International Journal of Information Security (2024) 23:3109–3133
https://doi.org/10.1007/s10207-024-00877-1

REGULAR CONTRIBUTION

Using the ACE framework to enforce access and usage control with notifications of revoked access rights

Marco Rasori[1] · **Andrea Saracino**[1,2] · **Paolo Mori**[1] · **Marco Tiloca**[3]

Published online: 8 July 2024
© The Author(s) 2024

Abstract

The standard ACE framework provides authentication and authorization mechanisms similar to those of the standard OAuth 2.0 framework, but it is intended for use in Internet-of-Things environments. In particular, ACE relies on OAuth 2.0, CoAP, CBOR, and COSE as its core building blocks. In ACE, a non-constrained entity called Authorization Server issues Access Tokens to Clients according to some access control and policy evaluation mechanism. An Access Token is then consumed by a Resource Server, which verifies the Access Token and lets the Client accordingly access a protected resource it hosts. Access Tokens have a validity which is limited over time, but they can also be revoked by the Authorization Server before they expire. In this work, we propose the Usage Control framework as an underlying access control means for the ACE Authorization Server, and we assess its performance in terms of time required to issue and revoke Access Tokens. Moreover, we implement and evaluate a method relying on the Observe extension for CoAP, which allows to notify Clients and Resource Servers about revoked Access Tokens. Through results obtained in a real testbed, we show how this method reduces the duration of illegitimate access to protected resources following the revocation of an Access Token, as well as the time spent by Clients and Resource Servers to learn about their Access Tokens being revoked.

Keywords Access control · Usage control · ACE framework · CoAP · Observe · Internet of Things · Revocation

1 Introduction

The Internet of Things (IoT)[1] is becoming increasingly pervasive in today's networked environments and systems. This has been fostering the development of applications and services for several use cases, ranging from home and building automation, to monitoring and control of production systems and critical infrastructure. In such contexts, enforcing access control is one of the key security requirements to fulfill. That is, it is vital to effectively and efficiently control the rights to perform specific operations or to access resources at IoT devices. A number of frameworks have been proposed in order to manage access control in different IoT environments [1, 2]. More recently, the new *Authentication and Authorization for Constrained Environments (ACE)* framework [3] has also been standardized.

The ACE framework is based on the well-known Open Authorization (OAuth) 2.0 framework [4], and enables its functionalities in constrained environments. In particular, the ACE framework relies on the lightweight *Constrained Application Protocol (CoAP)* [5] application-layer web-transfer protocol, on *Concise Binary Object Representation (CBOR)* for data encoding [6], and on *CBOR Object Signing and Encryption (COSE)* [7, 8] for data encryption and authentication.

✉ Marco Rasori
marco.rasori@iit.cnr.it

Andrea Saracino
andrea.saracino@santannapisa.it

Paolo Mori
paolo.mori@iit.cnr.it

Marco Tiloca
marco.tiloca@ri.se

1 Institute of Informatics and Telematics, National Research Council, Via Giuseppe Moruzzi 1, Pisa, Italy

2 Department of Excellence in Robotics, AI, Scuola Superiore Sant'Anna, Via Giuseppe Moruzzi 1, Pisa, Italy

3 RISE Cybersecurity, RISE Research Institutes of Sweden AB, Isafjordsgatan 22, Kista, Sweden

[1] Appendix A provides the full list of the abbreviations used throughout this paper.

In ACE, an *Authorization Server (AS)* acts on behalf of resource owners and can give *Clients* access grants to access protected resources hosted at *Resource Servers*. These grants are in the form of *Access Tokens*, and a Client proves its rights to access protected resources by providing the corresponding Resource Server with an Access Token. The Access Tokens are issued following the evaluation of access policies, and, according to the ACE specification, the AS is not devoted to any specific mechanism for carrying out policy evaluation.

The validity of an Access Token has an expiration time, which is set by the AS at issuing time; a Resource Server does not accept expired Access Tokens and expunges Access Tokens when they eventually expire. However, due to various reasons, the AS might want to revoke an Access Token before its expiration time comes. This might be the case, for example: (i) when a Resource Server has been compromised, or it is suspected of being compromised; (ii) when there has been a change in the Client's access rights, as possibly determined by dynamic conditions in the execution environment, the user context, or the resource utilization.

Currently, the ACE framework does not provide a mechanism for the AS to deliberately inform the interested parties that an Access Token has been revoked, but the AS can optionally make available an */introspect* endpoint to be queried by Resource Servers, in order to verify the validity of a specific Access Token. However, this mechanism is not available to Clients, and, more importantly, it is not proactive. That is, if an Access Token is revoked, its revocation may remain unnoticed by the Resource Server until the latter requests the AS to "introspect" it, i.e., to verify the validity of that specific Access Token.

A recent proposal [9] aims at extending the ACE framework with a mechanism that allows Clients and Resource Servers to subscribe to a revocation list at the AS, and to be notified when Access Tokens pertaining to them are added to or removed from such a list. This mechanism relies on the Observe extension [10] for CoAP to proactively notify observing parties about occurred revocations of Access Tokens. At the same time, it also allows both Clients and Resource Servers to retrieve the revocation list on demand, by making single GET requests to a dedicated endpoint at the AS, namely */trl*.

In this work, we propose and evaluate the adoption of the *Usage Control* framework [11] as an underlying access control tool for the ACE Authorization Server. Usage Control offers a significant enhancement over traditional access control. Indeed, through Usage Control, it is possible to define, evaluate, and enforce security policies that take into account attributes whose value can change over time, thus introducing the possibility of rapidly revoking the right to access a resource by taking appropriate countermeasures. In the context of the ACE framework, this means revoking issued Access Tokens that are not expired yet. Information about occurred revocations, however, has to be communicated to Clients and Resource Servers, in order for them to stop relying on a revoked Access Token.

The adoption of the Usage Control framework as an access control tool for the ACE Authorization Server considerably improves the security posture of IoT deployments, and displays the following benefits: (i) it allows defining access control policies that take into account dynamic factors in the decision process, namely, the values of attributes that can change over time; and (ii) thanks to the continuous monitoring of such attributes, it automatically detects if an Access Token must be revoked. This tool, combined with the notification mechanism described in [9], minimizes the time for Clients and Resource Servers to learn about revocation of an Access Token, and, consequently, the duration of illegitimate accesses to protected IoT resources following the revocation.

Summarizing, the main contributions of this paper are the following:

- We propose the exploitation of the Usage Control framework as underlying access control tool for the ACE framework, and we explain how to integrate it in the Authorization Server. This enables the Authorization Server to issue and revoke Access Tokens considering a dynamic access context, where the environmental attributes, as well as Clients' and resources' attributes, change over time.
- We develop the notification mechanism proposed in [9] and the integration with the Usage Control framework, and accordingly extend an existing Java implementation[2] of the ACE framework.
- We deploy a real testbed and conduct a performance evaluation, by using our software implementation of the extended ACE framework.[3] During our experiments, we measure and compare the time performance displayed by the different mechanisms for disseminating information about revoked Access Tokens to the affected Clients and Resource Servers. Our results show that the on-demand, polling-based acquisition of the revocation list as per [9] is more effective and efficient than the original ACE introspection mechanism. Moreover, our results show that, when relying on the Observe extension for CoAP, the acquisition of the revocation list is faster and more efficient than its on-demand alternative. In addition, the time performance of operations specifically involving and carried out by the Usage Control framework is also evaluated and discussed.

[2] https://bitbucket.org/marco-tiloca-sics/ace-java/src/master/.

[3] https://bitbucket.org/marco-rasori-iit/ace-java/src/ucs/.

To the best of our knowledge, this is the first contribution that integrates into the ACE framework and evaluates a concrete, advanced access control engine and a mechanism to inform about revoked Access Tokens.

The rest of the paper is organized as follows. Section 2 reports background concepts on the Usage Control framework and on the ACE framework. Section 3 focuses on presenting cases in which a token should be revoked and describes the revoked token notification mechanism implemented in this work. Section 4 explains how the Usage Control framework has been integrated into the ACE Authorization Server. Section 5 describes the experimental methods, and Sect. 6 discusses the obtained results. Section 7 discusses related work. Finally, Sect. 8 draws our conclusive remarks.

2 Background

This section introduces the background and the concepts used in the rest of the paper.

2.1 The usage control framework

The *Usage Control (UCON) framework* regulates the exercise of rights on resources by subjects following the UCON model [12]. In particular, the UCON framework proposes an architectural layer that extends the eXtensible Access Control Markup Language (XACML) reference architecture and language [13] (standardized by OASIS). The UCON framework includes, in fact, the components required for (i) the evaluation of *Usage Control Policies* (UCPs) compliant with the UCON model's decision factors; (ii) the continuous monitoring and revocation of ongoing usages; and (iii) the management of mutable attributes.

The main difference between traditional access control and UCON is that the former verifies that a subject has the rights to access a resource only at the time when the subject's request is evaluated. Instead, the latter extends this behavior by continuously monitoring that the rights hold for all the duration of the access. The access grant might be revoked due to a change in the value of the subject's, resource's, or environment's attributes that are specified in the pertinent access policies. For example, a subject is allowed to perform an operation in a room only if and as long as the room temperature is between 10 and 35 degrees. Traditional access control cannot express this requirement: it could only allow the subject to perform the operation if the room temperature is in the allowed range at the time of the request. On the contrary, UCON monitors that the condition holds both at the time when the subject's request is evaluated as well as throughout the lifetime of an access grant. If, during the lifetime of the access grant, the room temperature goes out of range, the grant to perform the operation is revoked.

The UCON framework specifies access and usage strategies by means of usage control policies. A UCP is an XACML-based policy written in the U-XACML language [11], which can be evaluated against a *UCON request*, encoded in XACML. The U-XACML language is an extension of the XACML language, which is completely compatible with the OASIS standard and includes time-related tags to enable dynamic policy evaluation. The evaluation produces an *authorization decision*, which can take either the value Permit, if the policy is satisfied by the UCON request, or Deny otherwise. A UCP is composed of three different sections, i.e., pre-, ongoing-, and post-sections, which are evaluated separately and at different times. Each section is evaluated against the same UCON request. A section contains a *condition*, i.e., a Boolean formula over some attributes, e.g., (`attr1` = `value1`) AND (`attr2` ≠ `value2`). Evaluating a section means evaluating its condition.[4] Please note that, in this paper, we use an XACML-like notation, where the term *condition* refers to a Boolean formula involving attributes related to subjects, resources, and environment.

The pre-section is evaluated at the time of the request, and the evaluation produces a *pre-decision*. If the result of the evaluation is Permit, the subject is provided with an access grant for the specified resource. Then, the ongoing-section is evaluated, and the evaluation produces an *ongoing-decision*. If it is Permit, the UCP is continuously monitored in the context of the ongoing-section. The ongoing-section contains *mutable attributes*, i.e., attributes whose value can change over time. When the value of a mutable attribute changes, a *policy re-evaluation* is performed to verify whether the access grant is still legit or should be revoked, i.e., whether the ongoing-section is still satisfied or not after the attribute value change. Finally, when the access grant is terminated, the post-section is evaluated.

In order to represent the life cycle of access grants, the UCON framework relies on the concept of *session*. Each session is identified by a unique *session identifier* and has a *status*, whose value is set to: `TRY_ACCESS` when the pre-decision returns Permit; `START_ACCESS` when the ongoing-decision returns Permit; and `REVOKE_ACCESS` when the ongoing-decision returns Deny. Sessions whose status is `START_ACCESS` are called *ongoing usage sessions*.

The *Usage Control System (UCS)* is the core of the UCON framework. Figure 1 shows the UCON framework architecture and its main components, which are described in the following.

[4] Also obligations are defined in the UCON framework but, for the sake of simplicity, they are not discussed here since they are not relevant to the scope of this work.

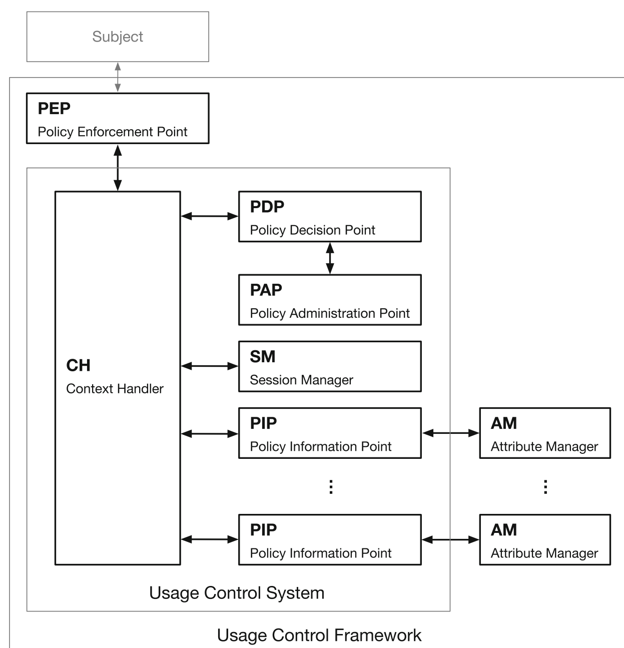

Fig. 1 UCON framework architecture

The **Policy Enforcement Point (PEP)** actually enforces the access right to a resource, either granting or not the access according to the UCS evaluation. The PEP intercepts the access attempts from subjects and generates UCON requests, starting the authorization workflows by issuing towards the UCS three different types of messages: `tryAccess` to request access with static conditions, `startAccess` to complete the previous request with dynamic conditions, and `endAccess` to communicate the end of the action on the resource. In the UCON framework, the PEP is also able to dynamically revoke access rights, thus making it possible to block ongoing resource accesses. To this end, the PEP is able to handle and enforce `revokeAccess` requests issued by the UCS.

The **Policy Administration Point (PAP)** component manages and stores the UCPs.

