{enc}}_{(e,v_e)}$ of $-$ |
| $\mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}(e,v_e)}(-)$ | Decryption with private key $\mathbf{k}^{\mathbf{dec}}(e, v_e)$ of $-$ |
| $\mathbf{Enc}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}}(m)$ | Symmetric encryption with key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}$ of $m$ |
| $\mathbf{Dec}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}}(c)$ | Symmetric decryption with key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}$ of $c$ |
| $\langle \mathbf{U_t}, \mathbf{R_t}, \mathbf{F_t}, \mathbf{UR_t}, \mathbf{PA_t} \rangle$ | The state of the traditional AC policy |
| $\mathbf{U_t}$ | Set of users; a member is a single value $u$ |
| $\mathbf{R_t}$ | Set of roles; a member is a single value $r$ |
| $\mathbf{F_t}$ | Set of topics; a member is a single value $f$ |
| $\mathbf{UR_t}$ | Set of user-role pairs; a member is a tuple $(u, r)$ |
| $\mathbf{PA_t}$ | Set of role-permissions; a member is a tuple $(r, \langle f, op \rangle)$ |
| $\langle \mathbf{U_c}, \mathbf{R_c}, \mathbf{F_c}, \mathbf{UR_c}, \mathbf{PA_c} \rangle$ | The state of the CAC policy |
| $\mathbf{U_c}$ | Set of users; a member is a tuple $(u, \mathbf{k}^{\mathbf{enc}}_u, \mathbf{k}^{\mathbf{ver}}_u)$ |
| $\mathbf{R_c}$ | Set of roles; a member is a tuple $(r, \mathbf{k}^{\mathbf{enc}}_{(r,v_r)}, v_r)$ |
| $\mathbf{F_c}$ | Set of topics; a member is a tuple $(f, v_f)$ |
| $\mathbf{UR_c}$ | Set of user-role pairs; a member is a tuple $\left(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_u}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), v_r\right)$ |
| $\mathbf{PA_c}$ | Set of role-permission pairs; a member is a tuple $\left(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_r, v_A, op\right)$ |

accordingly. To delete a topic, the administrator removes the retained message, notifying all users to unsubscribe from the topic.

### 5.2 Full Construction

As illustrated in Section 3.2, the Mosquitto MQTT broker can enforce dynamic RBAC policies through the DYNSEC plugin as a centralized entity. Therefore, to provide an additional security layer besides cryptography, we synchronize the DYNSEC AC policy with the CAC policy. In this way, every modification performed in one policy is mirrored in the other. For instance, adding a user in the CAC policy implies adding a user in the DYNSEC policy as well, although the two actions are implemented differently. We highlight that the same kind of synchronization can also be implemented with other traditional AC enforcement mechanisms to enable the evaluation of permissions depending on contextual (e.g., time-based) conditions (e.g., such as the approach presented in [9]).

The state of the traditional AC policy enforced through the DYNSEC plugin can be described as a tuple $\langle \mathbf{U_t}, \mathbf{R_t}, \mathbf{F_t}, \mathbf{UR_t}, \mathbf{PA_t} \rangle$ (where the subscript $\mathbf{t}$ stands for "traditional"), while the state of the CAC policy can be described as a tuple $\langle \mathbf{U_c}, \mathbf{R_c}, \mathbf{F_c}, \mathbf{UR_c}, \mathbf{PA_c} \rangle$ (where the subscript $\mathbf{c}$ stands for "cryptographic"). Essentially, the CAC policy extends the traditional AC policy with additional metadata (e.g., digital signatures, public keys, version numbers). Figure 3 presents the pseudocode of each action available in the CAC scheme which acts on both the state of the traditional AC policy and that of the cryptographic AC policy by updating the components of the tuples. In detail, we write $\mathbf{P_t} \overset{\alpha}{\Longrightarrow} \mathbf{P_t}'$ and $\mathbf{P_c} \overset{\alpha}{\Longrightarrow} \mathbf{P_c}'$ to denote that the action $\alpha$ (selected among those in Figure 3) transforms the state $\mathbf{P_t}$ or $\mathbf{P_c}$ into the state $\mathbf{P_t}'$ or $\mathbf{P_c}'$, respectively. An action $\alpha$ in Figure 3 belongs to one of two categories:

- *administrative* - includes all actions performed by the administrator for the management of the AC and the CAC policies. First, the administrator initializes the system ($\mathit{initA}()$). Then, she can add and delete users ($\mathit{addU}(u)$, $\mathit{delU}(u)$ ), roles ($\mathit{addR}(r)$, $\mathit{delR}(r)$) and topics ($\mathit{addP}(f)$, $\mathit{delP}(f)$). Finally, she can assign and revoke users from roles ($\mathit{assignU}(u, r)$, $\mathit{revokeU}(u, r)$) as well as assign and revoke permissions from roles ($\mathit{assignP}(r, \langle f, \mathit{op} \rangle)$, $\mathit{revokeP}(r, \langle f, \mathit{op} \rangle)$);
- *operative* - includes all actions performed by the MQTT clients. After the administrator created the corresponding user in the AC policy, an MQTT client can generate her asymmetric keys ($\mathit{init}_u()$). Afterwards, she can subscribe to a topic $f$ ($\mathit{sub}_u(f, c)$) and also publish messages ($\mathit{pub}_u(f, m)$), according to the policy defined by the administrator. Messages received from a topic are en/decrypted as described in Section 5.1.

