Access Control Enforcement Architectures for Dynamic Manufacturing Systems

Björn Leander
School of Innovation, Design and Engineering, Mälardalen University
and
Process Automation, ABB AB
Västerås, Sweden
bjorn.leander@mdu.se

Aida Čaušević
School of Innovation, Design and Engineering, Mälardalen University
and
Alstom AB
Västerås, Sweden
aida.causevic@alstomgroup.com

Tomas Lindström
Process Automation, ABB AB
Västerås, Sweden
tomas.lindstrom@se.abb.com

Hans Hansson
School of Innovation, Design and Engineering, Mälardalen University
Västerås, Sweden
hans.hansson@mdu.se

***Abstract*—Industrial control systems are undergoing a transformation driven by business requirements as well as technical advances, aiming towards increased connectivity, flexibility and high level of modularity, that implies a need to revise existing cybersecurity measures. Access control, being one of the major security mechanisms in any system, is largely affected by these advances.**

In this article we investigate access control enforcement architectures, aiming at the principle of least privilege[1] in dynamically changing access control scenarios of dynamic manufacturing systems. Several approaches for permission delegation of dynamic access control policy decisions are described. We present an implementation using the most promising combination of architecture and delegation mechanism for which available industrial standards are applicable.

***Index Terms*—Cybersecurity, Access Control, Industrial Automation and Control Systems, Dynamic Manufacturing**

I. Introduction

Manufacturing systems are undergoing a transformation driven by business requirements as well as technical advances [2], [3], [4]. One direction of development is towards systems which can easily be adapted to changing business requirements, fast innovation cycles and mass customization. Such systems are dynamic by their nature, as both the system composition and functionality is changing over time.

Access control in industrial automation systems. Traditionally, cybersecurity in industrial automation system has focused on physical and logical perimeter protection of strictly hierarchical systems [5]. Access control is therefore typically limited to role-based access control at the higher level components for human operator and engineering staff [6].

This work is supported by ABB AB; the industrial postgraduate school Automation Region Research Academy (ARRAY), funded by The Knowledge Foundation; and the European Union's Horizon 2020 ECSEL JU project InSecTT under grant agreement No 876038. The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

[1]The principle of least privilege dictates that no entity should have privileges beyond what is required to fulfill its task [1].

There is a trend in manufacturing system architectures, transitioning from the hierarchical controller-centric model, towards a network-centric design strategy using a common network back-bone [7]. This implies a need for a unified access control mechanism, spanning the entire system.

Modular Automation [8], [9], [10] is one example of an emerging design strategy for process industries, which exhibits three distinct levels of dynamicity:

1) Dynamic system composition - available processing modules and how they are interconnected will change over time, due to changing high-level production requirements.
2) Dynamic production schemes - available and active recipes describing the production workflow and synchronization change on a daily basis, based on business requirements.
3) Dynamic operations - during recipe execution, different steps of the recipe-workflow are activated, implying different processing operations being executed.

To follow the least-privilege principle, the dynamic properties of the emerging manufacturing systems should ideally be mirrored by the access control mechanisms. This could be achieved using different methods of dynamic access control [11], [12], which provide active permissions adaptable to system changes over time, following, e.g., active workflows or environmental conditions. Several dynamic policy models have been proposed for use within industrial systems, but only a few of them consider the enforcement architecture (further discussed in Section II).

Problem statement. At the moment, dynamic access control is not widely adopted in manufacturing systems, but for evolving system types it is highly relevant, due to the inherently dynamic nature of these systems. Assuming a dynamic access control policy framework is in place, the mechanisms for enforcing the stated policies must be equally dynamic. They must be practically useful in a manufacturing context, with resources often protected by constrained devices. To the best of our knowledge, available enforcement architectures cannot provide such characteristics.

Paper objectives. This article focuses on constructing an access control enforcement architecture that:

1) Allows dynamic access control.
2) Is applicable to industrial manufacturing scenarios.
3) Uses available industrial standards for communication.

The goal of the architecture is to be practically applicable in real industrial systems, being potentially interoperable with COTS components. Therefore, adhering to a selected set of available technical standards is a requirement.

Combining objectives 1 and 2 is the main challenge, as we need a mechanism which can make dynamic, secure and timely policy decisions enforced on devices responsible for safety critical operations in the manufacturing systems. The mechanisms have to be light-weight and scale well with regards to communication-load.

Contributions. The following are our main contributions:

- Presentation of four enforcement architectures, along with an evaluation of their respective usefulness for enforcing dynamic access control in manufacturing systems (Section III).
- Four approaches for communication of dynamic access control policy decisions within the architecture, using JSON Web Tokens (JWT) [13] (Section IV).
- Implementation of an enforcement architecture, using one of the suggested JWT formats and the Open Process Communication Unified Automation (OPC UA) [14] framework (Section V).
- A verification of the provided implementation, using propositional logic (Section VI).

The results are discussed in Section VII, and conclusions and ideas for potential future work are presented in Section VIII.

II. Background and Related Work

The unique challenges and requirements for access control in Industrial Internet of Things and Smart Manufacturing systems are discussed in some previous works, e.g., by Salonikas *et al.* [15] and Leander *et al.* [16]. These works both describe the need for mechanisms that are resource efficient and scalable, while being flexible and able to make decisions based on context awareness or executing workflows, in contrast to the access control architectures currently used in industrial control systems, which are typically static and management-intensive to update. Our approach aims to fulfill the described requirements from an architectural point of view.

There are several authorization policy models able to handle dynamic scenarios, coming from two main policy model-families: Attribute Based Access Control (ABAC) [17] and Task Based Access Control (TBAC) [18]. In ABAC, the dynamic behavior is captured by changing attributes of subjects, resources and the environment. In TBAC, a workflow-engine keeps track of task states and transitions, thus limiting allowed activities to what is mandated by the workflows.

In the literature, TBAC is limited to operations within a closed Process Aware Information System (PAIS), e.g., a document handling system or banking system [19], making available solutions less useful in manufacturing scenarios, however, there are examples of transforming formalized workflows into ABAC-rules, e.g., as proposed by Leander *et al.* [20], or extending access control matrices with workflow-data as suggested by Knorr [11].

