End-to-End Protection of IoT Communications Through Cryptographic Enforcement of Access Control Policies

Stefano Berlato[1,3(✉)], Umberto Morelli[3], Roberto Carbone[3], and Silvio Ranise[2,3]

[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

Published by Springer Nature Switzerland AG 2022
S. Sural and H. Lu (Eds.): DBSec 2022, LNCS 13383, pp. 236–255, 2022.
https://doi.org/10.1007/978-3-031-10684-2_14

concerns for IoT scenarios, as clearly shown by the 2021 Eclipse IoT survey.[1] 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 [7,12]. 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 and prevent possible attacks on the confidentiality and integrity of data. Indeed, these two security properties are of the utmost importance, especially in scenarios involving personal information (e.g., users' health data) or providing vital services in which integrity is crucial (e.g., smoke sensors). A popular cryptographic-based solution to protect communications by guaranteeing confidentiality and integrity is 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 [13,18]. 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;[2]
- 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.

As a final remark, we acknowledge that the use of cryptography alone to enforce AC policies makes the evaluation of permissions depending on dynamic

[1] https://outreach.eclipse.foundation/iot-edge-developer-2021.
[2] https://github.com/stfbk/CryptoAC.

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

2 Related Work

During the analysis of the large number of papers devoted to secure data in IoT, 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.

As constrained IoT devices can hardly support TLS, in [18] the authors propose an alternative security mechanism: 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). The smart card relieves resource consumption, while symmetric cryptography ensures confidentiality (but not end-to-end

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 Sect. 1 and discussed at the end of Sect. 5.2, we can easily complement our CAC scheme with traditional AC enforcement mechanisms for context awareness

[3] An extended version of this work with more details on the CAC scheme is at https://st.fbk.eu/complementary/assets/DBSEC2022/DBSEC2022_Extended.pdf.

encryption). However, the broker has to encrypt messages for each client separately, yielding a non-negligible overhead, and AC policies are 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.[4] Each client establishes a symmetric key with the broker, while topics are associated with authorization tokens. As in [18], the per-client re-encryption makes the solution hardly scalable, and the broker can read 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.

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

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

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,[5] 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[6] 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.

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 (Sect. 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 (Sect. 4.2)

[5] https://www.iso.org/standard/69466.html.
[6] https://mosquitto.org/.

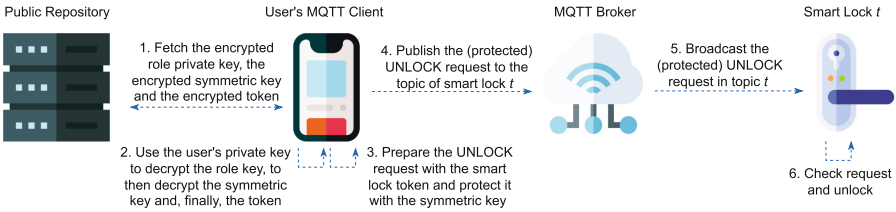

Fig. 1. Instance of an UNLOCK Request to a Smart Lock

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 Sect. 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 [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 Sect. 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 Fig. 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 Manin-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[7]). 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 (Sect. 5.1). Then, we present our CAC scheme (Sect. 5.2). Finally, we provide some considerations on the security of the scheme (Sect. 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 **U** and resources **F** of the RBAC policy, respectively. The set of roles **R** is instead defined by the administrator, as described in Sect. 4.1, thus roles are not mapped to any MQTT concept. The set of operations **OP** is composed by publish (Pub) and subscribe (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 $(\mathbf{k}_A^{\mathbf{ver}}, \mathbf{k}_A^{\mathbf{sig}})$ for verification/creation of digital signatures. Each topic f is assigned to a symmetric key $\mathbf{k}_f^{\mathbf{sym}}$, used to encrypt each message m in f, resulting in $\mathbf{Enc}^{\mathbf{S}}_{\mathbf{k}_f^{\mathbf{sym}}}(m)$.