All metadata are digitally signed by the administrator with her signature creation key $\mathbf{k}_A^{\mathbf{sig}}$ and verified with her verification key $\mathbf{k}_A^{\mathbf{ver}}$. The integrity of MQTT messages is protected by using (symmetric) authenticated encryption, which is usually implemented through Message Authentication Codes (MACs)

[6]; for the sake of simplicity, in Figure 3 we omit these details and also other trivial checks like the uniqueness of identifiers. We highlight that there exist different approaches to authenticated encryption, depending on the padding strategy, on whether the MAC is computed over the ciphertext or the plaintext and, in the latter case, on whether the MAC is encrypted together with the plaintext. The security of these approaches depends on the context and the specific cryptographic algorithm employed; we discuss more in detail this point when presenting the implementation of the pseudocode in Section 6.

Instead, here we focus on the key relationships between traditional and cryptographic policies that can be stated as invariants of the operations in Figure 3. Formally, we define a predicate as a Boolean-valued function on the set of states of a traditional or cryptographic AC policy. The value of a predicate $Q$ on a state $\mathbf{P}$ is denoted with $\mathbf{P} \models Q$. An action $\alpha$ leaves a predicate $Q$ invariant if and only if $\mathbf{P} \models Q$ implies $\mathbf{P'} \models Q$ whenever $\mathbf{P} \overset{\alpha}{\Longrightarrow} \mathbf{P'}$. In detail, any action $\alpha$ in Figure 3 leaves the following predicates invariant:

- $(u, -, -) \in \mathbf{U_c}$ iff $u \in \mathbf{U_t}$;
- $(r, -, -) \in \mathbf{R_c}$ iff $r \in \mathbf{R_t}$;
- $(f, -) \in \mathbf{F_c}$ iff $f \in \mathbf{F_t}$;
- $(u, r, -, -) \in \mathbf{UR_c}$ iff $(u, r) \in \mathbf{UR_t}$;
- $(r, f, -, -, -, op) \in \mathbf{PA_c}$ iff $(r, \langle f, op\rangle) \in \mathbf{PA_t}$.

By using the invariant above and recalling, from Section 3.1, the definition of a user $u \in \mathbf{U_t}$ being able to use a permission $\langle f, op\rangle$ in state $\langle \mathbf{U_t}, \mathbf{R_t}, \mathbf{F_t}, \mathbf{UR_t}, \mathbf{PA_t}\rangle$, i.e., $\exists r \in \mathbf{R_t} : (u, r) \in \mathbf{UR_t} \wedge (r, \langle f, op\rangle) \in \mathbf{PA_t}$, it is easy to see that the user $u$ can use permission $\langle f, op\rangle$ iff $\exists (r, \mathbf{k}^{\mathbf{enc}}_{(r,v_r)}, -) \in \mathbf{R_c} : (u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_u}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), -) \in \mathbf{UR_c}$ and $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), -, -, op) \in \mathbf{PA_c}$. In turn, by recalling the definitions of the operations $\underline{assignU}$ and $\underline{assignP}$ in Figure 3, this implies that the user $u$ has access to the corresponding symmetric key $\mathbf{k}^{\mathbf{sym}}_f$ in the state $\langle \mathbf{U_c}, \mathbf{R_c}, \mathbf{F_c}, \mathbf{UR_c}, \mathbf{PA_c}\rangle$ by using her private key $\mathbf{k}^{\mathbf{dec}}_u$ to decrypt $\mathbf{k}^{\mathbf{dec}}_r$, and then using the role's private key $\mathbf{k}^{\mathbf{dec}}_r$ to decrypt $\mathbf{k}^{\mathbf{sym}}_f$, as explained at the beginning of Section 5.1.

The invariants above show that the traditional and cryptographic policies are synchronized on the authorization conditions depending on a pure RBAC model. This allows refining such conditions with additional ones depending on contextual information (e.g., time-based permissions) as discussed at the end of Section 1 that can be checked only with a traditional enforcement mechanism.

### 5.3 Security Considerations

Our CAC scheme allows administrators to enforce RBAC policies both traditionally and cryptographically. This capability restricts access to MQTT topics to authorized users only. Besides, it provides end-to-end protection for guaranteeing confidentiality and integrity of MQTT messages from both external attackers and the (partially-trusted agent managing the) MQTT broker. We assume that cryptographic primitives are perfect, i.e., the confidentiality and integrity of encrypted MQTT messages cannot be violated except by (computationally infeasible) brute force attacks. Then, the traditional AC policy enforced by DYNSEC allows only authorized users to publish and subscribe to topics. We note that, without the corresponding symmetric key, an unauthorized user could not produce a valid MQTT message anyway.

In our CAC scheme, accountability—the ability to map MQTT messages to the corresponding publishers—is currently not ensured cryptographically, since messages are (hashed and) signed with symmetric keys known by all authorized users, as presented in Section 5.2. However, since users have to authenticate toward the MQTT broker, the broker itself can provide accountability by mapping each MQTT message to the MQTT client (thus, the user) that published it. Nonetheless, a scenario based on a Zero Trust model may call for a stronger guarantee of accountability. In this case, users can easily be provided with an additional pair of asymmetric keys and required to sign MQTT messages to guarantee accountability through cryptography, at the cost of incurring additional overhead. The modification to the CAC scheme for implementing this requirement would be straightforward and can be seen as an instance of the AC model introduced in [4, 5] that considers the features of the client used by a subject to access a certain resource. Despite a subject (e.g., a general practitioner) can be entitled to read a sensitive resource (e.g., the healthcare information of a patient), it can be denied such a right because of the low level of protection offered by the client (e.g., personal smartphone) or it can be granted when an adequate client is used (e.g., desktop operated by the hospital). Finally, we note that protection against replay attacks has to be provided by smart-lock supported mechanisms (e.g., timestamps, challenge-response protocol) and it is out of the scope of this paper.