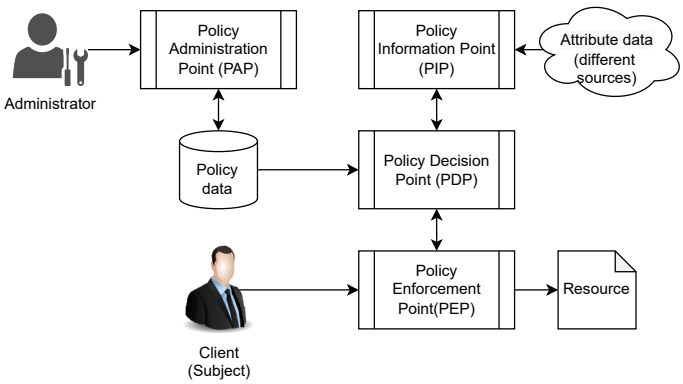


Fig. 1: Basic Elements of the XACML Reference Architecture.

Similarly as in this article, the reference architectures part of the eXtendable Access Control Modeling Language (XACML) [21], [22] (Fig. 1) is often used as a basis for enforcement architecture descriptions related to ABAC. The architecture contains a Policy Administration Point (PAP), providing administrative functionality, i.e., updating of Policy Data; Policy Information Points (PIP) to provide interfaces for querying attribute data; Policy Decision Points (PDP) that contain the logic for deciding if a resource request from a subject shall be granted or denied; and Policy Enforcement Points (PEP) that enforce the policy decision [23].

Chandramouli *et al.* [24] describe an enforcement architecture for micro-services applications executing in containerized environments, with a side-car container acting as a common PEP. Martinelli *et al.* [25] describe an enforcement architecture for OPC UA, tailor-fit for the Usage Control (UCON) [26] access control model, an extension of ABAC, which supports mutable attributes. Bhatt *et al.* [27] describe ABAC for Amazon Web Services (AWS) Internet of Things (IoT), with PEP and PDP placed in edge-nodes, aiming for usability in the cloud-connected factories of the future.

While all these approaches are related to our work, none of them is directly applicable and interoperable in dynamic manufacturing systems. Our approach focuses on providing an architecture that can enforce dynamic access control decisions on constrained resource servers, while following available industrial standards.

OPC UA [14], part IEC 62541-4, describes a session-based application-layer protocol for client-server communication that provide integrity and confidentiality using mechanisms similar to Transport Layer Security [28] (TLS)[2]. It also contains

[2]OPC UA also supports the publish/subscribe pattern, out of scope for this paper.

sections covering access control, including several different authentication models, and protocols for authorization. For authorization, the main supported model is Role-Based Access Control (RBAC) [29], described in IEC 62541-2, with static role-permissions defined relative to the resources of a server. User-to-role mapping is either configured at the server, or communicated using access tokens issued by an authorization service.

Using access tokens, such as JWT, is a common mechanism for permission delegation in the IT-system domain, described in the OAuth 2.0 standard [30]. Even though it is supported by OPC UA, it is currently not widely adopted in manufacturing systems.

Some previous works look at different enforcement architecture mechanisms and similar concepts, such as sub-granting techniques, in IoT and Cyber Physical Systems (CPS) [31], surveyed by Sudarsan *et al.* [32], and Dramé-Maigné *et al.* [33]. They provide information on, e.g., current state of the art for authorization governance and permissions delegation. Among other things, Dramé-Maigné *et al.* present four basic architectural concepts for sub-granting: centralized, hierarchical, federated and distributed. Our approach follows a hybrid of the centralized and distributed approach, with centralized policy decisions for dynamic permissions and role assignment, and local decisions for static role-permissions.

III. Enforcement Architecture - proposed models

In order for a set of policy rules to be effectively enforced in a system, the design of the enforcement architecture is crucial. In this section we introduce four different architectures, which we find useful for achieving an enforcement architecture supporting dynamic access control. We reason on the respective model's applicability in a dynamic manufacturing system, based on the previously stated objectives.

Based on the previously introduced XACML reference architecture (Fig. 1) we vary the placements of the PEP and PDP elements, using the following entities:

- **Client** is an entity that wants to access a resource. Clients may be operators (human users supervising the process), digital units executing high-level control operations, data-historian services, sibling modules, etc. Clients that represent human users as well as purely digital entities are possible.
- **Resource Server** is a service providing logical access to a set of resources. In the context of a dynamic manufacturing system, it may be a direct-connected sensor, an interaction point for a manufacturing module, a purely digital service, etc.
- **Authorization Service** (AS) is a service that contains logic and interfaces for policy decisions, i.e., facilitating outsourced policy decisions [23].

Our focus is on policy decision and enforcement, hence the PAP and PIP are omitted in the following, assuming that the PDP can access the policy data it needs, and that secure and efficient administration of policy data is provided.

The client is seen as potentially untrusted, and should therefore not contain any of the architectural elements. However, clients can be used for mediation of information on policy decisions, i.e., transport policy decision data from a PDP to a PEP, provided that the PEP can validate the integrity of the data. Following by the principle of full mediation [1], the policy enforcement mechanism must be kept close to the resources to minimize the risk of alternative routes for access (i.e., back-doors). Therefore we use the limitation that the resource server shall contain a PEP.

Using these entities and limitations, the following architectural models can be constructed:

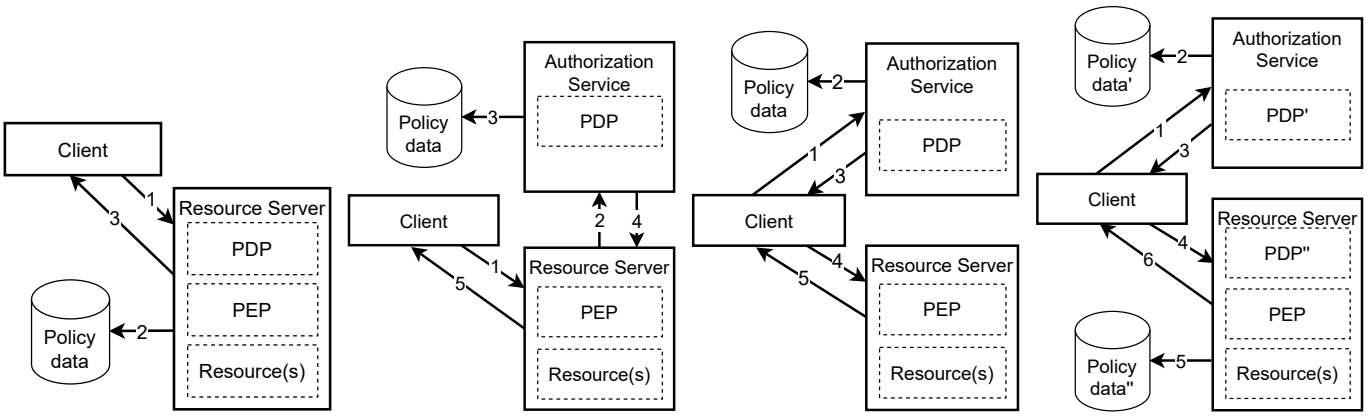
(a) Traditional client-server model (Fig. 2a) - The resource server contains both the PDP and PEP, and must therefore handle all the logic. For this model to be useful for dynamic access control, the policy data contained by the resource server must be reconfigured every time policies concerning the resource server are updated (i.e., policy provisioning). The model is similar to what can be used in OPC Classic[3] which is a common industrial communication protocol. However, OPC Classic authorization is based on Windows OS Access Control Lists (ACL) on DCOM-objects [34], heavily limiting expressiveness and granularity of the policy rules.