The **Policy Decision Point (PDP)** component evaluates a UCON request against a section of a UCP. When serving a `tryAccess` message, the PDP is also responsible for finding an *applicable* UCP $UCP^{\star}$ among those stored at the PAP to be used for evaluation. An applicable UCP is a policy whose XACML `<Target>` field matches the UCON request, and is therefore such that the UCON request can be evaluated against it. Both the UCON request and the access policy (either the pre-, ongoing-, or post-section) are expressed in XACML format, thus the PDP can use a standard XACML engine, such as WSO2 Balana,[5] for evaluation.

[5] https://github.com/wso2/balana.

The **Attribute Managers (AMs)** are external components that handle subjects', resources', and environmental attributes. An AM offers an interface from which it is possible to simply query or subscribe to a specific attribute. AMs are truth points for the attribute values, i.e., the current value of an attribute is always immediately available to its AM. Examples of AMs can be local and remote databases, a file stored on the file system, a resource reachable at a URL, or an Identity Provider controlling users' information such as nationality or age.

The **Policy Information Points (PIPs)** are adapters situated between the Context Handler (CH)—the component coordinating the evaluation process—and the AMs, and their duty is to provide the CH with fresh attribute values. They all offer a common interface to the CH, while the interface with the AMs is PIP specific. The PIP-CH interface consists of four methods: (i) *retrieve*, (ii) *subscribe*, (iii) *unsubscribe*, and (iv) *update*. The *retrieve* method is invoked by the CH to obtain current attribute values for the attributes that the PIP is responsible for. When calling this method, the CH can specify a value, e.g., an identity number, that the PIP uses to query the AM in order to obtain the corresponding attribute value. With reference to the previous example, the PIP could send the identity number to the AM, which returns the current value of the attribute "age" associated with that identity number.

The *subscribe* method is invoked by the CH to get notified when an attribute value changes. When a PIP receives a subscription request, it starts a continuous monitoring of the attribute at the AM. The way in which the PIP retrieves an attribute value is clearly dependent on the specific AM it is attached to. Examples are resource polling at the AM, subscription to a publish-subscribe topic resource at the AM, and resource observation through the Observe extension of CoAP [10, 14]. As soon as the attribute value changes, the PIP notifies the CH, which performs a policy re-evaluation.

The *unsubscribe* method is invoked by the CH to stop receiving notifications from a PIP. When a PIP receives a request for subscription cancellation, it stops the attribute monitoring.

Additionally, the *update* method is also available to be invoked by the CH to change the value of an attribute at the AM. When a PIP receives an update request, it asks the AM to update the attribute value with the one provided by the CH.

The **Session Manager (SM)** component keeps track of and administers the life cycle of usage control sessions. The information stored within each session includes the session identifier, the status of the access, the UCON request as arrived from the PEP, and a UCP to be used for evaluation against the UCON request.

The **Context Handler (CH)** component interacts with the PEP according to the protocol shown in Fig. 2 and coordinates the process of evaluation of UCON requests.

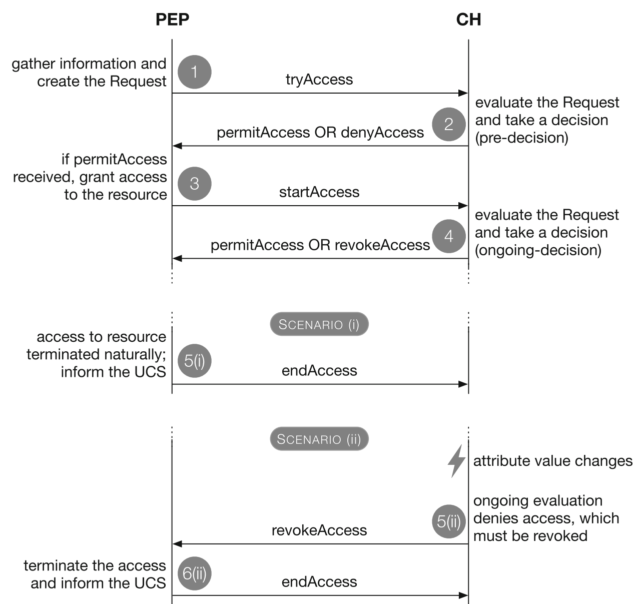

Fig. 2 PEP-CH interaction

The PEP issues a `tryAccess` message when a subject s wants to perform an action a on a resource r. First, the PEP generates the UCON request including the minimum information needed to identify the subject s, the resource r, and the action a. Thus, a `tryAccess` message containing the UCON request is sent to the CH for evaluation (step 1 in Fig. 2). The CH extracts the UCON request, invokes the retrieve method at all the PIPs, and builds an *enriched UCON request* by adding these attributes and their values to the original UCON request. Next, the CH asks the PDP to find an applicable UCP $UCP^{\star}$ to be used for evaluation. When $UCP^{\star}$ is found, its pre-section is extracted, and the PDP evaluates it against the enriched UCON request. If the result of the pre-decision is Deny, the CH sends a `denyAccess` message to the PEP. On the contrary, if the result is Permit, the CH communicates to the SM that a new session must be created. The information saved within the session includes the session identifier $SID^{\star}$, the original UCON request, $UCP^{\star}$, and the status of the access, which in this case is `TRY_ACCESS`. Then, the CH includes the session identifier $SID^{\star}$ in a `permitAccess` message and sends it to the PEP (step 2 of Fig. 2). Finally, the PEP authorizes the subject s to access the resource r and perform the action a on it.

Upon receiving a `permitAccess` message following a `tryAccess` message, the PEP grants the subject access to the resource. Next, it sends a `startAccess` message containing the session identifier $SID^{\star}$ to the CH (step 3 of Fig. 2). By querying the SM with $SID^{\star}$, the CH retrieves the original UCON request and $UCP^{\star}$, enriches the UCON request with the current attributes' values, and provides the PDP with the enriched UCON request and with the ongoing-section of $UCP^{\star}$. The PDP performs the evaluation and produces an ongoing-decision. If the result of the ongoing-decision is Deny, the CH asks the SM to update the session with the status `REVOKE_ACCESS`. Then, the CH sends a `revokeAccess` message to the PEP, which determines that the access to the resource is not legit anymore for the subject. On the contrary, if the result is Permit, the CH asks the SM to update the session with the status `START_ACCESS`. From that moment on, the mutable attributes in the ongoing-section of $UCP^{\star}$ are continuously monitored: the CH subscribes to the pertaining PIPs, which will notify it in the event of attribute value change. Finally, the CH includes the session identifier $SID^{\star}$ in a `permitAccess` message and sends it to the PEP (step 4 of Fig. 2).

When the value of an attribute present in the ongoing-section of $UCP^{\star}$ changes, the CH starts a policy re-evaluation. The CH enriches the original UCON request and provides the PDP with the enriched UCON request and with the ongoing-section of $UCP^{\star}$. The PDP performs the evaluation and produces an ongoing-decision. If the result of the ongoing-decision is Deny, the CH asks the SM to update the session with the status `REVOKE_ACCESS` and sends a `revokeAccess` message to the PEP (step 5(ii) of Fig. 2). On the contrary, if the result is Permit, the CH determines that the access grant is still legit and performs no further actions.

After the access to the resource has terminated, the PEP sends an `endAccess` message containing the session identifier $SID^{\star}$ to the CH. An access can be terminated for two different reasons: (i) the access has naturally ended (step 5(i) of Fig. 2), or (ii) the PEP received a `revokeAccess` message from the CH (step 5(ii) of Fig. 2). In the latter case, upon receiving the `revokeAccess` message, the PEP first undertakes actions to terminate the access and then informs the CH through an `endAccess` message (step 6(ii) of Fig. 2). When the CH receives an `endAccess` message, it asks the SM to delete the session with session identifier $SID^{\star}$.

2.2 CoAP

The Constrained Application Protocol (CoAP) [5] is an application-layer, web-transfer protocol based on the Representational State Transfer (REST) paradigm [15], and is now a de-facto standard application-layer protocol for the IoT. CoAP is designed to support applications for resource-constrained devices and networks, with the aim of integrating massive IoT in the existing Internet infrastructure. Although support for additional transports has been defined, CoAP typically runs on top of the unreliable transport protocol UDP [16]. Also, CoAP is not session-based and can handle loss or delayed delivery of messages.

IoT-based network deployments may include devices with limited resources in terms of memory, computing power,

and energy (if battery powered). This results in constrained network segments, e.g., due to lossy channels and limited bandwidth. In order to deal with such limitations, CoAP features an asynchronous messaging model and has native support for intermediary proxies.

Being a RESTful protocol, CoAP considers a client and a server as communicating parties, where the client sends a request to the server, which replies by sending a response. Depending on the operation to perform, a CoAP request has one of the different REST methods, e.g., GET, PUT, POST, FETCH, PATCH/iPATCH, and DELETE.

A CoAP message is divided into header and payload. The header can include a number of CoAP *options*, specified according to a Type-Length-Value format and used to control additional features and extensions. For example, CoAP options can be used to instruct a proxy on how to handle messages, specify for how long a message is valid, or signal message fragmentation at the application layer.

A number of extensions for CoAP have been defined over the years. In particular, the Observe extension defined in [10] allows a CoAP client to "subscribe" for updates to a resource representation at a CoAP server. That is, the client sends a first request targeting a resource at the server that it is interested in observing, including a CoAP Observe option in the request. Following a first response where the server confirms that an observation has indeed started, the server will additionally send further responses, namely *notifications*, to the observing client, when the resource representation changes. All such notifications sent by the server will match the same original observation request.

The original CoAP specification [5] indicates only the Datagram Transport Layer Security (DTLS) 1.2 [17] protocol to secure message exchanges. More recently, the security protocol Object Security for Constrained RESTful Environments (OSCORE) [18] has been standardized to provide end-to-end security of CoAP messages at the application layer, as further discussed in Sect. 2.3.

2.3 OSCORE

Object Security for Constrained RESTful Environments (OSCORE) is a standard security protocol [18] for protecting CoAP messages at the application layer. OSCORE provides end-to-end security between the original producer and the final consumer of the data conveyed in a CoAP message. In particular, OSCORE provides *end-to-end* encryption, integrity protection, source authentication, and replay protection of CoAP messages.

Instead of protecting the whole communication channel between a CoAP client and a CoAP server, OSCORE is CoAP-aware and consistently encrypts only the parts of the CoAP message that require confidentiality. At the same time, any field meant to be used by proxies is left unprotected, or only integrity protected. OSCORE results in smaller power consumption and memory burden on constrained devices when compared to channel-security protection provided by the DTLS protocol [19].

Intuitively, OSCORE takes a CoAP message as input and produces as output a new protected CoAP message, i.e., an OSCORE message. The reverse process occurs when an OSCORE message is received, and the original CoAP message is recomputed. Also, OSCORE is independent of the specific transport layer underlying CoAP, hence it works wherever CoAP works. Furthermore, it is possible to combine OSCORE with communication security on other layers, e.g., to further protect an OSCORE-protected message using DTLS at the transport layer.

The lightweight design of OSCORE in turn relies on the efficient and small-size encoding scheme CBOR [6]. In particular, the specific data to be protected composes a CBOR structure, which is then encrypted and authenticated by using COSE [7, 8]. This yields a COSE object, which is finally carried out in a compact way within the OSCORE message. Thus, OSCORE follows the *object security* paradigm, with each data chunk secured separately.

Before two CoAP nodes can exchange secure data, they have to establish a shared *OSCORE Security Context*. However, as focused only on message protection, OSCORE itself does not provide a mechanism to do so. Instead, a number of methods have been developed to let two CoAP nodes establish an OSCORE Security Context. These include the OSCORE profile of the ACE framework for authentication and authorization [20] (see Sect. 2.4) as well as the lightweight key establishment protocol Ephemeral Diffie-Hellman Over COSE (EDHOC) [21].

2.4 The ACE framework

In order to access protected resources at a given host device, a requesting device has to explicitly be granted to do so, in accordance with pertaining access control policies.

Since IoT-based network deployments can include several resource-constrained host devices, it is beneficial to entirely offload decision making, authorization-related cryptographic operations, and similar from the host devices to a dedicated management service. This is accomplished by separating authorization to access a resource from the actual resource itself. Additionally, it is convenient to centrally manage the granting to resource access in a network.

The ACE framework for Authentication and Authorization in Constrained Environments [3] is based on the widely deployed OAuth 2.0 [4] authorization framework, and enables its functionalities in the IoT. The ACE framework mainly relies on the following components.

The main functionality and overall approach are inherited from the OAuth 2.0 framework, a standard, widely adopted

solution for authorization and access control. Another component is COSE [7, 8], a compact encoding format for security information based on CBOR [6], which is in turn a binary encoding format designed for small message sizes and code. Furthermore, the lightweight, web-transfer protocol CoAP [5] is used. Lastly, CoAP messages can be secured at the transport layer by using the DTLS protocol suite [17], and/or end-to-end at the application layer by using the OSCORE security protocol [18].

2.4.1 The ACE entities and workflow

The entities involved in a typical interaction as defined by the ACE framework are the following.

The *Client* (C) is a device wanting access to specific resources at a given host, namely the *Resource Server* (RS), with the permission of the corresponding resource owner. The *Authorization Server* (AS) is responsible for authorizing client devices to access resources at an RS according to policies provided by the resource owner, and for providing them with evidence of such authorization in the form of an Access Token. The AS is a trusted third party, practically infeasible to compromise. By its nature, the AS is invested with a large amount of trust, since it manages keying material and generates Access Tokens used to establish secure communication associations between Clients and Resource Servers.

An Access Token is used by C to access protected resources on the RS. Typically, an Access Token is represented as a CBOR Web Token (CWT) [22, 23] efficiently encoded in CBOR [6], or alternatively as a JSON Web Token (JWT) [24, 25] encoded in JavaScript Object Notation (JSON) [26].

Details on how ACE should be implemented for different scenarios can be found in related application and security profiles (see Sect. 2.4.2). In particular, the ACE framework delegates to the profiles the description of how to use the main specification with concrete transport and communication security protocols between the involved entities.

The following describes a typical interaction in the ACE framework between the involved entities C, AS, and RS. In particular, as also shown in Fig. 3, the following steps occur during a full ACE transaction. Note that C, RS, and AS can act as CoAP client or CoAP server, when sending a CoAP request or response, respectively.