As illustrated in Table 2, the CAC policy contains public (e.g., public keys) or encrypted (e.g., encrypted private keys) information only. Indeed, as shown in Section 5.2, the symmetric keys of topics are encrypted with the public keys of authorized roles, while the private keys of roles are encrypted with the authorized users' public keys. Therefore, by construction, only authorized users can decrypt roles' private keys and, consequently, access symmetric keys to en/decrypt MQTT messages. In other words, even though the CAC policy is public, only authorized users can obtain secret keys (i.e., the roles' private keys and the topics' symmetric keys). Adding permissions to the CAC policy is a straightforward operation, as it consists in encrypting private information (i.e., secret keys) with the public key of the newly authorized users (or roles). On the other hand, the revocation of permissions requires careful management of cryptographic material. We consider the worst-case scenario in which a user $u$ previously cached all secret keys she could access, both of roles and topics. Hence, when revoking permissions from $u$ (or from one of the roles that $u$ is assigned to), we need to renew the affected secret keys. In detail, when revoking a permission $\langle f, op\rangle$ from a role $r$ (i.e., when invoking the function $revokeP(r, \langle f, op\rangle))$ in our CAC scheme, we distinguish two cases:

- if $r$ already had permission $op'$ so that $op' \subseteq op$, we generate a new key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)} \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$ for $f$, which we distribute to all *other* authorized roles. In this way, users belonging to $r$ do not have access to the new key;
- if $r$ already had permission $op'$ so that $op' \cap op \neq \emptyset$, we simply update $\mathbf{PA_t}$ and $\mathbf{PA_c}$ by removing the permissions in $op$ from $op'$.

Similarly, when revoking a user $u$ from a role $r$ (i.e., when invoking the function $revokeU(u, r)$), we generate new keys $(\mathbf{k}^{\mathbf{enc}}_{(r,v_r+1)}, \mathbf{k}^{\mathbf{dec}}_{(r,v_r+1)}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$ for $r$ and distribute them to all *other* authorized users. In this way, $u$ does not have access to $r$'s new private key. Finally, we renew the symmetric keys of all topics that $r$ had permission over with a procedure similar to the $revokeP$ function.

## 6 Implementation and Experimental Evaluation

We implement the pseudocode presented in Figure 3 in a tool which we name *"CryptoAC"* that we made open-source and freely available.[11] *CryptoAC* is supposed to be deployed as a standalone MQTT client (i.e., *CryptoAC* and the MQTT client coincide and physically run on the same IoT device). Altogether, the CAC scheme involves three entities, i.e., *CryptoAC*/MQTT client, the MQTT broker and the public repository. Below, we provide details on the implementation (or configuration) of each of these entities and their interactions (Sections 6.1, 6.2, 6.3). Finally, we present our performance evaluation of *CryptoAC* (Section 6.4).

### 6.1 CryptoAC

As the programming language to implement *CryptoAC*, we choose Kotlin[12] for the intrinsic portability and the possibility of natively deploying *CryptoAC* in IoT devices. Indeed, Kotlin supports multiplatform programming[13] which allows compiling Kotlin code directly into native code, thus avoiding the computational overhead due to the use of a Java Virtual Machine (JVM). This is especially true since Kotlin mainly targets Linux-based environments,[14] which are the most deployed in IoT devices.[15]

As the cryptographic provider we choose Sodium,[16] a modern and portable cryptographic library. Sodium is also (reasonably) secure, as its implementation was thoroughly audited and no critical flaws or vulnerabilities were found.[17] Sodium uses the Elliptic Curves Diffie-Hellman (ECDH) algorithm (X25519) to generate public-private keys and the Edwards-curve Digital Signature Algorithm

[11] https://github.com/stfbk/CryptoAC

[12] https://kotlinlang.org/

[13] https://kotlinlang.org/docs/multiplatform.html

[14] https://www.jetbrains.com/lp/devecosystem-2019/

[15] https://outreach.eclipse.foundation/iot-edge-developer-2021

[16] https://libsodium.gitbook.io/doc/

[17] https://www.privateinternetaccess.com/blog/libsodium-audit-results/

(EdDSA) for digital signatures (Ed25519). Like many ciphers in TLS 1.3, Sodium supports authenticated encryption with associated data (AEAD), which is a more robust and secure variant of authenticated encryption (recall the discussion in Section 5.2) allowing to bind the ciphertext to the context where it is supposed to be used. We observe that the usage of AEAD is in line with the requirements contained in the call for Lightweight Cryptography to protect small electronics (thus including IoT devices) issued by NIST.[18] In detail, the MAC is computed over the combination of the ciphertext with contextual information (e.g., version numbers, nonces), which is attached as a plaintext along with the ciphertext. In this way, attackers cannot reuse a valid ciphertext in a different context.