(b) Common Open Policy Server (COPS) model (Fig. 2b) - Clients provide credentials to the resource server and all logic for inference of permissions resides in an AS, i.e., the resource server asks an external PDP for each resource request. This is following the same principle as the COPS-protocol [35]. The policy decision is offloaded to an external AS. For dynamic access control, the AS must be updated whenever policy data changes, but the resource server does not need additional configurations. However, the resource server may need to validate user credentials, implying that the resource server needs to maintain the user account database. If client identities change, the resource server must be reconfigured.

(c) OAuth2 model (Fig. 2c) - differs from **(b)** only in that the client communicates the policy decisions. This model is similar to permission delegation based on OAuth2 [30]. When the client makes the resource request, the request itself contains all the logic needed to infer permissions, issued by a trusted AS. Therefore, the resource server does not need any additional configuration to infer dynamic permissions. In this model, the resource server does not need a user database, if it can check that the policy decision is issued for the client, and that the decision is issued by a trusted AS.

(d) OPC UA model (Fig. 2d) - is similar to **(c)**, with additional PDP capabilities in the resource server. This is similar to the OPC UA model described in Section II. For the model to be useful for dynamic access control scenarios, the policy data on either the AS, the resource server or both must be updated as policy-decisions change.

[3]https://opcfoundation.org/about/opc-technologies/opc-classic



(a) Traditional client-server model. All logic in resource server.

(b) COPS-model. Resource server communicates directly w. PDP.

(c) OAuth2-model. PDP in authorization service, PEP in resource server, client mediates.

(d) OPC UA-model. PDP distributed over authorization- and resource-server, client mediates.

Fig. 2: Four authorization architecture models. The numbers indicate the order of messages in respective protocols.

As described, these models are all used in different forms in the state of practice, however, using them for providing enforcement of dynamic access control policies in a manufacturing system is a novel approach and is a contribution of this article.

A. Evaluation of models

As mentioned, all the above architectures are theoretically feasible for dynamic access control. In the following we will attempt to evaluate the models from the perspective of an industrial automation and control system.

Based on the requirements from Section II above, we derive three evaluation criteria for the architectures. In network-centric manufacturing systems, constrained devices may act as resource servers, e.g., simple sensor devices. Therefore a reasonable goal is to minimize workload of the resource server. Furthermore, to enhance scalability and robustness, the access control mechanism dependence on reliable, fast, and deterministic network performance should be reduced. Below we refer to this as network load, where high means that there is a need for high network performance, e.g., caused by a need for many message exchanges (per operation).

The following criteria will therefore be used: 1) execution load of the resource server (lower is better), 2) amount of network traffic involving the resource server (lower is better), and 3) flexibility with regards to dynamic access enforcement (high is better). The evaluation is summarized in Table I.

Architecture	Workload	Network load	Flexibility
(a)	High	Low	Low
(b)	Medium	High	High
(c)	Low	Low	Medium-High
(d)	Medium-Low	Low	Medium-High

TABLE I: An Evaluation of Architectures

Architecture **(a)** puts all work related to policy decision and enforcement, as well as user authentication, at the resource server. However, the network load is low since the only interactions between the client and the resource server are the resource requests. From a flexibility perspective the architecture is rated as "Low", since keeping the local policy data up to date with dynamic policies would require frequent resource server reconfiguration.

In architecture **(b)**, the resource server is the active entity in relation to the PDP and it is responsible for initiating a communication channel for policy decisions. Even though policy decisions and policy data are not needed in the resource server, a lot of communication work is required, putting the workload to "Medium" and network load to "High". This setup will allow high flexibility with regards to dynamic access control, assuming the AS has access to up-to-date policy data.

In architecture **(c)**, the client is responsible for communication with the AS, which removes communication and processing load from the resource server ("Low" on resource server workload and network load). The architecture allows high flexibility with regards to access control, according to the same reasoning as architecture **(b)**. However, as the client is the mediator, the policy decision must come with a validity time-span, and the length of that time-span, in practice limiting the dynamicity of the architecture.

Architecture **(d)** places some policy data in the AS and some in the resource server, and lets the client mediate the AS policy decisions to the resource server, similar to architecture **(c)**. This will put some additional workload onto the resource server, but allows for the same low amount of network traffic and the same level of flexibility as architecture **(c)**. Architecture **(d)** however has an advantage: In case policy decisions by the AS cover more than one resource request, the required work and network load between client and AS will be lower than for architecture **(c)**. In an industrial control system a session between a client and a server will typically include

several interactions and resource requests, prompting several client-AS interactions. Furthermore, it is quite common that, e.g., permission to read data is allowed by most interaction parties, while the permissions to write data or execute methods are restricted until needed.

Architecture **(d)** also has the advantage that it enables an attractive separation of work: policy decisions related to static permissions can be made in the resource server, while policy decisions related to dynamic permissions can be made in the AS.

Based on the collected information, architectures **(c)** and **(d)** are the best choices for the given evaluation criteria. In both cases the client communicates policy decisions from the AS to the resource server. Access tokens is one mechanism for secure communication of policy decisions. In the following section we explore how to use this mechanism for dynamic access control in manufacturing systems.

IV. Access tokens for dynamic access control

In this section we explore different ways of formulating access tokens to support dynamic access control using the previously described architectures **(c)** and **(d)**.