Step 1—C sends a request for an Access Token to the AS, targeting the */token* endpoint. When doing so, C specifies:

- The target RS as "audience".
- The requested "scope", i.e., the resources it wishes to access at the RS and through which operations.
- When expected by the used ACE profile and its selected mode, its own public key.

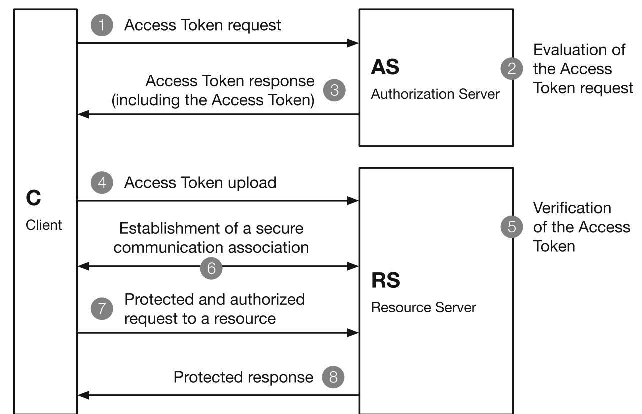

Fig. 3 ACE framework workflow

Step 2—The AS evaluates the request from C against access control policies for the RS, as pre-established by the resource owner. In case of success, the AS produces an Access Token as evidence of the granted authorization. Depending on the used ACE profile and its mode, the AS also generates a symmetric key K, intended to be shared between C and RS. Then, the AS includes, among other elements, the following information in the Access Token.

- The "audience" and the "scope" granted to C.
- Optionally, an indication of the used ACE profile.
- When expected by the used ACE profile and its selected mode, the symmetric key K or the public key of C.

Step 3—The AS provides C with the following.

- The "scope" actually granted to C and specified in the Access Token, if it was possible to only partially satisfy the request.
- Either the newly generated symmetric key K or the public key of the RS, depending on the used ACE profile and its selected mode.
- Optionally, an indication of the profile of ACE to use.
- The Access Token, as either encrypted and authenticated, or instead signed. If encrypted, the Access Token is protected with keying material shared only between the AS and the RS. In either case, the Access Token is opaque to C, which does not understand its content and structure.

Step 4—In case of positive response from the AS, then C *uploads* the Access Token to the RS. This typically happens by sending a request to the */authz-info* endpoint at the RS.

Step 5—The RS verifies that the Access Token is intact and actually originated by the AS, by possibly decrypting it. Then, the RS verifies that the Access Token is still valid (e.g., it is not expired), and that its content is consistent with

the RS' resources and scopes. If so, the RS stores the Access Token.

Step 6—Depending on the used profile of ACE and its selected mode, C and RS perform possible additional exchanges and operations, in order to establish a *secure communication association*, e.g., based on the DTLS or OSCORE security protocol. To this end, they rely on the keying material facilitated by the AS during the previous steps, i.e., each other's public keys or the symmetric key K. Also depending on the used profile of ACE, parts of this step might be combined with the uploading of the Access Token at step 4.

Step 7—C sends a request to RS, in order to perform an operation at one of the resources hosted at RS, consistently with the "scope" granted by the AS at step 3 above. The request is protected using the established secure communication association.

Step 8—The RS checks the request against the Access Token stored for C, and verifies that the requested access and specific operation are in fact consistent with the "scope" in the stored Access Token. In case of positive outcome, the RS processes the request from C and possibly replies with a response. The response is protected using the established secure communication association.

As an optional feature, the AS can provide an additional service to the RS called "introspection". That is, upon receiving an Access Token—or later on while storing it—the RS can send a request to the */introspect* endpoint of the AS, specifying the whole Access Token or a reference to it. The AS can then return fresh information on the current status and validity of the Access Token, which the RS considers to determine whether to accept or preserve the Access Token, or not.

2.4.2 ACE security profiles

Among other things, an ACE profile specifies the following.

- The communication and security protocol for interactions between the involved entities, as providing encryption, integrity protection, replay protection, and binding between requests and responses.
- The method used by C and RS to mutually authenticate.
- The (secure) methods for C to upload an Access Token at the RS.
- The specific key types used (e.g., symmetric/asymmetric), and the protocol for the RS to assert that C possesses such keys (proof-of-possession).

While it is not devoted to any particular profile of ACE, the work presented in this paper specifically relies on the OSCORE profile of ACE defined in [20]. In particular, the OSCORE profile describes how C and the RS can engage in the ACE workflow and establish an OSCORE Security Context for securely communicating with one another using the OSCORE security protocol (see Sect. 2.3).

Upon receiving the Access Token request from C, the AS generates an OSCORE Security Context Object. This includes information and parameters for C and the RS to establish an OSCORE Security Context, such as and especially an OSCORE Master Secret. The AS includes the OSCORE Security Context Object into the Access Token to be released. After that, the AS provides C with both the Access Token and the OSCORE Security Context Object. For the sake of proof-of-possession, C has to prove to the RS to possess the OSCORE Master Secret specified in the Access Token.

Upon uploading the Access Token to the RS, both C and the RS exchange a pair of nonces as well as the respective OSCORE identifiers they intend to use. Then, C and the RS use such values together with the OSCORE Security Context Object received from the AS, to derive a complete, fresh OSCORE Security Context. After that, C sends a first secure request to the RS, protected with the new OSCORE Security Context. Proof-of-possession is achieved when the RS receives such first request and verifies it as cryptographically correct.

3 Revoked token notification mechanism

Access Tokens issued by the AS eventually expire, and the AS can give an explicit indication of expiration time within an Access Token itself. When an Access Token expires, both C and the RS owning that Access Token should discard it. In particular, C would have to get a new Access Token from the AS and upload it to the RS, before continuing accessing resources hosted at that RS.

On top of that, there are additional circumstances when an Access Token may be revoked before its expiration time comes. Practical effects are the same ones mentioned above for the case of expiration, and they apply to both C and the RS, hereafter referred to as *registered devices*, due to their registration at the AS.

Examples of situations resulting in revoking an Access Token before its expiration time include:

- a registered device has been decommissioned;
- a registered device has been compromised, or it is suspected of being compromised;
- there has been a change in the ACE profile for a registered device;
- there has been a change in access policies for a registered device;
- there has been a change in the outcome of policy evaluation for a registered device (e.g., if policy assessment

depends on dynamic conditions in the execution environment, the user context, or the resource utilization).

With particular reference to the ACE framework (see Sect. 2.4), an RS would be able to learn about revoked Access Tokens that it owns, by checking at the AS through the introspection mechanism (see Sect. 2.4.1), in case the AS provides such an optional service. On the other hand, C has no means to learn whether any of the Access Tokens it owns has been revoked.

More generally, it is not possible for the AS to take the initiative and notify registered devices about pertaining Access Tokens that have been revoked, but are not expired yet. Specifically, an Access Token pertains to a Client if the AS has issued the Access Token and provided it to that Client. Also, an Access Token pertains to a Resource Server if the AS has issued the Access Token to be consumed by that Resource Server.

The novel approach specified in the standard proposal [9] and implemented in the work presented in this paper aims to fill this gap. That is, it specifies a method for registered devices to access and observe a *Token Revocation List (TRL) resource* [9] on the AS, in order to get an updated list of revoked, but yet not expired, pertaining Access Tokens. The main benefits of this method are that it complements the introspection mechanism, and it does not require any additional resources or endpoints to be created on the registered devices.

In particular, registered devices can rely on resource observation [10] for CoAP [5]. That is, the AS would automatically send a notification to an observer registered device, when the status of the TRL resource changes. Specifically, this happens when an Access Token pertaining to that device gets revoked, or a revoked Access Token previously included in the list eventually expires.

The TRL resource at the AS does not contain the full representation of the Access Tokens that are revoked but are not expired yet. Instead, the TRL includes ad-hoc identifiers of Access Tokens, namely *token hashes*. These are computed as cryptographic hashes of the Access Tokens as per [27], and make it possible to correctly handle different types of Access Tokens conveyed over different transports. When an Access Token is revoked, its token hash is added to the TRL resource. When, later on, that Access Token expires, its token hash is removed from the TRL resource.

As mentioned above, a registered device can at any time send a request to the TRL resource at the AS, or, in addition to that, specifically observe the TRL resource in order to receive notification responses in case of changes in the resource representation. In either case, the registered device is not going to receive the full content of the TRL, but rather only a pertaining subset of it, which includes only the token hashes of the Access Tokens pertaining to that registered device, as extracted from the whole current representation of the TRL resource.

More specifically, a registered device can access the TRL resource at the AS in two different modes, which result in different responses from the AS.

Full query mode—The AS returns the token hashes of the revoked Access Tokens currently in the TRL and pertaining to the registered device that has sent the request.

Diff query mode—The AS returns a set of *diff entries*. Each entry is related to one of the N most recent updates in the portion of the TRL pertaining to the registered device that has sent the request, where N is specified as a query parameter of the request. In particular, the entry associated with one of such updates contains a list of token hashes, such that: (i) the corresponding revoked Access Tokens pertain to the requester; and (ii) they were added to or removed from the TRL at that update. This mode of operation can further rely on its “cursor” extension, in order to allow a registered device to retrieve a set of diff entries not only as limited to the most recent TRL updates, but rather starting from an arbitrary TRL update taken as resumption point.

4 Usage control in ACE

The AS in the ACE framework implements a decision engine to determine whether a Client can be granted access rights to protected resources at some Resource Server. Additionally, given the dynamic conditions in the execution environment, in the user context, and in the resource utilization, as well as for the other reasons discussed in Sect. 3, the AS should be able to revoke previously granted access to resources by invalidating Access Tokens. The ACE framework is not devoted to any specific decision engine used by the AS.

In this section, we present how we have integrated the UCON framework (see Sect. 2.1) as the decision engine used by the AS in the ACE framework. The adoption of the UCON framework significantly improves the security of the IoT deployments with respect to traditional access control frameworks for several reasons.

First of all, using the UCON framework allows for automatically, effectively, and timely detecting when an Access Token must be revoked. Specifically, the UCON framework makes it possible to easily express conditions taking into account dynamic factors that affect the issuing and revocation of Access Tokens through the U-XACML language and its time-related tags. The latter makes it possible to easily specify which conditions in the policy have to remain valid in order to not revoke granted access rights before the originally intended expiration time.

Secondly, for the evaluation and the enforcement of U-XACML policies, the UCON framework relies on an

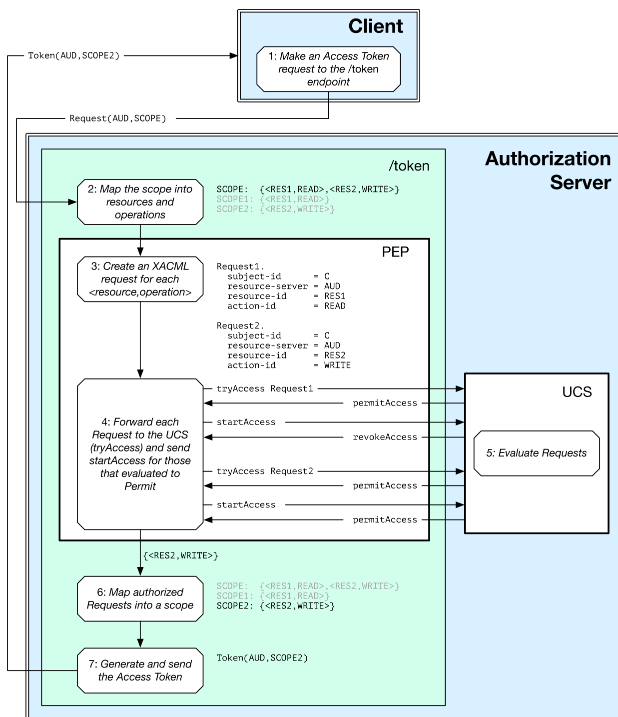

Fig. 4 Access Token request and response, showing the integration of the UCON framework in the AS

architecture that can be easily integrated into the ACE framework to enforce the automatic revocation of Access Tokens.

The following describes how the PEP and UCS components presented in Sect. 2.1 have been integrated in the AS.

4.1 Integration of the UCON framework in the ACE authorization server

According to the ACE workflow, a Client C sends an Access Token request to the */token* endpoint at the AS, specifying an audience AUD and a scope SCOPE (step 1 of Fig. 3). Through this request, the Client is asking for an Access Token that enables it to perform an operation OP (or more than one) on a specific resource RES (or more than one, as inferred from the scope) at the target RS AUD. Upon receiving the Client's request, the AS evaluates whether the Client C can be granted access to the resource RES and operation OP at the target RS AUD (step 2 of Fig. 3).

In the proposed integration, the AS delegates this decision to the UCS. To this aim, the Client's request is translated into one or more UCON requests. This task is entrusted to the PEP component, which is embedded in the */token* endpoint at the AS, and which communicates with the UCS, as shown in Fig. 4. In the figure and in the proposed implementation, the UCS resides inside the AS, but it could also be external and be accessed remotely by the AS, e.g., through REST calls.

The key to integrate UCON in the AS is therefore the translation of the Client's request into UCON requests that the UCS is able to interpret and evaluate. From a UCON perspective, the previous Access Token request translates to: the *subject* with "subject-id"[6] C wants to access the *resource* with "resource-id" RES at the target RS AUD and perform the *action* with "action-id" OP on it. Figure 5 shows the UCON request in XACML format, as translated by the PEP. The target RS is specified as an attribute of category *resource* and with identifier "resource-server".

The PEP sends the UCON request in a tryAccess message to the UCS, which evaluates it, produces a pre-decision, and replies with either a permitAccess or a denyAccess message as per steps 1 and 2 of Fig. 2. If a permitAccess message is received, the PEP sends a startAccess message to the UCS. Then, if the UCS replies again with a permitAccess message, the PEP tells the */token* endpoint the set of allowed resources and operation, i.e., {<RES,OP>}, and the */token* endpoint proceeds with the Access Token generation including the scope SCOPE. Finally, the Access Token is sent to the Client.