In this regard, Sodium proposes the use of the XSalsa20 symmetric stream cypher (i.e., Salsa20 with 192-bit nonce extension) together with the Poly1305 universal hash function as the best option, instead of using 256-bit AES in Galois/Counter Mode (GCM) with, e.g., the SHA-384 hash function. The latter is typically used in TLS 1.3 deployments labelled as TLS_AES_256_GCM_SHA384, and it will be used in our experiments as discussed at the end of Section 6.4 below. One of the reasons for this choice is that, although hardware acceleration for AES is often available in modern processors, its performance on platforms that lack such hardware is considerably lower. Another issue is that many software-only AES implementations are vulnerable to cache-collision timing attacks. Instead, XSalsa20 is faster than (non-accelerated) AES and it achieves homogeneous performance independently of the underlying hardware, enhancing portability.

Finally, we use the Eclipse Paho client for Java[19] to interact with the MQTT broker. It is worth noting that *CryptoAC* caches symmetric keys of topics at the client-side after the first use, so to avoid having to obtain them (as described in Section 5.1) every time a message is received or needs to be published. In this way, we increase the efficiency of the implementation by eliminating superfluous cryptographic computations. Of course, when symmetric keys are renewed (e.g., after a revocation), the cache is invalidated. All secret keys are securely stored in a Java Keystore.

*CryptoAC* can also run as an administrative tool by acting as a web server with the Ktor library[20] offering its functions through either RESTful Application Programming Interfaces (APIs) or a web interface developed with Kotlin/JS and the React library.[21] Through the administrative interface of *CryptoAC*, the administrator can manage the (traditional and cryptographic) AC policy. All the inputs to the interface are validated with OWASP-approved regular expressions[22] to avoid web-based attacks (e.g., injection, Cross-Site Scripting).

[18] https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics

[19] https://www.eclipse.org/paho/

[20] https://ktor.io/

[21] https://it.reactjs.org/

[22] https://owasp.org/www-community/OWASP_Validation_Regex_Repository

### 6.2 MQTT Broker

We choose Mosquitto[23] as MQTT broker. As introduced in Section 5.2, we enable the DYNSEC plugin for traditional AC enforcement on top of CAC. This additional security layer guarantees redundancy and allows to restrict the permissions of the users (i.e., to specify whether a user can subscribe, publish or perform both actions on a topic). Of course, a user could potentially collude with the (service provider hosting the) MQTT broker to bypass the DYNSEC AC policy enforcement and gain publish and/or subscribe privileges. However, we highlight that this kind of collusions may happen regardless of whether the DYNSEC plugin is enabled, and that the colluding user should have the symmetric key of the topic anyway to en/decrypt messages on that topic (i.e., the CAC policy should already give publish or subscribe permissions to the colluding user on that topic). Intuitively, the same may happen if secret or symmetric keys are stolen or leaked from an IoT device. However, the physical and cyber security of IoT devices themselves (e.g., concerning physical attackers or firmware vulnerabilities) is out of the scope of this paper. Finally, access to the MQTT broker is protected by individual passwords.

### 6.3 Public Repository

As in [9], we choose Redis[24] to store metadata related to the CAC scheme. Indeed, Redis is primarily an in-memory storage, a characteristic that allows for low response time to queries. Moreover, the (dedicated) protocol used by Redis (i.e., RESP, REdis SerializationProtocol) is geared toward low latency. The metadata of each user (i.e., the public keys) are stored under a unique Redis key, while a list collects all users' Redis keys. We follow the same approach for the metadata of roles, topics, user-role assignments and role-topic permissions. Finally, access to the Redis datastore is protected by individual passwords.

### 6.4 Performance Evaluation

The authors in [9] deploy a reference monitor as a proxy between the MQTT clients and the MQTT broker. For this reason, they evaluate the scalability of their solution when varying the number of publishers and subscribers, i.e., when increasing the computational load on the reference monitor. Differently, *CryptoAC* is deployed as (part of) an MQTT client. As such, we are not interested in measuring scalability, as each MQTT publisher or subscriber is an instance of *CryptoAC* on its own. In other words, we achieve scalability by design, since we distribute our solution for data protection and AC enforcement on IoT devices, thus avoiding the need for a centralized solution and allowing each MQTT client to stand by itself.

Therefore, below we present a preliminary experimental evaluation to analyze the computational overhead of our solution on a single IoT device, which can

[23] https://mosquitto.org/

[24] https://redis.io/

become the bottleneck for the performance of the entire scenario. We consider three configurations for the Smart Lock service presented in Section 4.1, as shown in Table 3. Note that, since they provide different although partially overlapping security guarantees, we explicitly distinguish between channel and end-to-end encryption:

- *C1*: in this configuration, the communication channel between the MQTT client and the MQTT broker is protected by neither TLS nor CAC, and the broker enforces AC policies through the DYNSEC plugin. We use this configuration as the baseline;
- *C2*: in this configuration, the communication channel between the MQTT client and the MQTT broker is protected with unilateral TLS 1.3 (i.e., MQTT clients verifies the broker's certificate but they are not required to provide a certificate in turn) and the MQTT broker enforces AC policies through the DYNSEC plugin. This configuration corresponds to the traditional solution for the protection of data in-transit, even though the confidentiality of data is not preserved from partially-trusted service providers hosting the broker. For fairness, we remark that in this configuration we do not measure the overhead due to the TLS handshake and session key derivation algorithms between MQTT clients and the MQTT broker, as it has already been found by other works that TLS as a whole is hardly usable by constrained IoT devices [18, 13]. Therefore, we just measure the transmission time *after* having fully established a TLS session;
- *C3*: in this configuration, we use the CAC scheme presented in Section 5 to provide end-to-end encryption while the MQTT broker enforces AC policies through the DYNSEC plugin.