As previously mentioned, an access token must have a validity time to be useful. Deciding on the length of validity for an access token is a trade-off between dynamicity and performance. Too long validity time will lead to outdated policy-decisions being enforced by the resource server, while a too short validity time will lead to a communication overhead, with the risk of network congestion and impaired real-time characteristics. If the token lifetime is of the same order as, or greater than, the dynamicity of the system, a token revocation mechanisms may be needed. If the token lifetime is significantly smaller than the dynamicity of the system, the amount of time when incorrect tokens can be used in the system is small. In the following, we assume that the token validity time is significantly shorter than the average time between the state changes of the system.

Information stated in access tokens is often refereed to as *claims*. In the following section we will present strategies for formulation access token claims, two related to architecture **(c)** and two for architecture **(d)**.

A. Architecture *(c)*: Session-wide explicit permissions

Architecture **(c)** does not group permissions into, e.g., roles and therefore the AS populates the access token with a list of all permissions valid for the session. The token have to contain explicit permissions related to all the relevant resources of the resource server. This means that for a resource server protecting more than a few resources, there is a risk that the token size will become unmanageable.

B. Architecture *(c)*: Specific explicit permissions

A way of countering the issue with growing access tokens expressed above, is to include a set of needed permissions in the access token request, such that the client requests the specific permissions it intends to use. This protocol will limit the content of the token to only necessary information.

C. Architecture *(d)*: Roles with modifiers

Architecture **(d)** allows the resource server to be aware of, e.g., roles that the AS can refer to (using role claims), instead of just listing explicit permissions. Therefore, the access tokens can in many cases be smaller than in architecture **(c)**. Additionally it is possible to combine the dynamic policies with static local permission definitions. The static local permissions can be referred to with role claims, as described above, and the effectively granted set of permissions can be modified by adding explicit permissions or restrictions.

The resource server policy data comprise a static set of roles, mapped to permissions, the AS knows all the resource servers role permissions, and uses this knowledge to populate an access token which combines the roles with explicit permissions or restrictions to mirror the currently active set of permissions. Compared to using only explicit permissions, this allows for a more compact way of communicating claims in the access token, while keeping the flexibility.

D. Architecture *(d)*: Specific explicit permissions with roles

A further development would be to combine the specific explicit permission scheme, as in Section IV-B above, with local static roles. This would result in a protocol were the client requests specific permissions from the AS, and as a response receives a token encoded with the roles that are *fully granted*, i.e., that has no restrictions, and explicit permissions to the requested operations (if granted).

For some types of interactions, this would be a more practical concept. Consider, e.g., all the potential actions an operator is allowed to do in relation to all process objects in a complex manufacturing environment. It would be wasteful to have all of them pre-calculated and put into access tokens for all open sessions[4]. Instead, permissions could be evaluated and granted just-in-time and on demand.

E. OPC UA considerations

To the best of our knowledge, OPC UA is the only industrial communication framework which includes access tokens for permission delegation. However, the authorization flow described in the OPC UA specification is tailored for communicating role-permissions, and therefore carries a major limitation in the client to AS communication protocol: there is no room for context specific information in the access token request sent by the client, and therefore the AS cannot use anything else than the client identity and resource server identity for compiling the content of the token.

To keep the solution interoperable with OPC UA clients, we want to avoid altering the existing protocol for authorization delegation. This means that among the proposed solutions we are limited to session-wide explicit permissions (as described in IV-A) or roles with modifiers (as described in IV-C) to implement dynamic access control in a manufacturing system

[4]Wasteful, considering the execution time in the AS for calculating and compiling never used permissions into tokens, for network utilized for communicating these tokens, and on resource server memory and execution for storing and evaluating the decisions.

utilizing OPC UA. To mitigate the risk of generating oversized access tokens we focus on the approach using roles with modifiers in the implementation.

V. Implementation

This section provides an example on how an enforcement architecture can be implemented, with the goal to demonstrate the feasibility of the suggested approach.

A. Protocol description

For a client to communicate with a resource server, with the goal of accessing resources safeguarded by the resource server following Architecture **(d)** and using permission delegation according to the approach using roles with modifiers (see IV-C) the following protocol is proposed (illustrated in Fig. 3).

First a session between the client and resource server must be created. To activate the session the client needs an access token, which can only be received from an AS trusted by the resource server (see arrow "1. Request token" in Fig. 3). The request includes credentials for the client and the identity of the resource server. The AS evaluates the currently available permissions that the Client holds related to the resources which the resource server safeguards (arrow "2. Evaluate" in Fig. 3), which will result in an Access Control List (ACL) representing all client permissions on resource server resources. The ACL is then compared with resource server local RBAC policies, as detailed below (arrow "3. compare" in Fig. 3), in order to create the access token (arrow "4. Create" in Fig. 3) containing a combination of roles and explicit permissions/restrictions. The token is signed by the AS and returned to the client (arrow "5. Return" in Fig. 3), which uses it to activate the session with the resource server (arrow 6 in Fig. 3). Upon session activation the resource server validates the basic data of the token, e.g., signatures and validity. If the token is valid, the session is accepted, and the resource server stores the token for future policy enforcement tasks. When a resource is requested (arrow 7 in Fig. 3), the resource server enforces policies related to the request using explicit permissions and restrictions from the access token, and permissions related to the granted roles from the local RBAC policy repository. Further details on the sequence of the protocol given in Fig. 5.

B. Access token content

In our approach, the policy-related data in the access token can be defined formally as $token = (id_c, id_s, T_{id}, P_X, Z_X)$, where id_c is a unique identification of the client and id_s is a unique identification of the resource server. T_{id} is a set of identities of zero or more roles known by the resource server, P_X is a set of zero or more explicit permissions of the form $p = (o, r)$ granting execution of an operation o on a resource r. Z_X is a set of zero or more explicit restrictions, written in the same form as permissions, i.e., $z = (o, r)$, but instead implying the restriction that an operation o is not allowed to be executed on the resource r. All the resources, operations, and roles referred by the token are assumed to be known by the resource server.