Note that this flow slightly differs from the one of the UCON framework (described in Sect. 2.1). That is, since the PEP does not have direct control over the resource (which resides on the Resource Server), it cannot enforce the access decision after the reception of a permitAccess message following a tryAccess message, as per step 3 of Fig. 2. Instead, after receiving a permitAccess message following a startAccess message, it informs the AS about whether the resource and the operation can be granted to the Client. This results in the AS creating an Access Token that includes such information specified as a scope.

In general, the Client can specify a scope that is mapped to more than one resource and operation. However, the AS may grant the Client only a subset of resources and operations, depending on the Client's access privileges. These access privileges are compiled within usage control policies and stored at the PAP. Figure 4 shows an advanced example in which the specified scope is mapped to two resources, and the right to access only one resource is eventually granted to the Client.

In this example, the scope SCOPE refers to the set of resources and operations {<RES1,READ>,<RES2,WRITE>}, the scope SCOPE1 to {<RES1,READ>}, and the scope SCOPE2 to {<RES2,WRITE>}. In step 1, the Client C specifies SCOPE as scope and AUD as audience in its Access Token request. The AS maps the scope SCOPE

[6] For the sake of readability, in the text we use only the last part of the actual XACML attributes' names defined within the standard. In this paper, this does not create ambiguity. The full name of the attributes is reported in Fig. 5 as the value of AttributeId within the <Attribute> element.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Request ReturnPolicyIdList="false" CombinedDecision="false"
  xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17">
  <Attributes Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id" IncludeInResult="false">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">C</AttributeValue>
    </Attribute>
  </Attributes>
  <Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" IncludeInResult="false">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">RES</AttributeValue>
    </Attribute>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-server" IncludeInResult="false">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">AUD</AttributeValue>
    </Attribute>
  </Attributes>
  <Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" IncludeInResult="false">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">OP</AttributeValue>
    </Attribute>
  </Attributes>
</Request>
```

Fig. 5 Example of UCON request created by the PEP and submitted to the UCS. The Client C (attribute “subject-id”) requests access privileges to perform the operation OP (attribute “action-id”) on the resource RES (attribute “resource-id”) at the audience AUD (attribute “resource-server”) associated with the target RS

to {<RES1,READ>,<RES2, WRITE>} and passes this set to the PEP (step 2). Then, the PEP creates two UCON requests (step 3) containing C as “subject-id” and AUD as “resource-server”. Additionally, one UCON request contains RES1 as “resource-id” and READ as “action-id”, while the other contains RES2 as “resource-id” and WRITE as “action-id”.

In step 4, the PEP sends the first UCON request to the UCS through a tryAccess message. The UCS replies with a permitAccess message (thus indicating that the result of the pre-decision was Permit), and then the PEP sends a startAccess message. The result of the ongoing-decision is Deny, hence the UCS replies with a revokeAccess message. The PEP repeats the same message exchange for the second UCON request, but this time the result of the ongoing-decision is Permit, hence the UCS replies with a permitAccess message.

The PEP creates a set of resources and related operations that can be granted to the Client, i.e., {<RES2,WRITE>}, according to the UCS decision, and propagates it to the */token* endpoint. Finally, the */token* endpoint expresses {<RES2, WRITE>} through the scope SCOPE2 (step 6), generates the Access Token (step 7), and includes such a scope together with the Access Token in the Access Token response to the Client.

Note that, when an Access Token grants access to n resources, then n sessions with status START_ACCESS are present at the SM of the UCS.

4.2 Access token revocation and notification

The UCON framework provides a continuous evaluation of the usage control policies paired with the ongoing usage sessions, and it can revoke accesses to resources according to the mechanisms described in Sect. 2.1. Its revocation mechanism is fine grained and capable of revoking access to a specific resource for performing a specific operation on that resource. On the other hand, the ACE framework considers the revocation of an Access Token as a whole. Since a single Access Token can grant access to more than one resource for one or more operations on the target resource(s), this may result in the creation of more than one session per Access Token at the UCS. That is, in ACE, the revocation of a Client’s access rights on a resource at an RS implies the revocation of the whole Access Token issued to that Client and to be consumed by that RS. This means that, for example, if an Access Token grants the Client access to RES1 and RES2 at a Resource Server, and only RES1 gets compromised, the AS must revoke the whole Access Token, thus preventing the Client from accessing RES2 too. There is no means for the AS to revoke the access grant to RES1 only. On the contrary, the UCON framework allows to invalidate accesses to single resources, as it associates one resource with one session. In order to correctly integrate the UCON framework into the AS, it has to be ensured that, when an Access Token is revoked, all the associated ongoing usage sessions are terminated.

In the proposed integration, the PEP embedded in the AS implements the logic to group together the sessions related to the same Access Token, as all pertaining to the scope of that Access Token. By doing so, when the PEP receives a revokeAccess message following a policy re-evaluation

for a specific session identifier, it first finds the Access Token associated with that session, and it communicates to the AS that such an Access Token must be revoked. The AS expunges the Access Token and stores its token hash in the TRL resource. Once the revocation is complete, the PEP looks for other session identifiers associated with the same Access Token, and, finally, it sends an `endAccess` message to the UCS for each session associated with the Access Token.

Note that this flow differs from the one of the UCON framework described in Sect. 2.1. That is, since the PEP does not have direct control over the resource (which resides on the Resource Server), it cannot directly terminate the access after the reception of a `revokeAccess` message, as per step 6(ii) of Fig. 2. Instead, after receiving a `revokeAccess` message, it informs the AS about which Access Token has to be revoked. This results in the AS updating the TRL resource, notifying the registered devices, and deleting the Access Token.

4.3 Reference examples

In order to motivate the automatic and policy-driven revocation mechanisms described in this work, the following provides a practical example concerning a smart home environment. In particular, we describe the workflow from the Access Token request to the Access Token revocation and related notification. Next, additional use cases regarding other IoT scenarios are also discussed.

Tenants of a smart home can program the timing and settings of the washing cycle of a washing machine (which acts as an ACE Resource Server), by means of an application installed on their smartphone (acting as an ACE Client). The washing machine provides a number of resources, such as `spin cycle` and `high-temp cycle`, and the actions defined on such resources might be `start` and `stop`. Tenants could be allowed by the administrator to trigger the execution of high-temperature wash cycles, as long as the threshold of daily energy consumption of the overall household is not passed.

Consistently, the administrator defines a policy for the tenants and the resource `high-temp cycle`, and it adds a condition concerning the attribute `daily energy consumption` in the ongoing-section of the UCP, specifying that its value must be lower than a certain threshold in order to let tenants start high-temperature wash cycles. The value of the overall daily energy consumption in the household is managed by an AM, which is the smart meter deployed in the smart home.

Within the application, the tenant selects the high-temperature wash cycle and then clicks the start button to begin the washing process. If an Access Token has not already been issued to the tenant and uploaded to the washing machine, or if such an Access Token has ceased to be valid due to expiration or revocation, an Access Token request for the resource `high-temp cycle` is sent to the AS. The PEP creates a UCON request and sends it within a `tryAccess` message to the UCS, which replies with a `permitAccess` message and creates a session with session identifier $SID^{\star}$ and status `TRY_ACCESS`. Then, the PEP sends a `startAccess` message to the UCS, which replies with a `permitAccess` message—if the daily energy consumption is currently lower than the threshold specified in the ongoing-section of the UCP—and updates the session with the status `START_ACCESS`. The access is therefore granted, and an Access Token is issued to the Client. Then, the Client uploads the Access Token to the washing machine and establishes a secure communication association with it. After that, as per the granted access, the tenant is able to trigger the execution of the high-temperature wash cycle by sending a protected request, e.g., a POST request, to the corresponding resource `high-temp cycle` at the washing machine.

From the moment when the UCS updates the status of the session to `START_ACCESS`, the UCS has been continuously monitoring the mutable attribute `daily energy consumption`, since it is in the ongoing-section of an active session. In case the daily consumption threshold is passed, the Access Token is revoked. That is, the policy re-evaluation at the UCS returns a Deny authorization decision because the current daily energy consumption value (retrieved from the smart meter) is higher than the threshold value set within the ongoing-section of the UCP. Then, the UCS updates the session with the status `REVOKE_ACCESS` and sends a `revokeAccess` message containing the session identifier $SID^{\star}$ to the PEP. The PEP first finds the Access Token associated with that session and tells the AS that such an Access Token must be revoked. Hence, the AS expunges the Access Token, and the token hash of the revoked Access Token is stored in the TRL of the AS. Then, the AS can send notifications to the registered devices observing the TRL resources and to which the Access Token pertains (see Sect. 3). Finally, the PEP selects all the session identifiers associated with the same Access Token—only $SID^{\star}$ in this example—and sends one `endAccess` message, specifying $SID^{\star}$, to the UCS, which deletes the related session.

If the tenant attempts to obtain another Access Token within the same day, the AS does not grant a scope allowing access to the resource `high-temp cycle` for performing the previous operation at the washing machine, because the ongoing-section of the UCP is not satisfied at the time of Access Token request.

Note that, in this example, the occurred revocation does not terminate any ongoing high-temperature wash cycle that is already ongoing as previously started based on the old Access Token. It only prevents from starting a new one,

since this would be upon attempting to access the resource `high-temp cycle` without being allowed to. However, if stopping an ongoing wash cycle—after the daily threshold has been reached—is critical, other mechanisms should be implemented, such as having the logic on the washing machine to terminate any ongoing operation started by the tenant, as consequence of having expunged the Access Token.

While keeping the focus on the smart home scenario, another particularly apt example involves a security camera, which acts as the Resource Server. The administrator defines a policy that grants access to the live footage captured by such a camera to a guardian appointed by a surveillance agency (which acts as the Client) only when the smart home tenant is not on the premises. The tenant's location can be determined, e.g., by means of the GPS module of their smartphone, acting as an AM, and is used within the ongoing-section of a UCP. Such a UCP is intended to protect the privacy of the smart home tenant, by preventing the guardian from accessing the footage while the tenant is at home.

In this example, an Access Token is issued to the Client only if, at the time of the request, the tenant is off the premises. If so, the Client can upload the Access Token to the security camera, establish a secure communication association with it, and access the video footage. Moreover, the Access Token will be revoked when the tenant returns to the premises. Following such a revocation, if the security camera has been observing the TRL resource at the AS, the camera is promptly notified by the AS. Consequently, the camera deletes the Access Token and terminates the secure association with the Client, thus preventing the guardian from accessing the captured footage from then on. A new Access Token will not be issued as long as the tenant remains on their premises.

The examples provided above explore scenarios involving the revocation of Access Tokens to safeguard the Resource Server from potential misconduct by the Client. In the following, we illustrate a reverse scenario where Access Tokens are revoked to protect the Client from the result of actions performed on a compromised or malfunctioning Resource Server. This situation arises when there is a need to prevent the Client from performing actions on the Resource Server that could potentially harm the Client's interests or security.