To assign a user u to a role r, r's decryption key $\mathbf{k}_r^{\mathbf{dec}}$ is encrypted with u's encryption public key $\mathbf{k}_u^{\mathbf{enc}}$, resulting in $\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_u^{\mathbf{enc}}}\left(\mathbf{k}_r^{\mathbf{dec}}\right)$. 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 $\mathbf{k}_f^{\mathbf{sym}}$ is encrypted with r's encryption public key $\mathbf{k}_r^{\mathbf{enc}}$, resulting in $\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_r^{\mathbf{enc}}}\left(\mathbf{k}_f^{\mathbf{sym}}\right)$.

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

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

5.2 Full Construction

As illustrated in Sect. 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

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 {Sub}, {Pub} or {Sub, Pub}
$\mathbf{N}$	Null (i.e., empty) value
m	A generic plaintext
c	A generic ciphertext
–	Wildcard
$\mathbf{Gen^{Pub}}$	Generation of key pair for en/decryption
$\mathbf{Gen^{Sig}}$	Generation of key pair for signatures
$\mathbf{Gen^{Sym}}$	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 $(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_u}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), v_r)$
$\mathbf{PA_c}$	Set of role-permission pairs; a member is a tuple $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_r, v_A, op)$

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). We report in Fig. 2 (Appendix A) 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. An action α in Fig. 2 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 ($initA()$). Then, she can add and delete users ($addU(u)$, $delU(u)$), roles ($addR(r)$, $delR(r)$) and topics ($addP(f)$, $delP(f)$). Finally, she can assign and revoke users from roles ($assignU(u, r)$, $revokeU(u, r)$) as well as assign and revoke permissions from roles ($assignP(r, \langle f, op \rangle)$, $revokeP(r, \langle f, op \rangle)$);
- *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 ($init_u()$). Afterwards, she can subscribe to a topic f ($sub_u(f, c)$) and also publish messages ($pub_u(f, m)$), according to the policy defined by the administrator. Messages received from a topic are en/decrypted as described in Sect. 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 Fig. 2 we omit these details and also other trivial checks like the uniqueness of identifiers.

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 Sect. 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 Sect. 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

The pseudocode presented in Appendix A has been paired with an MQTT client and deployed as standalone, open-source software named *"CryptoAC"*.[8] Altogether, the CAC scheme involves three entities: *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 (Sects. 6.1, 6.2, 6.3). Finally, we present our preliminary performance evaluation of *CryptoAC* (Sect. 6.4).

6.1 CryptoAC

CryptoAC has been developed in Kotlin multiplatform[9] due to the intrinsic portability and the possibility of a native deployment avoiding the computational overhead of a Java Virtual Machine (JVM). This is especially true since Kotlin mainly targets Linux environments,[10] which are the most deployed in IoT.[11]

As the cryptographic provider, we choose Sodium,[12] a modern cryptographic library whose security has been thoroughly audited.[13] To invoke Sodium's APIs easily, we use an open-source Kotlin multiplatform wrapper.[14] Sodium uses the Elliptic Curves Diffie-Hellman (ECDH) algorithm (X25519) to generate public-private keys and the Edwards-curve Digital Signature Algorithm (EdDSA) for digital signatures (Ed25519). Like many cyphers in TLS 1.3, Sodium supports (AEAD), which is a more robust and secure variant of authenticated encryption (recall the discussion in Sect. 5.2) allowing to bind the ciphertext to the context where it is supposed to be used (to, e.g., avoid replay attacks). We observe

[8] The code is freely available at https://github.com/stfbk/CryptoAC.
[9] https://kotlinlang.org/docs/multiplatform.html.
[10] https://www.jetbrains.com/lp/devecosystem-2019/.
[11] https://outreach.eclipse.foundation/iot-edge-developer-2021.
[12] https://libsodium.gitbook.io/doc/.
[13] https://www.privateinternetaccess.com/blog/libsodium-audit-results/.
[14] https://github.com/ionspin/kotlin-multiplatform-libsodium.