In the implementation, JWT is used as the container for claims. JWT is encoded in a JSON Web Signature (JWS) [36] or JSON Web Encryption (JWE) [37] structure, i.e., with signed and/or encrypted payload. There are several ways of expressing these as claims in a JWT, but in the implementation, the following are used:

1) **Roles**: an array of strings, representing roles, well-known by the resource server.
2) **Entitlements**: an array of strings, representing explicit permissions related to resources handled by the resource server. These permissions are to be seen as additions to permissions granted implicitly by the Roles claim.
3) **Restrictions**: an array of strings, representing explicit restrictions related to resources handled by the resource server. These permissions are to be seen as exclusions of permissions granted implicitly by the Roles claim.

Example of the claims-section of a JWT, using the above fields:

```
{
  "sub": "client_id",
  "aud": "resource_server_id",
  "name": "readable client name"
  "iat": 1516239022,
  "exp": 1516241022,
  "roles": ["role1", "role2"],
  "entitlements": ["method1", "signal1.read"],
  "restrictions": ["signal1.write"]
}
```

The audience (`aud`) field uniquely identifies the resource server, the subject (`sub`) field identifies the client, the `iat` field is the time of issuing the token, while `exp` is the token expiry time. The `roles` array lists roles, while `entitlements` and `restrictions` list additional permissions and restrictions for the client, respectively. The JWT also contains a signature from the AS, which ensures token integrity and the identity of the token issuer.

C. Policy data and policy decisions

In the following we detail policy data content for the respective component, and how the policy decision points work.

Policy data at the resource server: The resource server s provides the interface to a set of resources R. For each resource $r \in R$, a set of operations O are possible. From the resource server point of view, a permission p is defined as $p = (o, r)$, where $o \in O$, i.e., grant of permission p allows performing an operation o on a resource r. Furthermore, a resource server has a set of roles T, with $t \in T$ being a named collection of permissions $t = (id, P_t)$, where id is a unique identifier for the role and P_t is the set of permissions granted to the role.

Policy data at the AS: The AS has knowledge of each resource servers' internal role definitions, and thus contains data for a set of resource servers S, with associated permission definitions $s = (id, T)$ where id is a unique identification of the resource server and T is the set of roles used by the server s. The AS also contains a policy decision mechanism

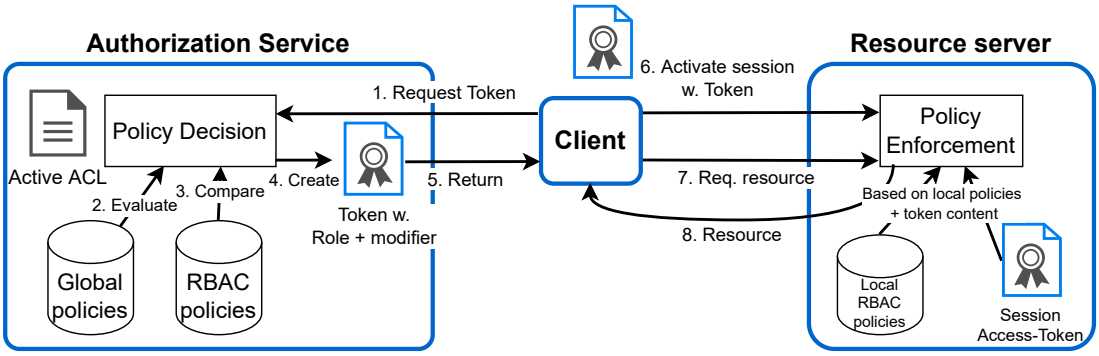


Fig. 3: A Proposed Authorization Protocol

so that, for a client and resource server s, it can construct an access control list P_{ACL} of granted permissions, based on dynamic policy data and rules. P_{ACL} should be interpreted as an exclusive set of granted permissions, and no permissions outside this set are granted, i.e., allow if a permission is in the set, otherwise deny.

Policy decision in authorization service: Upon a request from the client the AS constructs a token containing the combination of roles and explicit permissions or restrictions needed to faithfully represent the set of permissions present in P_{ACL} using Algorithm 1.

Algorithm 1 Populate token

1: $T_{id} := \emptyset, Z_X := \emptyset, P_X := P_{ACL}$
2: $t_i \in T$ for smallest $l_i := \mathbf{max}(\mid t_i.P_t \setminus P_{ACL} \mid, \mid P_X \setminus t_i.P_t \mid)$
3: **if** $l_i < (\mid P_X \mid)$ **then**
4: $\quad T_{id} := T_{id} \cup t_i.id$
5: $\quad Z_X := Z_X \cup (t_i.P_t \setminus P_{ACL})$
6: $\quad P_X := P_X \setminus t_i.P_t$
7: $\quad$ **goto** 2
8: **end if**

Algorithm 1 encodes the policy decision into an access control token, populating the claims for roles (T_{id}), explicit restrictions (Z_X) and explicit permissions (P_X). The policy decision is represented by the access control list P_{ACL}, and the role-permissions for the resource server is represented by T.

The algorithm Initiate the sets T_{id} and Z_X to empty set ($\emptyset$), and let $P_X := P_{ACL}$. For each iteration, the role $t_i \in T$ which has permissions ($P_t \in t_i$) with the largest overlap with P_X is found. If the permissions of t_i overlaps P_X so that the overall size of data represented in the token decreases, the role identity is added to the list of roles (T_{id}), the permissions granted by t_i is subtracted from P_X, and permissions granted by t_i which are not in P_{ACL} are added to the restriction list Z_X. If no role with big enough overlap has been found, the algorithm is completed.

Policy decision in resource server: When the client c activates a session, the protocol requires using an access token as one of the arguments. Part of the policy decision from the resource server will therefore take place also during *session activation*: Different aspects of the provided token are validated, e.g., the integrity and authenticity of the token, that id_c matches the client id and that id_s matches the resource server identity. The token is persisted together with the session, for use in consequent resource requests. A resource request, which can only be done within the scope of an active session, is defined as an operation on a resource $req = (o, r)$, i.e., on the same form as a permission. The resource server evaluates the request by matching it against the permissions granted by the session token using the following equation[5]:

$$d(req) = \begin{cases} \textbf{deny if } (req \in Z_X) \\ \textbf{allow if } (req \in P_X) \\ \textbf{allow if } (\exists (t \in T) : t.id \in T_{id} \wedge req \in t.P_t) \\ \textbf{deny otherwise} \end{cases} \quad (1)$$

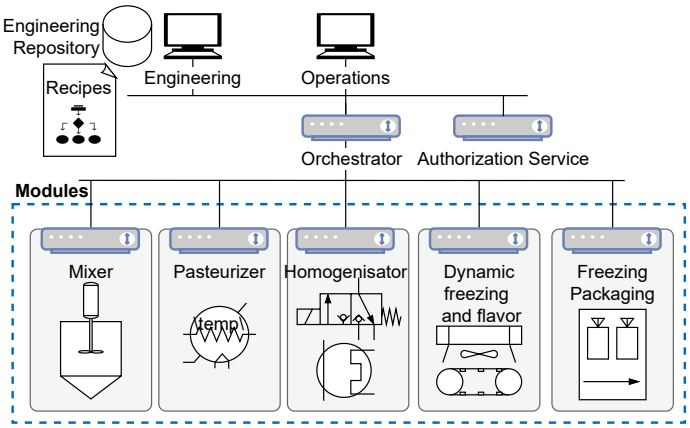


Fig. 4: Use Case: A modular ice-cream factory, architecture overview.