In office environments or shopping centers, individuals—be they staff members or customers—assume the role of Clients. These Clients obtain an Access Token to facilitate interactions via their smartphones with a vending machine, i.e., the Resource Server, dispensing food. The administrator defines a policy that grants usage of the vending machine only if its internal temperature remains below a specific threshold over a time window, ensuring freshness and integrity of the food, and preventing any potential spoilage. This information can be determined, e.g., by means of a smart device embedded within the vending machine, acting as an AM, and is used within the ongoing-section of a UCP. Such a UCP is intended to protect the physical safety of the Clients, by preventing them from consuming spoiled food.

At a certain point, the monitoring system detects malfunctioning of the vending machine, when the reading from the smart device is above the threshold. Upon reaching this conclusion facilitated by the UCS, the AS promptly revokes the Access Tokens issued to all Clients for that Resource Server. Moreover, until the reading is not back below the threshold and the potentially dangerous food in the vending machine has not been replaced, the AS does not issue new Access Tokens that allow purchasing food from the vending machine. This prevents the Clients from accessing what is deemed to be a dangerous service.

5 Experimental evaluation

In this section, we describe the execution workflow considered during our experiments, the measured time intervals of interest, and the specific setup used for the experiments. In Sect. 6, we report and discuss the obtained results.

The results from our experiments are not compared with those from other studies. Indeed, to the best of our knowledge, there are not other alternative, akin approaches to consider for comparison.

The aim of our experiments is to measure various time performance metrics, among which the time required by a Client for obtaining an Access Token and accessing a protected resource at a Resource Server, as well as the time required by the AS for revoking an Access Token. Moreover, we investigate and compare the time performance displayed by the different mechanisms used by Client and Resource Server for gaining knowledge of Access Token revocations, i.e., introspection (see Sect. 2.4.1), polling of the TRL resource at the */trl* endpoint, and observation of the TRL resource at the */trl* endpoint (see Sect. 3).

In Sect. 5.1, we introduce the considered workflow, describing the ACE and UCON configurations (e.g., the used ACE profile and UCP), and present the sequence of events occurring during the experiments.

Throughout the execution of the workflow described in Sect. 5.1, we identified four time intervals of interest to be measured, as detailed in Sect. 5.2. In Sect. 6.1, we assess and discuss how these time intervals vary with the mechanism used by Client and Resource Server for gaining knowledge of Access Token revocations. Moreover, since the scope requested by the Client and the complexity of the policies evaluated by the UCS affect the measured time intervals, in Sect. 6.2 we present further results, obtained from a second set of experiments where we assess the performance of the UCS playing the role of decision maker in the ACE framework.

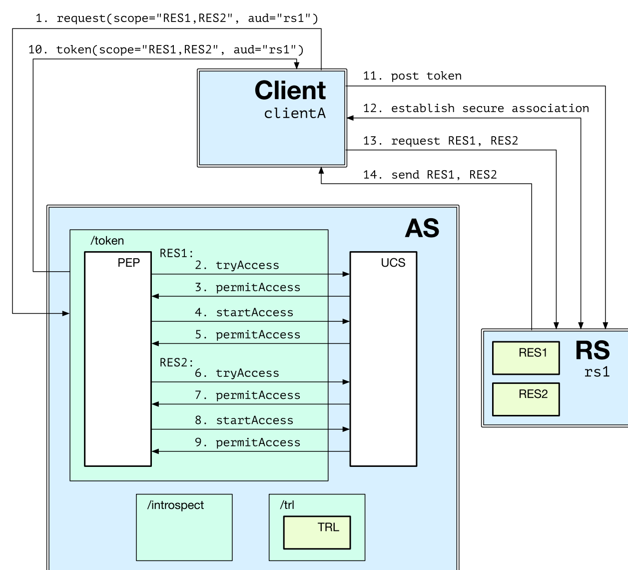

Fig. 6 Phase 1 of the workflow with detailed UCS steps. Access to both the resources `RES1` and `RES2` is granted to the Client

All the experiments were performed on a real testbed, consisting of a full-fledged workstation acting as the AS, and two IoT devices acting as Client and Resource Server. The testbed is described in detail in Sect. 5.3.

5.1 Workflow

The workflow considered in our experiments consists of three phases. Phase 1 is shown in Fig. 6, while Phase 2 and Phase 3 are shown in both Figs. 7 and 8.

In Phase 1, the Client `clientA` makes an Access Token request to the AS. It asks for the scope `"RES1 RES2"`, indicating the RS `rs1` as audience. The AS translates `clientA`'s request and generates the related UCON requests, as explained in Sect. 4.1, where the value of the attribute "subject-id" is `clientA`, the value of "resource-server" is `rs1`, and the value of "action-id" is `read` for both the UCON requests, while the value of "resource-id" is `RES1` for the first UCON request and `RES2` for the second one.

The PAP contains two UCPs: one UCP (`policy-1`) is applicable to the first UCON request, and its ongoing-section contains a condition concerning a mutable attribute `attr1`. The other UCP (`policy-2`) is applicable to the second UCON request, and its ongoing-section contains a condition concerning a mutable attribute `attr2`. The value of each mutable attribute is retrieved every 10 ms from a local AM (which is implemented as a file stored on the AS) by a dedicated PIP. If the retrieved value matches the attribute value defined in the UCP, the condition and, consequently, the ongoing-section of the policy are satisfied. For what concerns Phase 1, in the tests we conducted we set the initial values of the attributes in such a way that the result of the pre-decision and of the ongoing-decision is Permit for both the UCON requests, so the requested scope for both resources `RES1` and `RES2` is initially granted to `clientA`. Hence, the AS generates an Access Token `token-1` with scope `"RES1 RES2"` and sends it to `clientA`. In turn, `clientA` uploads the Access Token to `rs1` and establishes a secure communication association with it (in our case, an OSCORE Security Context [18] as per the OSCORE profile of ACE [20]). After that, according to a *request interval* of 1 s, `clientA` periodically and alternatively sends GET requests to the protected resources `RES1` and `RES2`, in order to retrieve their current representation.

Phases 2 and 3 are detailed as sequences of events in both Figs. 7 and 8, which also indicate the time intervals of interest discussed in the next section.

In Phase 2, the value of the mutable attribute `attr1` isintentionally changed at a point in time randomly selected between 30 and 60 seconds from the moment the AS was first initialized (step 1 of Figs. 7 and 8). Once the UCS notices that (step 2), it performs re-evaluation of `policy-1` since it contains `attr1` (step 3). In the tests we conducted, we have chosen the new value of `attr1` in such a way that the policy re-evaluation results in a Deny decision (step 4). Hence, the PEP receives the `revokeAccess` message from the UCS (step 5) and, consequently, the AS revokes `token-1` and adds the token hash to the TRL (step 6). As described in Sect. 4.1, after the token revocation is complete, the PEP sends to the UCS an `endAccess` message for each session that was associated with `token-1`, two in our case (step 7). However, at this stage, the Client and RS are still not aware that the revocation has taken place, hence they retain and keep relying on `token-1` until they learn about its revocation.

Phase 3 starts after Phase 2, and in the first step either the RS or the Client detects that `token-1` has been revoked. The order of these two events is nondeterministic. Figure 7 shows the scenario where the RS detects the token revocation before the Client (scenario *rsFirst*), while Fig. 8 shows the scenario where the Client detects the token revocation earlier than the RS (scenario *cFirst*).

The RS can learn of the token revocation either from the */trl* endpoint (through polling or Observe) or from the */introspect* endpoint at the AS. As soon as the RS detects that the Access Token was revoked, it deletes `token-1`, as well as the secure communication association with the Client. On the other hand, the Client can learn that the Access Token was revoked either from the */trl* endpoint at the AS (through polling or using Observe), or from a "4.01 Unauthorized" response received from the RS when attempting to access a protected resource. In the latter case, the Client *assumes* that `token-1` has been revoked, since it has no means to unambiguously learn whether that is actually the case.

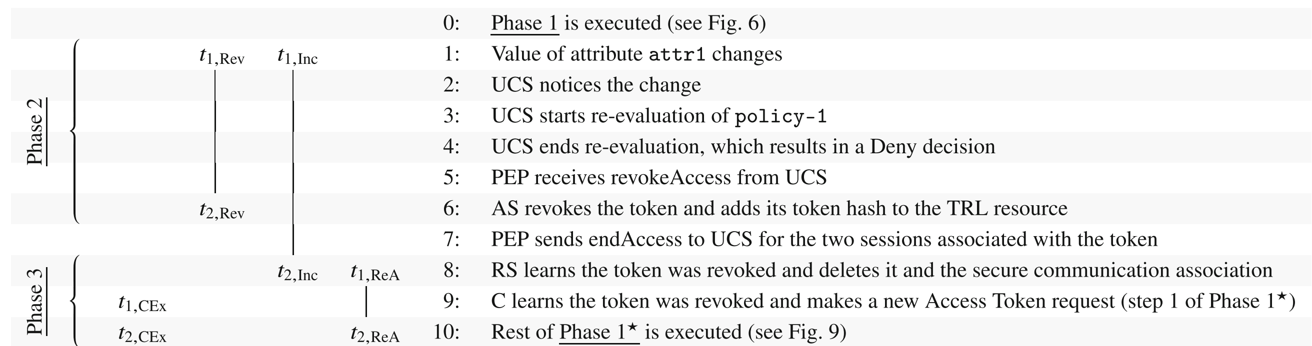

Fig. 7 Phase 2 and 3 of the workflow, where the RS learns that the token was revoked before the Client does (scenario rsFirst). In this scenario, $t_{1,\mathrm{ReA}}$ starts with $t_{2,\mathrm{Inc}}$. Also, time intervals of interest are reported

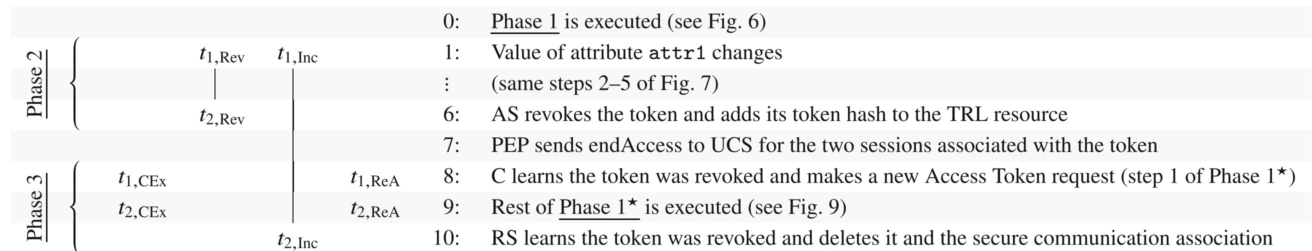

Fig. 8 Phase 2 and 3 of the workflow, where the Client learns that the token was revoked before the RS does (scenario cFirst). In this scenario, $t_{1,\mathrm{ReA}}$ starts with $t_{1,\mathrm{CEx}}$. Also, time intervals of interest are reported

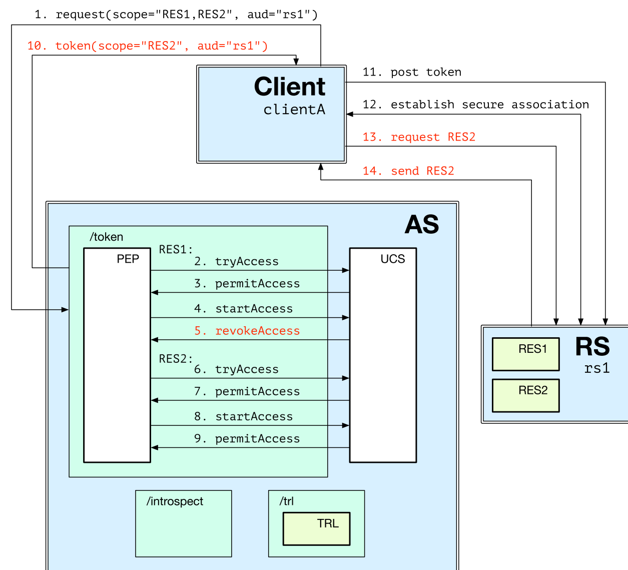

Fig. 9 Phase 1⋆ of the workflow with detailed UCS steps. Only the resource RES1 is granted to the Client

In either scenario, the workflow continues with Phase 1⋆ (Fig. 9), which is similar to Phase 1 (Fig. 6) but diverges starting from step 5.

This time, the evaluation of the ongoing-section of `policy-1` produces a Deny authorization decision, while the evaluation of `policy-2` still produces a Permit authorization decision (the value of `attr2` remains unchanged throughout the experiment). Hence, the AS generates a new Access Token, called `token-2`, with scope `"RES2"`, thus granting access only to the resource `RES2`, and sends it to the Client. The Client uploads this Access Token to `rs1` and establishes a new secure communication association with it. Finally, the Client makes a GET request to the protected resource `RES2` to access it.

All the experiments that we performed follow the workflow just described and are detailed in Sect. 6. In particular, they consider different ways according to which the Client and the RS learn about the revocation of `token-1`.

5.2 Time intervals of interest

We identified four time intervals of interest that we measured through several test campaigns. For each of them, Figs. 7 and 8 show the start time and the end time, along with a textual description of the corresponding event.

Client Experience Time, t_{CEx}—It ranges from $t_{1,\mathrm{CEx}}$ to $t_{2,\mathrm{CEx}}$, where:

$t_{1,\mathrm{CEx}}$ = time when the Client sends an Access Token request to the AS;

$t_{2,\mathrm{CEx}}$ = time when the Client obtains a successful protected response from the RS, following a resource access.

This time interval measures the time required for a complete execution of the ACE workflow, which consists of

the Access Token request and response, the Access Token upload to the RS, the establishment of a secure communication association with the RS, and the request and response exchanged between Client and RS when accessing the protected resource. From the Client's perspective, this time interval measures the time required to retrieve the representation of a protected resource, i.e., the time to complete the Client's "experience".

Revocation Time, t_{Rev}—It ranges from $t_{1,\text{Rev}}$ to $t_{2,\text{Rev}}$, where:

$t_{1,\text{Rev}}$ = time when the value of an attribute changes, in such a way to trigger the revocation of an Access Token. Note that the time when the UCS gains knowledge of this change may come later (i.e., the detection of the attribute value change could occur with a delay with respect to the time when the attribute value actually changed);

$t_{2,\text{Rev}}$ = time when the AS completes the revocation of the Access Token.

This time interval represents the total time required by the AS for performing the token revocation. It is intended to capture the time spent by the AS to detect that an attribute's value has changed (`attr1` in our workflow) plus the local processing at the AS for completing the token revocation.

Inconsistency Time, t_{Inc}—It ranges from $t_{1,\text{Inc}}$ to $t_{2,\text{Inc}}$, where:

$t_{1,\text{Inc}}$ = the same as $t_{1,\text{Rev}}$;

$t_{2,\text{Inc}}$ = time when the RS deletes its stored Access Token and the related secure communication association with the Client as a consequence of having gained knowledge of the Access Token revocation.

This time interval can be split into two parts. The first part is the Revocation Time (t_{Rev}) defined above. The second part is a sort of *information propagation time*. We expect that this time interval heavily depends on the mechanism that the RS uses to check the token validity, i.e., introspection, polling the */trl* endpoint at the AS, or observing the */trl* endpoint at the AS. This time interval measures an inconsistency, i.e., the time that the Client is still granted the access to a protected resource when it should not be, because the factors (i.e., the attribute) granting such an access have changed, thus causing the token revocation.

Re-Admission Time, t_{ReA}—It ranges from $t_{1,\text{ReA}}$ to $t_{2,\text{ReA}}$, where:

$t_{1,\text{ReA}}$ = the earliest among the time when the RS deletes its stored Access Token and the related secure communication association ($t_{2,\text{Inc}}$), and the time when the Client learns that the Access Token was revoked ($t_{1,\text{CEx}}$). That is, $t_{1,\text{ReA}} = \min(t_{2,\text{Inc}}, t_{1,\text{CEx}})$.

$t_{2,\text{ReA}}$ = time when the Client receives a successful, protected response from the RS, following a resource access based on the second Access Token, obtained after the revocation of the first one.

This time interval captures the time required by a Client to re-access (a subset of) protected resources after either: (i) it has learned that an Access Token has been revoked; or (ii) the RS has deleted its stored Access Token and the secure communication association with the Client. Consistently with the specific workflow of our experiments, this time interval measures the amount of time that the Client is denied access to `RES2`, i.e., a protected resource for which it should still have access rights.