*Experimental Settings* We are interested in measuring the overhead of cryptographic techniques for data protection (i.e., TLS in *C2* and CAC in *C3*) with respect to the baseline *C1*. Therefore, we reuse the same infrastructure and experimental settings across the three configurations to avoid possible measurement discrepancies. For instance, the use of another MQTT client (e.g., Mosquitto_sub[25] and Mosquitto_pub[26]) instead of Eclipse Paho could create biases in the measurements, as its implementation may be more (or less) performant than Paho. For this reason, we employ *CryptoAC* to implement all three

[25] https://mosquitto.org/man/mosquitto_sub-1.html

[26] https://mosquitto.org/man/mosquitto_pub-1.html

Table 3: Configurations for the Performance Evaluation

| Configuration | DYNSEC | Channel Encryption | End-to-end Encryption |
|---|---|---|---|
| *C1 (baseline)* | ✓ | ✗ | ✗ |
| *C2* | ✓ | ✓ (TLS) | ✗ |
| *C3* | ✓ | ✗ | ✓ (CAC) |

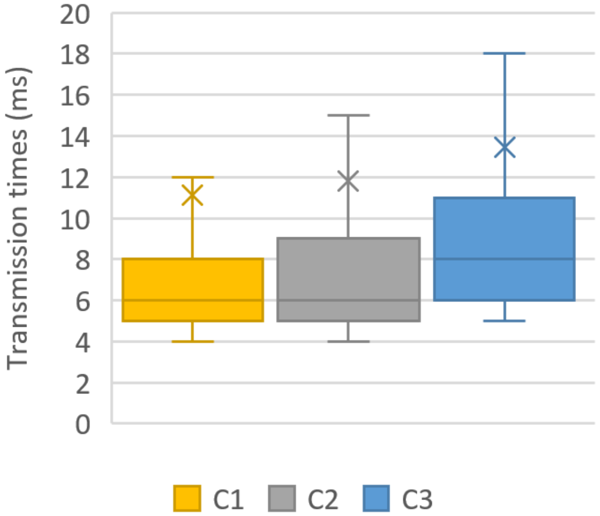

Fig. 2: Boxplot of transmission times for 1,000 MQTT messages exchanged in the configurations under test. The average of *C1* is 11.1ms, *C2* is 11.8ms and *C3* is 13.5ms

configurations in Table 3. In detail (and only during the performance evaluation), we remove all cryptographic computations from *CryptoAC* to implement *C1*. Similarly, we disable the CAC scheme but enable TLS in *CryptoAC* to implement *C2*. Finally, we use the original implementation of *CryptoAC* to implement *C3*. By doing so, we ensure that the underlying infrastructure remains the same across the different configurations and we are guaranteed to precisely measure the overhead of TLS (*C2*) and our CAC scheme (*C3*) with respect to the baseline (*C1*). Finally, we highlight that we compile *CryptoAC* to Java bytecode for ease of use, and leave the native deployment (which may be more suitable for IoT devices) for future work, as it is mainly an implementation effort.

We use Mosquitto 2.0.11 as the MQTT broker, running on an endpoint with Intel Xeon E3-1240 V2 (4 cores with Hyper-Threading @ 3.40GHz) as CPU and 16 GB of RAM. A Raspberry Pi 3 Model B+ (Cortex-A53 ARMv8 64-bit SoC @ 1.4GHz with 1GB of RAM) hosts two instances of *CryptoAC* (i.e., two MQTT clients): one that publishes to a topic a message with, as payload, a timestamp *T1* acquired just before (possibly encrypting and) publishing the message, and a second one that subscribes to that topic and acquires another timestamp *T2* after receiving and (possibly decrypting) the message from the broker. The network connections between the MQTT clients and the MQTT broker are configured as described in the Smart Lock service in Figure 1.

*Results and Discussion* As in [9], we measure the transmission time as the difference between *T2* and *T1*, with the two MQTT clients (a publisher and a subscriber) sharing the same host, avoiding therefore a possible time drift (i.e., avoiding the use of two hosts that lose clock synchronization over time). We repeat the measurements for *C1*, *C2* and *C3* with 1,000 individual MQTT messages exchanged and report the results as a box plot in Figure 2 (full results are available online in an anonymous repository[27]). The box is bounded by lower and upper quartiles, while the line indicates the median and the cross indicates the average. Upper and lower whiskers are computed as the default—1.5 × upper and lower interquartile range—while outliers have been omitted. On average, the measurements show an overhead of 0.7ms from *C1* to *C2* and 2.4ms from *C1* to *C3*.

As expected, the baseline configuration *C1* has the lowest average transmission time due to the fact that no cryptographic operation is executed. Once removed the burden of the TLS handshake and key derivation algorithms, *C2* incurs a negligible overhead. We believe that this is mainly due to the performance of the host running Mosquitto and the fact that the TLS session was using the TLS_AES_256_GCM_SHA384 cypher, for which the processor of the host supports hardware acceleration.[28] The use of the CAC scheme in *C3* yields an average overhead with respect to *C1* and *C2* of 2.4ms and 1.7ms, respectively. We believe that this is an acceptable overhead in a Smart Lock service, especially when considering the greater security guarantees offered by CAC. Furthermore, we believe that this overhead can be reduced by optimizing the implementation of *CryptoAC* and fine-tuning the parameters of the cryptographic algorithms employed. We leave the verification of these ideas as future work.