D. Use case

The implementation is done in the context of an ice-cream production process, illustrated in Fig. 4, which is designed

[5]Cases has to be checked in given order.

according to the modular automation system design strategy. Cybersecurity threats against ice-cream factories are limited, and so the target process is merely an illustration. However, the provided implementation is applicable to any manufacturing system using this strategy, e.g., pharmaceutical and chemical processing.

Modular automation uses a service-oriented approach with semi-autonomous modules, specialized in a certain production procedures, having standardized logical and physical interface. During operation, the current production needs decide which modules to use, and how they shall be interconnected. Production synchronization is done using high-level control from a central orchestration unit executing different recipes. This specific realization of the production process comprise five separate modules for mixing, pasteurization, homogenization, freezing and flavoring, and packaging. A more detailed description of the use case system is provided in [38].

All the interactions between entities in the system use OPC UA client/server communication. Issuing of access tokens follows the authorization flow, as defined in the OPC UA standard[6]. The implemented part of the authorization is related to recipe execution, with the orchestrator unit acting as a client, the modules are resource servers, with the high-level control commands as well as operational data as resources. Interactions related to policy decisions and enforcement between an orchestrator, a module, and the AS are illustrated in the sequence diagram in Fig. 5. The dynamic access control is related to which recipes and recipe steps that are active in the system, following the workflow-based approach suggested in [39].

The implementation uses dynamic policy-data in the AS modeled in Next Generation Access Control (NGAC) [40], [41], with software components developed using .NET Core, C#, and using the OPC Foundation .NET OPC UA stack for the communication interactions, the Microsoft.IdentityModel functionality for JWT creation, and the PolicyMachine[7] for NGAC policy model representation.

On a client access token request, the AS queries the NGAC database to compile a set of permissions, P_{ACL}, containing the client permissions relative the resource server resources. The list of permissions is used as input to Algorithm 1, which is executed to populate the access token, which is then returned to the client. The client use the token to activate a session with the resource server, and the resource server use the token to enforce the resource requests from the client, according to the previously described decision function.

A running example of using the token population algorithm and decision function on interactions in this use case is provided in Appendix A.

VI. Verification

In this section we evaluate the behavior of the suggested implementation. This is done by verifying that the set of

[6]IEC 62541 [14] Part 4, chapter 6.2 Authorization Services

[7]https://github.com/PM-Master/policy-machine-core

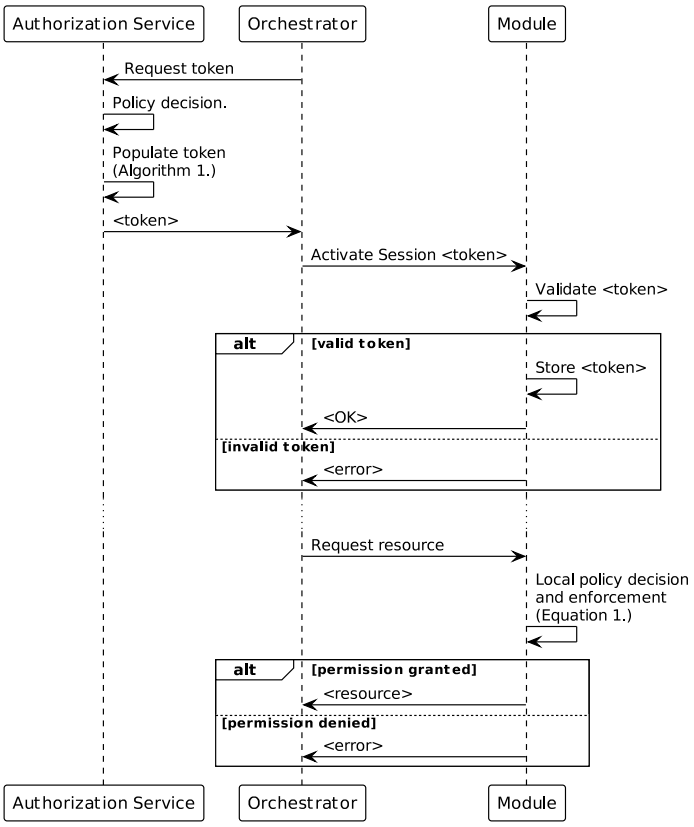


Fig. 5: A sequence diagram for the authorization protocol including policy decisions and enforcement related to session activation and resource requests.

permissions granted by the decision function in Equation 1 matches the access control list P_{ACL}, when the token policy data is generated using Algorithm 1.

Proving the following two theorems is sufficient to show that the decision function d will enforce permissions according to P_{ACL}, since the permissions in P_{ACL} are exclusive, as described in Section V-A:

Theorem 1. *Any request $req = (o, r)$ matching a permission $p \in P_{ACL}$ will result in a decision **allow**.*

Theorem 2. *Any request $req = (o, r)$ not matching a permission $p \in P_{ACL}$ will result in a decision **deny**.*

Proof. of Theorem 1 by contradiction: Let us assume that there exists a $req' \in P_{ACL}$ that would yield decision **deny**. According to the decision function, there are two cases in which this could happen: A) $req' \in Z_X$, or B) if req' is neither part of P_X, nor do any of the roles identified in T_{id} contain req'.

For 1.A) Z_X cannot contain req', as we have stated that $req' \in P_{ACL}$, according to line 5 of Algorithm 1, the elements added to Z_X are always added in the form: $Z_X = Z_X \cup (t_i.P_t \setminus P_{ACL})$.