Note that we assume the Client to be honest, i.e., it does not use an Access Token which it knows to be revoked. If the Client learns before the RS that the Access Token has been revoked (Fig. 8), this time interval coincides with the Client Experience Time (t_{CEx}).

5.3 Experimental setup

In our experiments, the AS process runs on a laptop computer (Dell Alienware m15 R7) equipped with 32 GB of RAM, an Intel® Core™ i7-12700 H CPU, and running the Ubuntu 22.04 LTS 64-bit operating system. The Client and the RS processes run on two Raspberry Pi 4 Mod. B Rev 1.4 equipped with 8 GB of RAM. The ACE entities are connected to a Linksys router (model WRT54GL) to form a wired network.

An existing Java implementation of ACE[7] has been extended[8] to integrate the customized UCON-based decision maker, together with the mechanism for the notification of revoked Access Tokens described in Sect. 3. All the processes run the extended implementation.

The AS device is configured to act as a Network Time Protocol (NTP) server, and the Client and the RS devices run NTP clients which synchronize with the NTP server. This ensures accurate time synchronization between the entities (from our experiments, an inaccuracy lower than 1 ms), which is essential to estimate the time intervals of our interest with acceptable precision. Indeed, for some time intervals, the start time and the end time are taken on different devices, and poorly synchronized clocks would lead to unreliable measurements.

For each test with a fixed configuration, 100 repetitions are performed, and the timestamps of the start time and end time of all the time intervals of interest are recorded. When performing the data analysis on the results, for each single repetition, the time when the attribute changes value ($t_{1,\text{Rev}}$) is taken as reference, i.e., $t_{1,\text{Rev}}$ is set to 0, and all the other times are computed as offsets from $t_{1,\text{Rev}}$.

The interquartile range (IQR) method is then applied to these results, and the identified outliers are dropped. Finally,

[7] https://bitbucket.org/marco-tiloca-sics/ace-java/src/master/.

[8] https://bitbucket.org/marco-rasori-iit/ace-java/src/ucs/.

Table 1 Tested configurations. For each configuration, the way the Client and the RS gain knowledge of the Access Token revocation is shown

Name	Client	RS
ua-i15	4.01 unauthorized	introspect every 15 s
p15-p15	polling every 15 s	polling every 15 s
p15-o	polling every 15 s	observe
o-p15	observe	polling every 15 s
o-o	observe	observe

the duration of each time interval of interest is computed and averaged, and 95 % confidence intervals are computed.

6 Experimental results

This section presents and discusses the results from our experimental performance evaluation, according to what has been defined in Sect. 5. In Sect. 6.1, we compare the time performance displayed by the different mechanisms that the Client and Resource Server use for gaining knowledge of an Access Token revocation. Then, in Sect. 6.2, we assess the time employed by the UCS to carry out the authorization tasks when varying the complexity and number of the evaluated access policies.

6.1 ACE workflow performance

All the performed tests follow the workflow described in Sect. 5.1, but they differ in the way the Client and RS gain knowledge of the Access Token revocation. Table 1 reports the tested configurations and the way in which the Client and the RS learn about the token revocation.

In the configuration ua-i15, which uses introspection, we basically test the performance of an AS not implementing the notification mechanism for revoked tokens. In particular, the RS performs introspection every 15 seconds (*introspect interval*), querying the AS for checking the validity of each Access Token that the RS stores. In our tests, the RS stores only the Access Token `token-1`, received by the Client during Phase 1. On the other hand, the Client has no means to learn that a token has been revoked, and thus it *assumes* so upon receiving a “4.01 Unauthorized” message from the RS, in response to a GET request to a protected resource, the first time that the Client attempts to access the resource again. Indeed, according to the ACE framework, this response is sent by the RS when no secure communication association exists with the Client. The tests performed using this configuration always relate to the scenario rsFirst, where the RS learns of the token revocation before the Client does. When the Client receives a 4.01 response, the workflow continues with the Client making the second Access Token request to the AS.

In the second configuration (p15-p15), both the Client and the RS poll the TRL resource at the AS. In our tests, the *polling interval* is set to 15 seconds, and we have Client and RS alternatively querying the AS, so that the AS receives a request about every 7.5 seconds. In order to model the worst case, the alternating pattern is intentionally enforced, by means of the configuration parameter values used in the experiments.

In the third configuration (p15-o), the Client polls the TRL resource at the AS every 15 seconds, while the RS observes the TRL resource. Although we expect that most of the times the RS learns of the token revocation before the Client does, we do not let the Client assume that the token was revoked when receiving 4.01 responses, since the Client has now the means to gain such a knowledge with certainty.

The fourth configuration (o-p15) is reversed compared to the previous one (p15-o), with the RS polling the TRL resource at the AS while the Client observes the TRL resource.

Finally, in the last tested configuration (o-o), both the Client and the RS observe the TRL resource at the AS. Since (i) the sending of Observe notifications to notify the registered devices does not have to follow a particular order; and (ii) the registered devices are both connected to a wired network and are one hop from the AS, we expect that the tests performed with this configuration roughly relate about half of the times to the scenario rsFirst, and half of the times to the scenario cFirst.

Figure 10 shows the measured time intervals of interest for each configuration through bar charts. Bars spread across a time scale (y-axis), and each bar represents one of the time intervals of interest characterized by its start time, duration, and end time. The x-axis (at the top of the graphs) reports the “ending actor”, which is the entity that determines the end of the time interval. For example, the Inconsistency Time (t_{Inc}) ends when the RS deletes the Access Token and terminates the secure communication association with the Client, while the Revocation Time (t_{Rev}) ends when the AS completes the revocation of the Access Token (see Sect. 5.2). A dashed magenta horizontal line is taken as reference and indicates the end time of the token revocation ($t_{2,\text{Rev}}$). Its value is on average 55 ms. A dashed vertical brown arrow and a dotted vertical blue arrow indicate the time employed by the RS and the Client, respectively, to learn of the token revocation. Hence, they span from $t_{2,\text{Rev}}$ to $t_{2,\text{Inc}}$ and from $t_{2,\text{Rev}}$ to $t_{1,\text{CEx}}$, respectively. These indications help better understand the insight of the results.

The first thing to notice in Fig. 10 is that the Revocation Time (t_{Rev}, magenta bar) is independent of the specific tested configuration, i.e., it is not influenced by the ways in which Client and RS learn about revoked Access Tokens. Indeed,

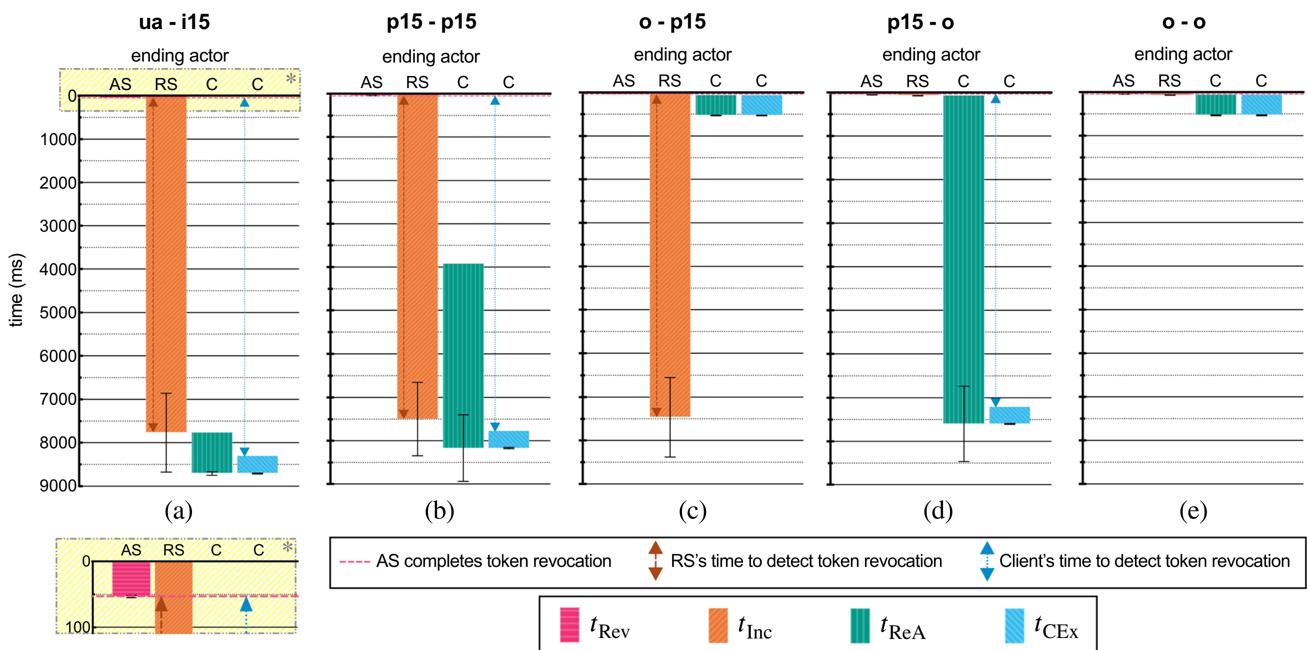

Fig. 10 Average Revocation Time (t_{Rev}), Inconsistency Time (t_{Inc}), Re-Admission Time (t_{ReA}), and Client Experience Time (t_{CEx}) with 95 % confidence intervals for all the configurations tested. The y-axis origin, i.e., $t = 0$ ms, is set to be the time when the attribute value changes, i.e., $t_{1,\text{Rev}}$. The yellow box zooms in on the first 100 ms of Fig. 10a

t_{Rev} mainly depends on the UCS processing time at the AS after an attribute's value changes. In Sect. 6.2, we present and discuss our performance evaluation of such specific process.

From Fig. 10, we observe that in the vanilla configuration ua-i15, as well as in p15-p15and in o-p15, on average, the RS detects the Access Token revocation about 7.5 seconds after the revocation is completed (i.e., half of the introspect/polling interval), and this determines the Inconsistency Time. Although polling and introspection achieve similar results in terms of Inconsistency Time (t_{Inc}, orange bar), they are quite different by their very nature. Indeed, the advantage of polling the TRL resource at the AS instead of introspecting an Access Token is twofold: (i) the RS saves upload bandwidth since polling the TRL resource simply requires a GET request, instead of a POST request that provides the whole Access Token or a reference to it as required by introspection; (ii) with just one polling request, the RS can learn about the revocation of multiple Access Tokens whereas, with one introspection request, the RS learns about the validity of one specific Access Token only. In other words, in order to gather the same information that can be obtained through a polling request to the TRL resource, the RS should make an introspect request for each Access Token that it stores, thus consuming additional time and bandwidth. Note that, in our tests, the RS introspects one Access Token only, i.e., `token-1`.

In the configuration ua-i15, shown in Fig. 10a, the Client learns about the Access Token revocation when it receives from the RS a 4.01 response to a request to access a protected resource. From the figure, we observe that the Re-Admission Time (t_{ReA}, green bar) is about 940 ms. Since, in our tests, the Client sends a request every second, the Re-Admission Time is expected to be, on average, half the request interval (0.5 s), plus the Client Experience Time (t_{CEx}), which is about 400 ms.

The results obtained with configuration p15-p15 (shown in Fig. 10b) are not so straightforward since they are the product of an average of two very different scenarios, i.e., rsFirst and cFirst. However, we note that in Fig. 10b, for both the RS and the Client, the time to learn about the Access Token revocation is about 7.5 seconds as expected (see the brown and light blue arrows in the figure). Figure 11 helps to better understand these results: Fig. 11a refers to the tests with configuration p15-p15 that relate to the scenario cFirst (49 %), while Fig. 11b refers to the tests that, instead, relate to the other scenario, rsFirst (51 %). In Fig. 11a, the Re-Admission Time starts with the Client Experience Time, therefore $t_{1,\text{ReA}} = t_{1,\text{CEx}}$, while in Fig. 11b, the Re-Admission Time starts when the Inconsistency Time ends, hence $t_{1,\text{ReA}} = t_{2,\text{Inc}}$, as per the Re-Admission Time definition (see Sect. 5.2). On the other hand, the results displayed in Fig. 10b are averaged, and the final outcome is a long Inconsistency Time as well as a long Re-Admission Time, even longer than the one in the configuration ua-i15.

The main problem of configurations ua-i15 and p15-p15 is that, after the revocation has occurred, the Client keeps rely-

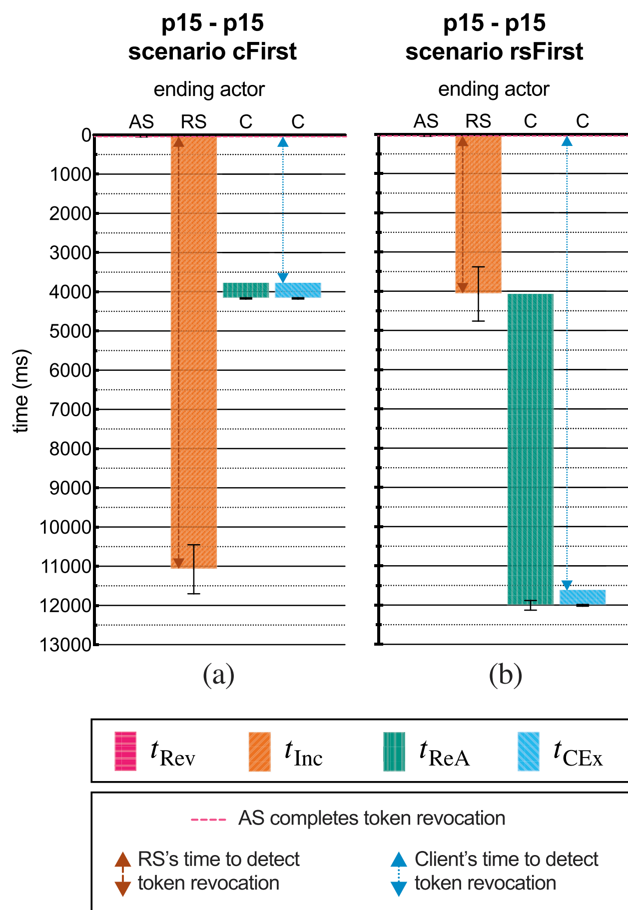

Fig. 11 Time intervals of interest obtained with the configuration p15-p15. Graph **a** shows the average results for the repetitions in which the Client detects the Access Token revocation before the RS. Graph **b** shows the average results for the repetitions in which the RS learns before the Client

ing on `token-1` for successfully accessing the protected resource at the RS—without knowing that it should not do itthat—until the RS learns about the Access Token revocation. This may result in a vulnerability window, since the revocation may have occurred because the RS (or a protected resource on it) has been compromised. For example, the AS in our workflow may revoke the Access Token because it knows that `RES1` on the RS has been compromised by an adversary, in order to send bogus data back to the Client. A Client that keeps accessing the protected resource as per the same Access Token may obtain tainted data or, even worse, be victim of code injection or other attacks.

In configuration o-p15 (Fig. 10c), the use of Observe makes it possible for the Client to learn about the Access Token revocation—through the notifications from the AS—a few milliseconds (about 20 ms) after the AS completes the revocation of `token-1`. In this configuration, the Re-Admission Time and the Client Experience Time are short and coincide, and this allows the Client to promptly re-obtain a grant to access the protected resource `RES2` by means of the second Access Token, i.e., `token-2`. However, we note that, in this configuration and in the configuration o-o, the Client Experience Time is longer than in the other configurations. The reason for this result is subtle: when the AS revokes the Access Token and adds its token hash to the TRL resource, an Observe notification is automatically sent to the registered devices (step 6 of Fig. 8); the Client learns about the token revocation (step 8) a few milliseconds after step 6, and it rapidly makes a new Access Token request. Nonetheless, after step 6, the AS is performing step 7, i.e., it is terminating the sessions at the UCS through two `endAccess` messages. Since the UCS is performing this task, the execution of the new `tryAccess` requests—concerning the second Access Token request—has to be scheduled after the conclusion of step 7. This results in a Client Experience Time slightly longer (about 100 ms longer) than the one obtained with the other configurations.

Note that, with this configuration (o-p15), since the RS polls the TRL resource at the AS every 15 seconds, it learns about the Access Token revocation at a later time, and this causes the Inconsistency Time to be long, as in the previous configurations (ua-i15 and p15-p15). With our assumption of an honest Client, this is not so critical, but it is worth noting that a misbehaving Client could keep accessing the protected resource `RES1` as per `token-1` for some time (about 7.5 seconds on average in our case) after the token revocation, when it is actually not supposed to.