Finally, we investigate two variants of *C3*, i.e., one which disables DYNSEC to measure its overhead on the broker (configuration *C3B*) and one that removes the caching mechanism for symmetric keys to consider the worst-case scenario in which *CryptoAC* obtains a symmetric key for a topic for the first time, as described in Section 5.1 (configuration *C3C*). The results show that in *C3B* there is a negligible improvement of 0.1ms on average, an indicator that DYNSEC does not have a significant impact on the performance of Mosquitto. The average transmission time on *C3C* is 20.9ms on average, 7.4ms more than *C3*, which denotes that a worst-case scenario is still acceptable for the Smart Lock service.

## 7 Conclusion and Future Directions

In this paper, we proposed a CAC scheme for IoT scenarios based on the MQTT protocol to secure sensitive data against external attackers, malicious insiders and partially trusted agents while providing end-to-end encryption and enforcing role-based AC policies. We implemented the scheme in an open-source tool and conducted a preliminary performance evaluation. In our experiments, the use of CAC introduces an overhead of 1.7ms with respect to a scenario employing TLS (but without considering the handshake and key derivation algorithms),

[27] https://docs.google.com/spreadsheets/d/1U4mCw9_mkrPIk1KiTlcLjWiep7L5YuqZ

[28] https://ark.intel.com/content/www/us/en/ark/products/65730/intel-xeon-processor-e31240-v2-8m-cache-3-40-ghz.html

and 2.4ms when the channel is not secured. These results are in line with the requirements of the smart lock use case and the additional security guarantees provided by CAC.

We plan to extend our work in several directions including the use of ABE to allow more expressive and fine-grained ABAC policies and of TEEs to guarantee confidentiality and integrity in IoT scenarios, as in [19], and provide different levels of security, as in [15]. We also intend to adapt the technique for optimizing deployments of cryptographic enforcement mechanisms in the cloud of [7] to IoT scenarios.

# Bibliography

[1] Tahir Ahmad, Umberto Morelli, and Silvio Ranise. Deploying access control enforcement for IoT in the cloud-edge continuum with the help of the CAP theorem. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, pages 213–220. ACM, 2020.

[2] Tahir Ahmad, Umberto Morelli, Silvio Ranise, and Nicola Zannone. A lazy approach to access control as a service (acaas) for iot: An aws case study. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '18, page 235–246, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Tahir Ahmad, Umberto Morelli, Silvio Ranise, and Nicola Zannone. Extending access control in AWS IoT through event-driven functions: an experimental evaluation using a smart lock system. *International Journal of Information Security*, 2021.

[4] Alessandro Armando, Matteo Grasso, Sander Oudkerk, Silvio Ranise, and Konrad Wrona. Content-based information protection and release in NATO operations. In *Proceedings of the 18th ACM symposium on Access control models and technologies - SACMAT '13*, page 261. ACM Press, 2013.

[5] Alessandro Armando, Sander Oudkerk, Silvio Ranise, and Konrad Wrona. Formal modelling of content-based protection and release for access control in NATO operations. In Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquin Garcia-Alfaro, and Nur Zincir Heywood, editors, *Foundations and Practice of Security*, volume 8352, pages 227–244. Springer International Publishing, 2014. Series Title: Lecture Notes in Computer Science.

[6] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.

[7] Stefano Berlato, Roberto Carbone, Adam J. Lee, and Silvio Ranise. Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. *ACM Trans. Priv. Secur.*, 25(1), nov 2021.

[8] Marco Calabretta, Riccardo Pecori, and Luca Veltri. A token-based protocol for securing MQTT communications. In *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, 2018.

[9] Pietro Colombo and Elena Ferrari. Access control enforcement within MQTT-based internet of things ecosystems. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 223–234. ACM, 2018.

[10] Judicael B. Djoko, Jack Lange, and Adam J. Lee. NeXUS: Practical and secure access control on untrusted storage platforms using client-side SGX.

In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413. IEEE, 2019.

[11] Eman Elemam, Ayman M. Bahaa-Eldin, Nabil H. Shaker, and Mohamed A. Sobh. A secure MQTT protocol, telemedicine IoT case study. In *2019 14th International Conference on Computer Engineering and Systems (ICCES)*, pages 99–105. IEEE, 2019.

[12] William C. Garrison, Adam Shull, Steven Myers, and Adam J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 819–838, 2016.

[13] Tobias Heer, Oscar Garcia-Morchon, René Hummen, Sye Loong Keoh, Sandeep S. Kumar, and Klaus Wehrle. Security challenges in the IP-based internet of things. *Wireless Personal Communications*, 61(3):527–542, 2011.

[14] Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. Keys in the clouds: Auditable multi-device access to cryptographic credentials. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–10. ACM, 2018.

[15] Lukas Malina, Gautam Srivastava, Petr Dzurenda, Jan Hajny, and Radek Fujdiak. A secure publish/subscribe protocol for internet of things. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–10. ACM, 2019.

[16] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. Mqttsa: A tool for automatically assisting the secure deployments of mqtt brokers. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642-939X, pages 47–53, 2019.

[17] Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171, pages 137–196. Springer Berlin Heidelberg, 2000.