For 1.B) Neither P_X nor any of the roles in T' contain

req'. Using $t' \in T' | t'.id \in T_{id}$, P_X is initiated to P_{ACL} (line 1 in Algorithm 1), and the only permissions removed from P_X are the ones from roles with identities added to T_{id} (line 6 in Algorithm 1). Therefore req' cannot be in P_{ACL} without being present in either P_X, or one of the roles with identities in T_{id}.

According to 1.A) and 1.B), there cannot be any requests $req' \in P_{ACL}$ resulting in decision function $d(req')$ evaluating to **deny**, consequently any $req' \in P_{ACL}$ yields decision **allow**, which is what we wanted to show. □

Proof. of Theorem 2 by contradiction: Let us assume there exists a $req' \notin P_{ACL}$ which leads to d evaluating to **allow**. This would only be possible if $req' \notin Z_X$ and either A) $req' \in P_X$ or B) $\exists (t \in T) : t.id \in T_{id} \wedge req' \in t.P_t$.

For 2.A) req' cannot be in P_X, as P_X in line 1 of Algorithm 1 is initiated to P_{ACL}, and then only altered by subtraction, in line 6, i.e., $req' \in P_{ACL} \wedge req' \notin P_{ACL} = false$.

For 2.B) Define $(T' | t' \in T', t'.id \in T_{id} \wedge req' \in t'.P_t)$. Then T' are all the roles that contain req'. If $req' \notin P_{ACL}$, then $\forall t' \in T', req' \in (t'.P_t \setminus P_{ACL})$. However, according to the line 5 of Algorithm 1, $Z_X = Z_X \cup (t_i.P_t \setminus P_{ACL})$, which means that all $req' \notin P_{ACL}$ for which there exists a role t' such that $(t' \in T', t'.id \in T_{id} \wedge req' \in t'.P_t)$, i.e., $req' \in Z_X$.

According to 2.A) and 2.B), there cannot be any request $req' \notin P_{ACL}$ resulting in decision function $d(req')$ yielding **allow**, consequently any $req' \notin P_{ACL}$ will evaluate to the decision **deny**, which proves Theorem 2. □

Provided that Algorithm 1, and decision function in Equation 1 are correctly implemented, and the AS has correct information regarding role definitions for the resource server, we have shown that Equation 1, implemented in the resource server, enforces exactly the policy decisions taken by the AS, using the token data produced by Algorithm 1.

VII. Discussion

Balancing usability and cybersecurity is a challenging task as increased cybersecurity may lead to decreased perceived usability. Formulating rules according to the least privilege principle is certainly very difficult and management intensive, and requires some amount of automation to be efficient [20], [42]. Still, we assume that it is possible to have an updated set of dynamic access control policies sufficiently close to this principle in a manufacturing system. The architectures described in this paper is developed to enable enforcement of such dynamic policies.

The implemented architecture provides the possibility of enforcing dynamic access control policies. It is applicable in industrial manufacturing scenarios, with the resource servers being constrained devices. Furthermore, the implementation uses the OPC UA communication standard to allow interoperability with clients following the standard. With this being achieved all our objectives are fulfilled.

Among the four described approaches of using access tokens for dynamic access control, only two of them were possible to implement, considering the OPC UA standard, and only one was deemed useful in scenarios comprising complex resource servers. To include the approaches resulting in the most precise and compact token formulations would require additions to the existing standard, including extending the token request protocol with explicit permission requests. The implemented solution yields correct results but is not guaranteed to produce compact access tokens.

The two suggested architecture models are using session-wide access control tokens, which requires the policy decision point to produce a full set of active permissions/restrictions. This is a limitation, as not all policy models can produce this kind of output, e.g., XACML [21] is optimized for handling atomic access control requests, but cannot effectively provide a list for all permissions that a client holds related to a specific resource [41].

The use of session-wide tokens also limits the dynamicity that the enforcement architecture can exhibit, as the token has a fixed validity time. If the system has a rate of change which is of the same order as the token validity time, there is a high probability that the granted permissions are outdated when used.

In the described architecture models, no functionality related to policy enforcement and decision is placed on the client side. From a cybersecurity perspective, this decision is easily motivated as the resources should not need to be protected by the logic placed in the client, since there may be malicious clients. However, from a usability point of view the client is often relying on knowledge on active permissions to hide or disable parts of the user interface related to disallowed functionality. The OPC UA information model contains metadata on active permissions for an open session, which the client can query from the resource server. This is information that could be used by a client user interface to adapt also to dynamically changing policies.

The provided implementation is done in the scope of a simulated manufacturing system with comparably low complexity, yet it covers all the necessary components of a real system, and is therefore generally applicable for machine-to-machine access control in manufacturing systems.

VIII. Conclusions

Access control is one of several cybersecurity mechanisms highly affected by technical advances and changing business objectives in manufacturing systems. Emerging manufacturing system types are increasingly dynamic, making dynamic access control a requirement in those systems. In this paper, we describe how an enforcement architecture can be modeled and implemented to enable dynamic access control in a manufacturing system, interoperable with the OPC UA standard.

Four architecture models are investigated and evaluated based on three criteria related to minimizing resource server workload and network utility and maximizing access control policy flexibility. Among the evaluated architecture models, the top two are selected for further discussions on policy delegation mechanisms resulting in four different approaches for encoding dynamic policy decisions in access tokens.

An implementation of the most promising combination of architecture and delegation mechanism is described, using the approach of distributed policy decision points, with the dynamic decision taking place in an authorization service, which encodes the policy decisions as access tokens using knowledge on static role-based permissions at the resource server combined with explicit permissions and restrictions.

The implementation includes details on how the authorization service populates access tokens, and how the resource service validates resource requests using the available claims, covering policy decision and enforcement mechanisms. Furthermore, the implementation uses OPC UA as communication protocol, JWT for access token formulations and NGAC for authorization service policy data. Although the implemented solution is simple, it contains essential components of a dynamic manufacturing system, and uses available standardized solutions.

As a future work, we aim to extend and integrate the implementation into a realistic manufacturing environment, based on the modular automation system design strategy. This will provide a better evaluation of the suggested approach with regards to scale and real-time properties.

The two approaches for token content which require explicit permission requests, as described in this paper, are currently not part of any standard, and therefore not part of the provided implementation. They do however hold some interesting properties, which could make it worthwhile to evaluate their usefulness, e.g., for enforcing permissions on safety-critical operations.