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.[15]

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 Sect. 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 Eclipse Paho[16] as MQTT client. It is worth noting that *CryptoAC* caches symmetric keys of topics at the client-side to avoid having to obtain them (as described in Sect. 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. 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 allowing the administrator to manage the (traditional and cryptographic) AC policy. All the inputs to the interface are validated with OWASP-approved regular expressions[17] to avoid web-based attacks (e.g., injection, Cross-Site Scripting).

6.2 MQTT Broker

We choose Mosquitto[18] as MQTT broker. As introduced in Sect. 5.2, we enable the DYNSEC plugin for traditional AC enforcement on top of CAC. This additional security layer guarantees redundancy and allows restricting 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

[15] https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics.
[16] https://www.eclipse.org/paho/.
[17] https://owasp.org/www-community/OWASP_Validation_Regex_Repository.
[18] https://mosquitto.org/.

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

We use Redis[19] to store metadata related to the CAC scheme. Redis is primarily an in-memory storage, a characteristic that allows for low response time to queries. 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 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; therefore, we distribute the computation at the edge nodes and achieve scalability by design.[20] Consequently, our preliminary evaluation aims at assessing the computational overhead on a single IoT device, where the cryptographic operations of CAC could strongly affect performance. In detail, we consider the following experimental configurations:

- *C1*: in this configuration, our baseline, 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;
- *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 the partially-trusted service provider 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

[19] https://redis.io/.

[20] Increasing the number of clients would only assess the scalability of the MQTT broker since the encryption/decryption are performed client-side.

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 [13,18]. 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 Sect. 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, using another MQTT client (e.g., Mosquitto_sub[21] and Mosquitto_pub[22]) 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 configurations. 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 @ 3.40 GHz) as CPU and 16 GB of RAM. A Raspberry Pi 3 Model B+ (Cortex-A53 ARMv8 64-bit SoC @ 1.4 GHz with 1 GB 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 Fig. 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: the average of *C1* is 11.1 ms, *C2* is 11.8 ms and *C3* is 13.5 ms;[23]

[21] https://mosquitto.org/man/mosquitto_sub-1.html.
[22] https://mosquitto.org/man/mosquitto_pub-1.html.
[23] https://st.fbk.eu/complementary/assets/DBSEC2022/experimental_results.xlsx.

As expected, the baseline *C1* has the lowest average transmission time due to avoiding cryptographic operations. Once removed the burden of the TLS handshake and key derivation procedure, *C2* incurs 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.[24] The use of CAC in *C3* yields an average overhead with respect to *C1* and *C2* of 2.4 ms and 1.7 ms, respectively. We believe that this is an acceptable overhead in a Smart Lock service, especially when considering the greater security guarantees offered by CAC. We also believe we can reduce this overhead by optimizing the code of *CryptoAC* and fine-tuning the parameters of the cryptographic algorithms employed. We leave the validation 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 Sect. 5.1 (configuration *C3C*). The results show that in *C3B* there is a negligible improvement of 0.1 ms on average, an indicator that DYNSEC does not have a significant impact on the performance of Mosquitto. The average transmission time on *C3C* is 20.9 ms on average, 7.4 ms 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 MQTT 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 1.7 ms with respect to a scenario employing TLS (but without considering the handshake and key derivation algorithms), and 2.4 ms 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 offered 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.

Acknowledgements. This work has been partially supported by "Futuro & Conoscenza Srl", jointly created by the FBK and the Italian National Mint and Printing House (IPZS), Italy.