[18] Eduardo Buetas Sanjuan, Ismael Abad Cardiel, Jose A. Cerrada, and Carlos Cerrada. Message queuing telemetry transport (MQTT) security: A cryptographic smart card approach. *IEEE Access*, 8:115051–115062, 2020.

[19] Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio Schiavoni. MQT-TZ: Hardening IoT brokers using ARM TrustZone : (practical experience report). In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 256–265. IEEE, 2020.

[20] Sherali Zeadally, Ashok Kumar Das, and Nicolas Sklavos. Cryptographic technologies and protocol standards for internet of things. *Internet of Things*, page 100075, 2019.

## A Pseudocode of the Cryptographic Access Control Scheme

*initA*()

- Generate encryption key pair $(\mathbf{k}^{\mathbf{enc}}_A, \mathbf{k}^{\mathbf{dec}}_A) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$, signature key pair $(\mathbf{k}^{\mathbf{ver}}_A, \mathbf{k}^{\mathbf{sig}}_A) \leftarrow \mathbf{Gen}^{\mathbf{Sig}}$
- Add $(A, \mathbf{k}^{\mathbf{enc}}_A, \mathbf{k}^{\mathbf{ver}}_A)$ to $\mathbf{U_c}$, $(A, \mathbf{k}^{\mathbf{enc}}_A, 1)$ to $\mathbf{R_c}$ and $(A, A, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_A}\left(\mathbf{k}^{\mathbf{dec}}_A\right), 1)$ to $\mathbf{UR_c}$
- Add $A$ to $\mathbf{U_t}$ and $(A, A)$ to $\mathbf{UR_t}$

*init*$_u$()

- Generate encryption key pair $(\mathbf{k}^{\mathbf{enc}}_u, \mathbf{k}^{\mathbf{dec}}_u) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- Replace $(u, \mathbf{N}, \mathbf{N})$ with $(u, \mathbf{k}^{\mathbf{enc}}_u, \mathbf{N})$ in $\mathbf{U_c}$

*addU*$(u)$

- Add $(u, \mathbf{N}, \mathbf{N})$ to $\mathbf{U_c}$
- Add $u$ to $\mathbf{U_t}$

*delU*$(u)$

- Delete $(u, -, -)$ from $\mathbf{U_c}$
- For every role $r$ that $u$ is a member of:
  - ∗ *revokeU*$(u, r)$
- Delete $u$ from $\mathbf{U_t}$ and $(u, -)$ from $\mathbf{UR_t}$

*addR*$(r)$

- Generate encryption key pair $(\mathbf{k}^{\mathbf{enc}}_{(r,1)}, \mathbf{k}^{\mathbf{dec}}_{(r,1)}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- Add $(r, \mathbf{k}^{\mathbf{enc}}_{(r,1)}, 1)$ to $\mathbf{R_c}$ and $(A, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_A}\left(\mathbf{k}^{\mathbf{dec}}_{(r,1)}\right), 1)$ to $\mathbf{UR_c}$
- Add $r$ to $\mathbf{R_t}$ and $(A, r)$ to $\mathbf{UR_t}$

*delR*$(r)$

- Delete $(r, -, -)$ from $\mathbf{R_c}$
- Delete all $(-, r, -, -)$ from $\mathbf{UR_c}$
- For every file $f$ that $r$ has access to:
  - ∗ *revokeP*$(r, \langle f, \{\mathsf{Sub}, \mathsf{Pub}\}\rangle)$
- Delete $r$ from $\mathbf{R_t}$ and $(-, r)$ from $\mathbf{UR_t}$

*revokeU*$(u, r)$

- Generate new role keys $(\mathbf{k}^{\mathbf{enc}}_{(r,v_r+1)}, \mathbf{k}^{\mathbf{dec}}_{(r,v_r+1)}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- For all $(u', r, -, -) \in \mathbf{UR_c} : u' \neq u$:
  - ∗ Add $(u', r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u'}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r+1)}\right), v_r + 1)$ to $\mathbf{UR_c}$
- For all $f \in \mathbf{F} : (r, f, -, -, -, op) \in \mathbf{PA_c}$
  - ∗ Generate new symmetric key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)} \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$ for $f$
  - ∗ Replace $(f, v_f)$ with $(f, v_f + 1)$ in $\mathbf{F_c}$
  - ∗ Add $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r+1)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)}\right), v_f + 1, v_r + 1, op)$ to $\mathbf{PA_c}$
  - ∗ For all $(r', f, -, -, v'_r, op') \in \mathbf{PA_c} : r' \neq r$:
    - · Add $(r', f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r',v'_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)}\right), v_f + 1, v'_r, op')$ to $\mathbf{PA_c}$
- Replace $(r, -, -)$ with $(r, \mathbf{k}^{\mathbf{enc}}_{(r,v_r+1)}, v_r + 1)$ in $\mathbf{R_c}$
- Delete all $(-, r, -, v_r)$ from $\mathbf{UR_c}$
- Delete all $(r, -, -, -, v_r, -)$ from $\mathbf{PA_c}$
- Delete $(u, r)$ from $\mathbf{UR_t}$