The configuration p15-o (Fig. 10d) is reversed compared to the previous one, i.e., the Client learns about the Access Token revocation after the RS. The long Re-Admission Time depends on the polling interval and on the duration Client Experience Time. In particular, it takes the Client half of the polling interval before learning about the token revocation and making the second Access Token request (see the light blue arrow). This time interval and the consecutive Client Experience Time determine the long Re-Admission Time. However, this configuration results in a short Inconsistency Time, which is good since `token-1` ceases to be accepted by the RS very early, i.e., a few milliseconds after the AS completes the Access Token revocation.

The last configuration (o-o), shown in Fig. 10e, makes use of Observe only. Both the RS and the Client are notified of the Access Token revocation after a few milliseconds from its completion at the AS. Despite some of the tests we conducted relate to the scenario cFirst and some to the scenario rsFirst, Fig. 10e shows that the difference between the two learning times in both scenarios is always less than 10 ms, thus a more detailed analysis like the one made for the configuration p15-p15 is unnecessary.

The configurations with the RS observing the TRL resource minimize the Inconsistency Time, thus ensuring that access requests to protected resources made after the Access

Token revocation and referring to token-1 are promptly not accepted by the RS. Moreover, the configurations with the Client observing the TRL resource minimize the Re-Admission Time and ensure that a Client promptly stops accessing possibly compromised resources.

6.2 UCON time performance

The time spent by the AS to issue and revoke an Access Token varies with: (i) the number of UCPs evaluated by the UCS; and (ii) the *evaluation complexity* of a single UCP. In turn, the evaluation complexity varies with: (i) the number of PIPs managed by the UCS; (ii) the number of attributes present in the pre-(only for token issuing) and ongoing-sections of the UCP; and (iii) the section's shape, i.e., how logical operators, such as AND and OR, combine the checks performed on attributes. In Sect. 6.2.1, we assess the time employed by the UCS to carry out the authorization tasks varying the complexity of the UCP, while in Sect. 6.2.2 we study how the number of UCPs to be evaluated affects the time intervals of interest.

6.2.1 Varying the evaluation complexity

The number of PIPs managed by the UCS determines the number of steps that have to be performed in order to enrich a UCON request. We recall that a UCON request is enriched every time it has to be evaluated against any section of a UCP, and that the values of all the attributes are retrieved by the PIPs from the AMs. Moreover, the number of attributes present in the condition of each section of the UCP, as well as its shape, affects the time spent by the PDP for producing an authorization decision. For example, a condition shaped as a Boolean OR between the attributes has in general better time performance than a condition shaped as a Boolean AND, because the PDP must evaluate all the attributes before producing the authorization decision. Of course, having fewer attributes in a condition results in a faster evaluation.

In this set of experiments, we evaluate the UCS performance to accomplish its internal phases, that we refer to as the *tryAccess phase*, the *startAccess phase*, and the *revokeAccess phase*. The tryAccess phase starts when the PEP sends the tryAccess message to the UCS and ends after it receives the permitAccess message. The startAccess phase starts when the PEP sends the startAccess message to the UCS and ends after it receives the permitAccess message. Since a startAccess message is sent by the PEP upon the reception of a pre-decision of Permit, and thus our flow slightly differs from typical one of the UCON framework [11], we merge and show in Fig. 12 the times of the tryAccess and startAccess phases together, and we call this the *(try+start)Access phase*. The revokeAccess phase starts when the attribute's value changes and ends after the PEP receives the revokeAccess message.

This set of experiments slightly deviates from the workflow described in Sect. 5.1: (i) in Phase 1, the Client makes an Access Token request including the scope "RES1" that translates into one UCON request only, for the resource RES1; and (ii) in Phase 1*, the workflow ends when the Client receives the response from the AS, which does not include an Access Token since the Client cannot be granted access to RES1 after the value of the attribute attr1 changed in Phase 2. Therefore, the duration of the (try+start)Access phase is recorded during Phase 1, and the duration of the revokeAccess phase is recorded during Phase 2. This deviation from the original workflow allows us to measure the performance of the UCS for the evaluation of a single UCON request.

We perform various tests varying the number of PIPs. We assume that each PIP is responsible for retrieving the current value of a single mutable attribute, so the number of PIPs corresponds to the number of mutable attributes managed by the UCS ($n_{m.attr}$). The AMs accessed by the PIPs are files stored at the AS and containing the values of the attributes. In each test, we create a UCP with a fixed number of non-mutable attributes (three) in the condition of the pre-section, and a variable number of mutable attributes ($n_{m.attr}$) in the condition of the ongoing-section. The values of the non-mutable attributes are taken from the original UCON request, while the values of the mutable attributes are retrieved by the PIPs. The condition of the pre-section is shaped as a Boolean AND between 3 attributes, and the condition of the ongoing-section is shaped as a Boolean AND between $n_{m.attr}$ mutable attributes; for example, if we test $n_{m.attr} = 10$, the pre-section contains three ANDed attributes, and the ongoing-section contains ten ANDed mutable attributes. In order for the ongoing-decision to be Permit, the value of each attribute in the ongoing-section must match the current value of the attribute contained in the enriched UCON request. Also, note that, when we test $n_{m.attr} = 10$, the UCS manages ten PIPs, hence a UCON request is enriched with the current value of ten mutable attributes.

Figure 12 shows the results for up to 40 mutable attributes in the ongoing-section of the UCP.

As expected, we observe that the time employed for both phases grows with the number of attributes managed by the UCS, and consequently, with the policy complexity. However, we point out that the increase of time with the number of attributes is not significant: at the two tested extremes, i.e., 1 and 40 attributes, the time difference is about 75 ms for the (try+start)Access phase and less than 50 ms for the revokeAccess phase.

In our tests, the request enrichment phase is very quick, since all the AMs reside on the AS (i.e., the PIP and the AM are on the same machine), and the PIPs rapidly retrieve the

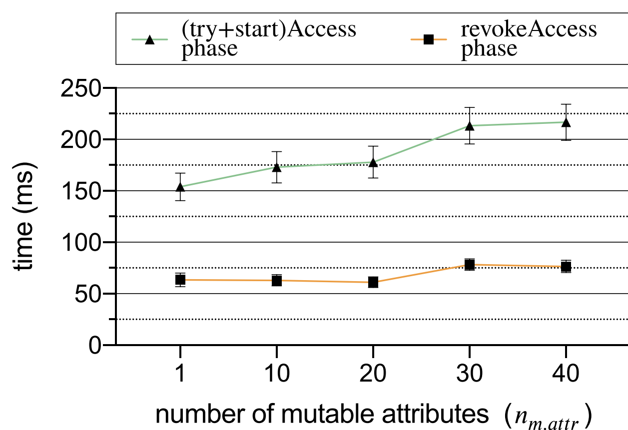

Fig. 12 UCS time performance varying the number of mutable attributes $n_{m.attr}$ managed by the UCS. The number of attributes in the pre-section is fixed to three, while the number of mutable attributes in the ongoing-section is equal to $n_{m.attr}$. The graph shows 95 % confidence intervals

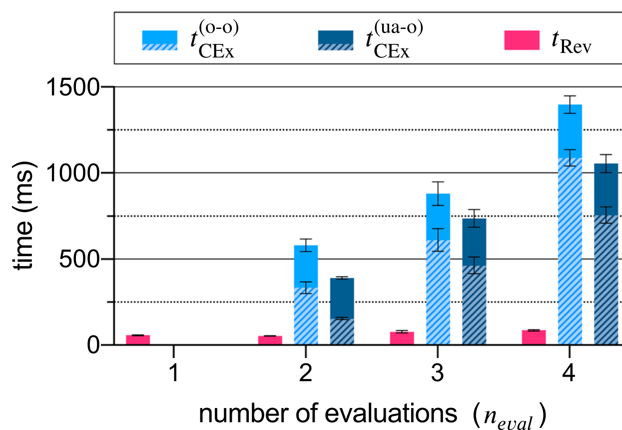

Fig. 13 Results varying the number of evaluations performed by the UCS. The graph shows the average Revocation Time (t_{Rev}), and the Client Experience Time for a configuration where the Client learns of the token revocation through the Observe extension of CoAP ($t^{(o\text{-}o)}_{CEx}$) and a configuration in which the Client learns of the token revocation at a later time ($t^{(ua\text{-}o)}_{CEx}$). The striped portion of the bars represents the time spent by the UCS for carrying out the authorization task

current values of the attributes. Nonetheless, if remote AMs are employed (e.g., files on a different host, remote databases, etc.), we expect that these times increase linearly with the number of remote calls and are affected by the means (e.g., REST APIs, WebSockets, etc.) that the PIPs use to retrieve the current values of the attributes. However, due to the very large number of technical solutions that could be adopted to implement an AM, this analysis is out of the scope of this paper.

6.2.2 Varying the number of policy evaluations

As explained in Sect. 4.1, the scope specified by the Client during the Access Token request is translated by the AS into one or more UCON requests. Every UCON request is enriched and then evaluated by the PDP, which produces a pre-decision and an ongoing-decision for each one. In this section, we evaluate the Client Experience Time and the Revocation Time as the number of UCON requests to be evaluated (n_{eval}) varies. To this aim, the Client specifies a scope that translates into a number of UCON requests ranging from 1 to 4. As in the tests of Sect. 6.2.1, all the UCPs used have 3 attributes in the pre-section. Instead, the number of mutable attributes in the ongoing-section is fixed to 1, and each UCP uses a distinct mutable attribute. Hence, the UCS manages a number of mutable attributes equal to the number of UCON requests the scopes translates into. For example, when we test $n_{eval} = 4$, the UCS manages 4 mutable attributes, and each of the 4 UCPs contains 3 attributes in the pre-section and 1 mutable attribute in the ongoing-section.

We run the tests for two configurations to compare the results in the case where the Client detects the token revocation quickly through the Observe extension of CoAP (configuration o-o), and in the case where the Client learns at a later time that a token revocation has occurred (configuration ua-o). As discussed in Sect. 6.1, when the Client quickly gains knowledge of the revocation of `token-1` and promptly makes the second Access Token request, the Client Experience Time is prolonged since the UCS is still processing the `endAccess` messages from the PEP after the revocation of `token-1`.

Figure 13 shows the results obtained from 100 independent repetitions for each configuration, and for n_{eval} in the range [1..4]. In the test with $n_{eval} = 2$, the Client makes an Access Token request specifying the scope "`RES1 RES2`", which translates into two UCON requests, exactly like in the workflow of Sect. 5.1.

In the tests with $n_{eval} = 1$, the Client makes an Access Token request for the scope "`RES1`", which the AS translates into one UCON request. In Phase 1, both the pre-decision and ongoing-decision results are Permit, so `token-1` is issued, and the resource `RES1` is granted to the Client. However, after `token-1` is revoked and the Client makes the second Access Token request, the AS does not issue `token-2` since, after the revocation of `token-1`, the Client has no access privileges to access other resources. Therefore, the test terminates early, i.e., after step 10 of Phase 1*. This is the reason why, in Fig. 13, the value of the Client Experience Time for $n_{eval} = 1$ is missing.

From Fig. 13, we note that the Revocation Time (t_{Rev}) slightly increases as n_{eval} grows, and, consistently with Fig. 10, its value is about 55 ms when $n_{eval} = 2$. We recall

Table 2 Percentage of the time spent by the UCS for the authorization tasks with regard, with respect to the overall Client Experience Time

n_{eval}	o-o	ua-o
1	–	–
2	57.5 %	39.2 %
3	69.5 %	62.7 %
4	77.8 %	71.7 %

that the Revocation Time consists of the time required for: (i) the revokeAccess phase, i.e., the time required to the UCS to detect that an attribute's value has changed, the time required for the policy re-evaluation, and the reception of the `revokeAccess` message by the PEP; plus (ii) the time employed for the enforcement of the revocation by the AS through Access Token deletion and the addition of its token hash to the TRL. What makes the Revocation Time grow with n_{eval} is the time required for the policy re-evaluation within the revokeAccess phase. As explained in Sect. 6.2.1, the duration of the revokeAccess phase grows with $n_{m.attr}$, which in turn grows with n_{eval}.

The results concerning the Client Experience Time show that t_{CEx} for the configuration o-o (light blue bars and denoted by the symbol $t_{\text{CEx}}^{(\text{o-o})}$) is always higher than the one for the configuration ua-o (dark blue bars and denoted by the symbol $t_{\text{CEx}}^{(\text{ua-o})}$). In particular, we note that the portion of the Client Experience Time concerning the UCS (the lower part of the bars, denoted by stripes) is always higher for the configuration in which the Client promptly gains knowledge of the token revocation (configuration o-o). As already mentioned, this is because, when the PEP sends a new `tryAccess` message related to `token-2` to the UCS (step 2 of Phase 1*), the UCS is still processing the `endAccess` messages related to `token-1`.

From the figure, we note that as n_{eval} grows, the time spent by the UCS for the authorization tasks grows, irrespective of the configuration. However, the non-striped part of the bars, which is related to the network communications between the Client and both the AS and the RS, is independent of n_{eval}. Therefore, the portion of t_{CEx} concerning the UCS increases with the number of evaluations, as shown in Table 2, where its impact is reported as a percentage of the Client Experience Time.

7 Related work