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

A Pseudocode of the Cryptographic Access Control Scheme

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

init$_u$()
- Generate encryption key pair $(\mathbf{k}_u^{\mathbf{enc}}, \mathbf{k}_u^{\mathbf{dec}}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- Replace $(u, \mathbf{N}, \mathbf{N})$ with $(u, \mathbf{k}_u^{\mathbf{enc}}, \mathbf{N})$ 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 $(\mathbf{k}_{(r,1)}^{\mathbf{enc}}, \mathbf{k}_{(r,1)}^{\mathbf{dec}}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- Add $(r, \mathbf{k}_{(r,1)}^{\mathbf{enc}}, 1)$ to $\mathbf{R_c}$ and $(A, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_A^{\mathbf{enc}}}(\mathbf{k}_{(r,1)}^{\mathbf{dec}}), 1)$ 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 $(\mathbf{k}_{(r,v_r+1)}^{\mathbf{enc}}, \mathbf{k}_{(r,v_r+1)}^{\mathbf{dec}}) \leftarrow \mathbf{Gen}^{\mathbf{Pub}}$
- For all $(u', r, -, -) \in \mathbf{UR_c} : u' \neq u$:
  - Add $(u', r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_{u'}^{\mathbf{enc}}}(\mathbf{k}_{(r,v_r+1)}^{\mathbf{dec}}), v_r + 1)$ to $\mathbf{UR_c}$
- For all $f \in \mathbf{F} : (r, f, -, -, -, op) \in \mathbf{PA_c}$
  - Generate new symmetric key $\mathbf{k}_{(f,v_f+1)}^{\mathbf{sym}} \leftarrow \mathbf{Gen}^{\mathbf{Sym}}$ for f
  - Replace (f, v_f) with $(f, v_f + 1)$ in $\mathbf{F_c}$
  - Add $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_{(r,v_r+1)}^{\mathbf{enc}}}(\mathbf{k}_{(f,v_f+1)}^{\mathbf{sym}}), v_f + 1, v_r + 1, op)$ to $\mathbf{PA_c}$
  - For all $(r', f, -, -, v_r', op') \in \mathbf{PA_c} : r' \neq r$:
    - Add $(r', f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_{(r',v_r')}^{\mathbf{enc}}}(\mathbf{k}_{(f,v_f+1)}^{\mathbf{sym}}), v_f + 1, v_r', op')$ to $\mathbf{PA_c}$
- Replace $(r, -, -)$ with $(r, \mathbf{k}_{(r,v_r+1)}^{\mathbf{enc}}, v_r + 1)$ 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 $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}_{(r,v_r)}^{\mathbf{enc}}}(\mathbf{k}_{(f,v_f)}^{\mathbf{sym}}), v_f, v_r, op)$ to $\mathbf{PA_c}$ and $(r, \langle f, op\rangle)$ to $\mathbf{PA_t}$

Fig. 2. Role-based Cryptographic Access Control for IoT Using MQTT

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

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

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 $(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}(m), v_r)$ to $\mathbf{UR_c}$
- Add (u,r) to $\mathbf{UR_t}$

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 $(u, r, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{u}}\left(\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}\right), v_r)$ in $\mathbf{UR_c}$
  - r has read access to the topic f, i.e., there exists $(r, f, \mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right), v_f, v_r, op)$ where $op \cap \{\mathsf{Sub}\} \neq \emptyset$
- 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 $\mathbf{k}^{\mathbf{sym}}_{(f,v_f)} = \mathbf{Dec}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{dec}}_{(r,v_r)}}\left(\left(\mathbf{Enc}^{\mathbf{P}}_{\mathbf{k}^{\mathbf{enc}}_{(r,v_r)}}\left(\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}\right)\right)\right)$
- Decrypt message $m = \mathbf{Dec}^{\mathbf{S}}_{\mathbf{k}^{\mathbf{sym}}_{(f,v_f)}}(c)$

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

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

Fig. 2. (*continued*)