*assignP*$(r, \langle f, op\rangle)$

- If $r$ already has $\langle f, op'\rangle$ permission, i.e., there exists $(r, f, c, v_f, v_r, op') \in \mathbf{PA_c}$ and $(r, \langle f, op'\rangle) \in \mathbf{PA_t}$:
  - ∗ Replace $(r, f, c, v_f, v_r, op')$ with $(r, f, c, v_f, v_r, op \cup op')$ in $\mathbf{PA_c}$ and $(r, \langle f, op'\rangle)$ with $(r, \langle f, op \cup op'\rangle)$ in $\mathbf{PA_t}$
- Else:
  - ∗ Add $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_f, v_r, op)$ to $\mathbf{PA_c}$ and $(r, \langle f, op\rangle)$ to $\mathbf{PA_t}$

Fig. 3: Role-based Cryptographic Access Control for IoT Using MQTT

$\underline{\mathit{addP}(f)}$
- Generate symmetric key $\mathbf{k}^{\mathbf{sym}}_{(f,1)} \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$
- Add $(f,1)$ to $\mathbf{F_c}$, $\left(A, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{A}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,1)}\right), 1, v_A, \{\mathsf{Sub}, \mathsf{Pub}\}\right)$ to $\mathbf{PA_c}$
- The broker publishes retained message $(f,1)$ to topic $f$
- Add $(A, \langle f, \{\mathsf{Sub}, \mathsf{Pub}\}\rangle)$ to $\mathbf{PA_t}$

$\underline{\mathit{delP}(f)}$
- Delete $(f,-,-)$ from $\mathbf{F_c}$ and $(-,f,-,-,-,-)$ from $\mathbf{PA_c}$
- Delete $(-,\langle f,-\rangle)$ from $\mathbf{PA_t}$
- The broker deletes retained message $(f,-,-)$ from the topic $f$

$\underline{\mathit{assignU}(u,r)}$
- Find $(A, r, c, v_r)$ in $\mathbf{UR_c}$
- Decrypt $m = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{A}}(c)$
- Add $\left(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}(m), v_r\right)$ to $\mathbf{UR_c}$
- Add $(u,r)$ to $\mathbf{UR_t}$

$\underline{\mathit{sub}_u(f,c)}$
- When receiving $c$ on $f$ from the broker, find a role $r$ such that the following hold:
  - ∗ $u$ is in role $r$, i.e., there exists $\left(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), v_r\right)$ in $\mathbf{UR_c}$
  - ∗ $r$ has read access to the topic $f$, i.e., there exists $\left(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_f, v_r, \mathit{op}\right)$ where $\mathit{op} \cap \{\mathsf{Sub}\} \neq \emptyset$
- Decrypt role key $\mathbf{k}^{\mathbf{dec}}_{(r,v_r)} = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{u}}\left(\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right)\right)$
- Decrypt file key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)} = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}}\left((\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right)\right)$
- Decrypt message $m = \mathbf{Dec}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}}(c)$

$\underline{\mathit{revokeP}(r, \langle f, \mathit{op}\rangle)}$
- Given $(r, f, c, v_f, v_r, \mathit{op}') \in \mathbf{PA_c}$ and $(r, \langle f, \mathit{op}'\rangle) \in \mathbf{PA_t}$, if $\mathit{op}' \subseteq \mathit{op}$:
  - ∗ Delete $(r, \langle f, \mathit{op}'\rangle)$ from $\mathbf{PA_t}$ and $(r, f, c, v_f, v_r, \mathit{op}')$ from $\mathbf{PA_c}$
  - ∗ Generate new symmetric key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)} \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$
  - ∗ Replace all $(r', f, -, -, v_{r'}, \mathit{op}'')$ with $\left(r', f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r',v_{r'})}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f+1)}\right), v_f + 1, v_{r'}, \mathit{op}''\right)$ in $\mathbf{PA_c}$
  - ∗ Replace $(f,-)$ with $(f, v_f + 1)$ in $\mathbf{F_c}$
- Else if $\mathit{op} \cap \mathit{op}' \neq \emptyset$:
  - ∗ Replace $(r, \langle f, \mathit{op}'\rangle)$ with $(r, \langle f, \mathit{op}' \setminus \mathit{op}\rangle)$ in $\mathbf{PA_t}$ and $(r, f, c, v_f, v_r, \mathit{op}')$ with $(r, f, c, v_f, v_r, \mathit{op}' \setminus \mathit{op})$ in $\mathbf{PA_c}$

$\underline{\mathit{pub}_u(f,m)}$
- Find a role $r$ such that the following hold:
  - ∗ $u$ is in role $r$, i.e., there exists $\left(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), v_r\right)$ in $\mathbf{UR_c}$
  - ∗ $r$ has write access to topic $f$, i.e., there exists $\left(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_f, v_r, \mathit{op}\right)$ where $\mathit{op} \cap \{\mathsf{Pub}\} \neq \emptyset$
- Decrypt role key $\mathbf{k}^{\mathbf{dec}}_{(r,v_r)} = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{u}}\left(\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right)\right)$
- Decrypt file key $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)} = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}}\left((\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right)\right)$
- Encrypt message $c = \mathbf{Enc}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}}(m)$
- Send $c$ to the broker
- The broker receives $c$ and verifies the following:
  - ∗ $u$ is assigned to $r$, i.e., there exists $(u,r)$ in $\mathbf{UR_t}$
  - ∗ $r$ has write access to topic $f$, i.e., there exists $(r, \langle f, \mathit{op}\rangle)$ in $\mathbf{PA_t}$ so that $\mathit{op} \cap \{\mathsf{Pub}\} \neq \emptyset$
- If verification is successful, the broker sends $c$ to all clients subscribed to topic $f$

Fig. 3: Role-based Cryptographic Access Control for IoT Using MQTT