Due to the continuous increase in the spreading of IoT devices in many aspects of our everyday life (e.g., smart homes, smart cities, connected vehicles, etc.), growing attention is being given to their security. As a matter of fact, plenty of works concerning the study of suitable access control systems tailored for the typical features of IoT devices can be found in the scientific literature, and a number of surveys has in turn analyzed them [28–30]. However, to the best of our knowledge, very few such works take into account UCON as access control model, although IoT devices are typically used in dynamic contexts.

The authors of [31] propose the adoption of the UCON model for enhanced authorization in IoT environments, but their work is quite high level, since it describes a possible architecture for the access control mechanism and demonstrates how the authorization process would work with a reference example. However, it does not present an implementation of the proposed architecture, nor does it provide experimental results to prove the feasibility of the proposed approach on IoT devices.

In [32], the authors propose LUCON, a data-centric and UCON-based framework aimed at controlling the flow of messages among IoT devices. The LUCON framework uses a customized language for defining usage control policies (LUCON DSL) that are compiled in Prolog programs to be enforced. With respect to our work, the LUCON framework differs because it is not aimed at regulating the access to resources residing on IoT devices, but it is focused on regulating the data flow among IoT devices.

In [1], the authors present an integration of the UCON framework with the IFTTT (If This Then That) application-level standard for enforcement of obligations. A seamless integration of the UCON framework in the target architecture is proposed, along with a reference implementation and an experimental evaluation. However, differently from our work, the integration is done with a web-service-based application model, and the focus is shifted to the obligation formalism, which is not addressed in the current work.

The work in [2] presents an integration of the UCON framework in the Message Queue Telemetry Transport (MQTT) publish-subscribe network protocol [33], with the UCS integrated in the MQTT broker, and the MQTT clients embedding the PEPs. The work is at a completely different level of maturity compared with the one presented in this paper, as it does not integrate with an already existing access control standard mechanism such as the ACE framework.

Many other works concerning authorization in IoT are based on the Role-Based Access Control (RBAC) model [34], such as [35–37], or on the Attribute-Based Access Control (ABAC) model [38], even adapted for IoT, such as [39–41], and, hence, they do not support continuous policy

evaluation for the detection of ongoing accesses that should be revoked.

8 Conclusions

In this paper, we have proposed the employment of the Usage Control framework as an underlying access control tool for the Authorization Server of the standard ACE framework for access control in the IoT, and we have assessed its performance in terms of time required to issue and revoke Access Tokens.

Moreover, we have implemented and evaluated the additional revoked token notification method, as relying on the Observe extension for the CoAP protocol, in order to automatically notify both Clients and Resource Servers about pertaining Access Tokens that have been revoked earlier than their natural expiration.

Our experimental results show how this method reduces both the time interval during which illegitimate accesses to protected resources can occur after the revocation of an Access Token, as well as the time experienced by Clients and Resource Servers to learn about the revocation of their pertaining Access Token.

Finally, an evaluation of the Usage Control performance has been performed, as a further set of experiments. This allowed us to study how the complexity of access policies and the number of evaluations to perform affect both the time spent by the Authorization Server to make authorization decisions for issuing an Access Token and the time spent by the Authorization Server to revoke an Access Token after a change of dynamic conditions that invalidates current access grants. Although the presented implementation specifically relies on the Usage Control paradigm, the proposed methodology is general enough to be adaptable to other Access Control management systems which exploit Policy Enforcement Points.

Future works will consider further extensions and assessments of the integrated Access and Usage Control framework, when relying on alternative profiles of the ACE framework, diff-based notifications of revoked access rights; as well as the use of (automatic) notification of revoked access rights in different frameworks for authentication and authorization enforcing access and usage control.

Appendix A

Table 3 provides a list of the abbreviations used throughout this paper.

Table 3 List of abbreviations

Abbreviation	Description
ABAC	Attribute-Based Access Control
ACE	Authentication and Authorization for Constrained Environments
AM	Attribute Manager
AS	Authorization Server
C	Client
CBOR	Concise Binary Object Representation
CH	Context Handler
CoAP	Constrained Application Protocol
COSE	CBOR Object Signing and Encryption
CWT	CBOR Web Token
DTLS	Datagram Transport Layer Security
EDHOC	Ephemeral Diffie-Hellman Over COSE
IFTTT	If This Then That
IoT	Internet of Things
IQR	Interquartile Range
JSON	JavaScript Object Notation
JWT	JSON Web Token
MQTT	Message Queue Telemetry Transport
NTP	Network Time Protocol
OAuth	Open Authorization
OSCORE	Object Security for Constrained RESTful Environments
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
RBAC	Role-Based Access Control
REST	Representational State Transfer
RS	Resource Server
SID	Session Identifier
SM	Session Manager
TRL	Token Revocation List
UCON	Usage Control
UCP	Usage Control Policy
UCS	Usage Control System
UDP	User Datagram Protocol
XACML	eXtensible Access Control Markup Language

Acknowledgements The authors sincerely thank the anonymous reviewers and the Associate Editor for their insightful comments and suggestions, which have helped improve the technical and editorial quality of the manuscript. This work has been partially supported by: the Sweden's Innovation Agency VINNOVA within the EUREKA CELTIC-NEXT project CYPRESS; the H2020 project SIFIS-Home (grant agreement 952652); and the SSF project SEC4Factory (grant RIT17-0032).

Funding Open access funding provided by Consiglio Nazionale Delle Ricerche (CNR) within the CRUI-CARE Agreement.

Data availability The code used to generate the results is publicly available at the referenced repository on Bitbucket. The raw data generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors do not have any Conflict of interest with any party which might be relevant to the result of this study.

Ethical approval We ensure compliance with ethical principles and guidelines throughout our study. The current study had not as object human participants, nor animals. All contributors of this work have been properly acknowledged through authorship. Participation and possibility to provide contributions to the current work has not been biased by traits such as gender, religious belief, or political orientation of the people involved.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Giorgi, G., La Marra, A., Martinelli, F., Mori, P., Rizos, A., Saracino, A.: Exploiting if this then that and usage control obligations for smart home security and management. Concurr. Comput. Pract. Exp. **34**(16), e1689 (2022). https://doi.org/10.1002/cpe.6189
2. La Marra, A., Martinelli, F., Mori, P., Rizos, A., Saracino, A.: Introducing usage control in MQTT. In: Computer Security - ESORICS 2017 International Workshops, CyberICPS 2017 and SECPRE 2017, Oslo, Norway, September 14-15, 2017, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 10683, pp. 35–43. Springer (2017). https://doi.org/10.1007/978-3-319-72817-9_3
3. Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., Tschofenig, H.: Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth). RFC 9200 (2022)
4. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749 (2012)
5. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014)
6. Bormann, C., Hoffman, P.E.: Concise Binary Object Representation (CBOR). RFC 8949 (2020)
7. Schaad, J.: CBOR Object Signing and Encryption (COSE): Structures and Process. RFC 9052 (2022)
8. Schaad, J.: CBOR Object Signing and Encryption (COSE): Initial Algorithms. RFC 9053 (2022)
9. Tiloca, M., Palombini, F., Echeverria, S., Lewis, G.: Notification of Revoked Access Tokens in the Authentication and Authorization for Constrained Environments (ACE) Framework. Internet-Draft draft-ietf-ace-revoked-token-notification-08, Internet Engineering Task Force (2024). Work in Progress
10. Hartke, K.: Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641 (2015)
11. Carniani, E., D'Arenzo, D., Lazouski, A., Martinelli, F., Mori, P.: Usage control on cloud systems. Futur. Gener. Comput. Syst. **63**, 37–55 (2016)
12. Park, J., Sandhu, R.: The $UCON_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. (TISSEC) **7**(1), 128–174 (2004)
13. eXtensible Access Control Markup Language (XACML) version 3.0 plus errata 01 (2017). http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html
14. Shelby, Z., Hartke, K., Bormann, C., Frank, B.: RFC 7252: The constrained application protocol (CoAP). Internet Engineering Task Force (IETF) (2014)
15. R. T. Fielding and R. N. Taylor: Architectural styles and the design of network-based software architectures. Doctoral dissertation (2000)
16. Postel, J.: User Datagram Protocol. RFC 768 (1980)
17. Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. RFC 6347 (2012)
18. Selander, G., Mattsson, J.P., Palombini, F., Seitz, L.: Object Security for Constrained RESTful Environments (OSCORE). RFC 8613 (2019)
19. Gunnarsson, M., Brorsson, J., Palombini, F., Seitz, L., Tiloca, M.: Evaluating the performance of the OSCORE security protocol in constrained IoT environments. Internet of Things **13**, 100333 (2021)
20. Palombini, F., Seitz, L., Selander, G., Gunnarsson, M.: The Object Security for Constrained RESTful Environments (OSCORE) Profile of the Authentication and Authorization for Constrained Environments (ACE) Framework. RFC 9203 (2022)
21. Selander, G., Mattsson, J.P., Palombini, F.: Ephemeral Diffie-Hellman Over COSE (EDHOC). RFC 9528 (2024)
22. Jones, M., Wahlstroem, E., Erdtman, S., Tschofenig, H.: CBOR Web Token (CWT). RFC 8392 (2018)
23. Jones, M., Seitz, L., Selander, G., Erdtman, S., Tschofenig, H.: Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs). RFC 8747 (2020)
24. Jones, M., Bradley, J., Sakimura, N.: JSON Web Token (JWT). RFC 7519 (2015)
25. Jones, M., Bradley, J., Tschofenig, H.: Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs). RFC 7800 (2016)
26. Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259 (2017)
27. Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keränen, A., Hallam-Baker, P.: Naming Things with Hashes. RFC 6920 (2013)
28. Qiu, J., Tian, Z., Du, C., Zuo, Q., Su, S., Fang, B.: A survey on access control in the age of internet of things. IEEE Internet Things J. **7**(6), 4682–4696 (2020). https://doi.org/10.1109/JIOT.2020.2969326
29. Ravidas, S., Lekidis, A., Paci, F., Zannone, N.: Access control in Internet-of-Things: A survey. Journal of Network and Computer Applications **144**, 79–101 (2019) https://doi.org/10.1016/j.jnca.2019.06.017. https://www.sciencedirect.com/science/article/pii/S108480451930222X

30. Ouaddah, A., Mousannif, H., Abou Elkalam, A., Ait Ouahman, A.: Access control in the internet of things: Big challenges and new opportunities. Computer Networks **112**, 237–262 (2017) https://doi.org/10.1016/j.comnet.2016.11.007. https://www.sciencedirect.com/science/article/pii/S1389128616303735
31. Zhang, G., Gong, W.: The research of access control based on UCON in the internet of things. J. Softw. **6**(4), 724–731 (2011). https://doi.org/10.4304/jsw.6.4.724-731
32. Schuette, J., Brost, G.S.: Lucon: Data flow control for message-based iot systems. In: 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pp. 289–299 (2018). https://doi.org/10.1109/TrustCom/BigDataSE.2018.00052
33. Banks, A., Briggs, E., Borgendale, K., Gupta, R.: OASIS Standard MQTT Version 5.0 (2019). http://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html
34. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29**(2), 38–47 (1996). https://doi.org/10.1109/2.485845
35. Fernández, F., Alonso, A., Marco, L., Salvachúa, J.: A model to enable application-scoped access control as a service for IoT using OAuth 2.0. In: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), pp. 322–324 (2017). https://doi.org/10.1109/ICIN.2017.7899433
36. Barka, E., Mathew, S.S., Atif, Y.: Securing the web of things with role-based access control. In: El Hajji, S., Nitaj, A., Carlet, C., Souidi, E.M. (eds.) Codes, Cryptology, and Information Security, pp. 14–26. Springer International Publishing, Cham (2015)
37. Jindou, J., Xiaofeng, Q., Cheng, C.: Access control method for web of things based on role and SNS. In: 2012 IEEE 12th International Conference on Computer and Information Technology, pp. 316–321 (2012). https://doi.org/10.1109/CIT.2012.81
38. Vincent C. Hu David, F., Rick, K., Adam, S., Sandlin K. Robert, M., Karen, S.: Guide to attribute based access control (ABAC) definition and considerations (2014)
39. Hussein, D., Bertin, E., Frey, V.: A community-driven access control approach in distributed IoT environments. IEEE Commun. Mag. **55**(3), 146–153 (2017). https://doi.org/10.1109/MCOM.2017.1600611CM
40. Ray, I., Alangot, B., Nair, S., Achuthan, K.: Using attribute-based access control for remote healthcare monitoring. In: 2017 Fourth International Conference on Software Defined Systems (SDS), pp. 137–142 (2017). https://doi.org/10.1109/SDS.2017.7939154
41. Alshehri, A., Sandhu, R.: Access control models for cloud-enabled Internet of Things: A proposed architecture and research agenda. In: 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), pp. 530–538 (2016). https://doi.org/10.1109/CIC.2016.081

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.