References

1. Ahmad, T., Morelli, U., Ranise, S.: 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, pp. 213–220. ACM (2020)
2. Ahmad, T., Morelli, U., Ranise, S., Zannone, N.: 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 2018, pp. 235–246. Association for Computing Machinery, New York (2018)
3. Ahmad, T., Morelli, U., Ranise, S., Zannone, N.: Extending access control in AWS IoT through event-driven functions: an experimental evaluation using a smart lock system. Int. J. Inf. Secur. **21**(2), 379–408 (2021)
4. Armando, A., Grasso, M., Oudkerk, S., Ranise, S., Wrona, K.: Content-based information protection and release in NATO operations. In: Proceedings of the 18th ACM Symposium on Access Control Models and Technologies - SACMAT 2013, p. 261. ACM Press (2013)
5. Armando, A., Oudkerk, S., Ranise, S., Wrona, K.: Formal modelling of content-based protection and release for access control in NATO operations. In: Danger, J.-L., Debbabi, M., Marion, J.-Y., Garcia-Alfaro, J., Zincir Heywood, N. (eds.) FPS-2013. LNCS, vol. 8352, pp. 227–244. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05302-8_14
6. Bellare, M., Namprempre, C.: Authenticated encryption: relations among notions and analysis of the generic composition paradigm. J. Cryptol. **21**(4), 469–491 (2008)
7. Berlato, S., Carbone, R., Lee, A.J., Ranise, S.: Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. ACM Trans. Priv. Secur. **25**(1), 1–37 (2021)
8. Calabretta, M., Pecori, R., Veltri, L.: A token-based protocol for securing MQTT communications. In: 2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–6. IEEE (2018)
9. Colombo, P., Ferrari, E.: Access control enforcement within MQTT-based internet of things ecosystems. In: Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, pp. 223–234. ACM (2018)
10. Djoko, J.B., Lange, J., Lee, A.J.: 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), pp. 401–413. IEEE (2019)
11. Elemam, E., Bahaa-Eldin, A.M., Shaker, N.H., Sobh, M.A.: A secure MQTT protocol, telemedicine IoT case study. In: 2019 14th International Conference on Computer Engineering and Systems (ICCES), pp. 99–105. IEEE (2019)
12. Garrison, W.C., Shull, A., Myers, S., Lee, A.J.: On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 819–838 (2016)
13. Heer, T., Garcia-Morchon, O., Hummen, R., Keoh, S.L., Kumar, S.S., Wehrle, K.: Security challenges in the IP-based internet of things. Wirel. Pers. Commun. **61**(3), 527–542 (2011)
14. Kurnikov, A., Paverd, A., Mannan, M., Asokan, N.: Keys in the clouds: auditable multi-device access to cryptographic credentials. In: Proceedings of the 13th International Conference on Availability, Reliability and Security, pp. 1–10. ACM (2018)

15. Malina, L., Srivastava, G., Dzurenda, P., Hajny, J., Fujdiak, R.: A secure publish/-subscribe protocol for internet of things. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, pp. 1–10. ACM (2019)
16. Palmieri, A., Prem, P., Ranise, S., Morelli, U., Ahmad, T.: MQTTSA: a tool for automatically assisting the secure deployments of MQTT brokers. In: 2019 IEEE World Congress on Services (SERVICES), vol. 2642–939X, pp. 47–53 (2019)
17. Samarati, P., de Vimercati, S.C.: Access control: policies, models, and mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 137–196. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45608-2_3
18. Sanjuan, E.B., Cardiel, I.A., Cerrada, J.A., Cerrada, C.: Message queuing telemetry transport (MQTT) security: a cryptographic smart card approach. IEEE Access **8**, 115051–115062 (2020)
19. Segarra, C., Delgado-Gonzalo, R., Schiavoni, V.: MQT-TZ: hardening IoT brokers using ARM TrustZone: (practical experience report). In: 2020 International Symposium on Reliable Distributed Systems (SRDS), pp. 256–265. IEEE (2020)
20. Zeadally, S., Das, A.K., Sklavos, N.: Cryptographic technologies and protocol standards for internet of things. Internet Things **14**, 100075 (2019)