ACKNOWLEDGEMENTS

Abhinav Sasikumar and Filip Johnsson at ABB made technical contributions to parts of the use case implementation.

REFERENCES

[1] J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," in *proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[2] A. Sigov, L. Ratkin, L. A. Ivanov, and L. D. Xu, "Emerging enabling technologies for industry 4.0 and beyond," *Information Systems Frontiers*, pp. 1–11, 2022.

[3] K.-d. Thoben, S. Wiesner, and T. Wuest, "Industrie 4.0 and smart manufacturing – a review of research issues and application examples," *Intl. Journal of Automation Technology*, January 2017.

[4] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1 – 10, 2017.

[5] "IEC 62443 security for industrial automation and control systems," standard, Internation Electrotechnical Commission, Geneva, CH, 2009-2018.

[6] E. D. Knapp and J. T. Langill, "Chapter 9 - establishing zones and conduits," in *Industrial Network Security (Second Edition)*, pp. 261–281, Boston: Syngress, second edition ed., 2015.

[7] "O-PAS Standard, Version 2.0: Part 1 – Technical Architecture Overview," Open Group Preliminary Standard (P201-1), The Open Group, February 2020.

[8] ZVEI—German Electrical and Electronic Manufacturers' Association, "Process INDUSTRIE 4.0: the age of modular production," White Paper, Frankfurt, 2019.

[9] J. Ladiges, A. Fay, T. Holm, U. Hempen, L. Urbas, M. Obst, and T. Albers, "Integration of modular process units into process control systems," *IEEE Transactions on Industry Applications*, vol. 54, pp. 1870–1880, March 2018.

[10] T. Seifert, S. Sievers, C. Bramsiepe, and G. Schembecker, "Small scale, modular and continuous: A new approach in plant design," *Chemical Engineering and Processing: Process Intensification*, vol. 52, pp. 140–150, 2012.

[11] K. Knorr, "Dynamic access control through Petri net workflows," *Proceedings - Annual Computer Security Applications Conference, ACSAC*, vol. 2000-January, pp. 159–167, 2000.

[12] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *International Joint Conference on Automated Reasoning*, pp. 632–646, Springer, 2006.

[13] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.

[14] "IEC 62541 OPC unified architecture, rev 1.05," standard, International Electrotechnical Commission, Geneva, CH.

[15] S. Salonikias, A. Gouglidis, I. Mavridis, and D. Gritzalis, "Access control in the industrial internet of things," in *Security and Privacy Trends in the Industrial Internet of Things*, Springer International Publishing, 2019.

[16] B. Leander, A. Čaušević, H. Hansson, and T. Lindström, "Access control for smart manufacturing systems," in *Software Architecture. ECSA 2020. Communications in Computer and Information Science.*, vol. 1269, (Cham), pp. 463–476, Springer, 2020.

[17] E. Yuan and J. Tong, "Attributed based access control (ABAC) for web services," in *IEEE Intl. Conference on Web Services*, vol. 2005, pp. 561–569, 2005.

[18] R. K. Thomas and R. S. Sandhu, "Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management," in *Database Security XI*, pp. 166–181, Springer, 1998.

[19] M. Uddin, S. Islam, and A. Al-Nemrat, "A Dynamic Access Control Model Using Authorising Workflow and Task-Role-Based Access Control," *IEEE Access*, vol. 7, pp. 166676–166689, 2019.

[20] B. Leander, A. Čaušević, H. Hansson, and T. Lindström, "Toward an ideal access control strategy for industry 4.0 manufacturing systems," *IEEE Access*, vol. 9, pp. 114037–114050, 2021.

[21] "eXtensible Access Control Markup Language (XACML) Version 3 . 0 Plus Errata 01," OASIS Standard incorporating Approved Errata., July 2017. Edited by Erik Rissanen.

[22] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," tech. rep., NIST, 2014.

[23] A. Westerinen *et al.*, "Terminology for Policy-Based Management Status." RFC 3198, Nov. 2001.

[24] R. Chandramouli, Z. Butcher, A. Chetal, *et al.*, "Attribute-based access control for microservices-based applications using a service mesh," *NIST Special Publication*, vol. 800, p. 41, 2021.

[25] F. Martinelli, O. Osliak, P. Mori, and A. Saracino, "Improving security in industry 4.0 by extending OPC-UA with usage control," in *15th Intl. Conference on Availability, Reliability and Security*, ACM, 2020.

[26] J. Park and R. Sandhu, "The UCON$_{ABC}$ usage control model," *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 128–174, 2004.

[27] S. Bhatt, T. K. Pham, M. Gupta, J. Benson, J. Park, and R. Sandhu, "Attribute-Based Access Control for AWS Internet of Things and Secure Industries of the Future," *IEEE Access*, vol. 9, pp. 107200–107223, 2021.

[28] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2008.

[29] V. Watson, J. Sassmannshausen, and K. Waedt, "Secure granular interoperability with opc ua," in *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft (Workshop-Beiträge)*, (Bonn), pp. 309–320, Gesellschaft für Informatik e.V., 2019.

[30] D. Hardt, "The OAuth 2.0 Authorization Framework." RFC 6749, October 2012.

[31] R. Baheti and H. Gill, "Cyber-pysical Systems," *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[32] S. V. Sudarsan, O. Schelen, and U. Bodin, "Survey on Delegated and Self-Contained Authorization Techniques in CPS and IoT," *IEEE Access*, vol. 9, pp. 98169–98184, 2021.

[33] S. Dramé-Maigné, M. Laurent, L. Castillo, and H. Ganem, "Centralized, Distributed, and Everything in between: Reviewing Access Control Solutions for the IoT," *ACM Computing Surveys*, vol. 54, no. 7, 2021.

[34] OPC Foundation, "OPC Security Custom Interface." Industry Standard Specification (Version 1.0), Oct. 2000.

[35] R. Rajan, J. Boyle, A. Sastry, R. Cohen, D. Durham, and S. Herzog, "The COPS (Common Open Policy Service) Protocol." RFC 2748, Jan. 2000.
[36] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Signature (JWS)." RFC 7515, May 2015.
[37] M. Jones and J. Hildebrand, "JSON Web Encryption (JWE)." RFC 7516, May 2015.
[38] B. Leander, T. Marković, A. Čaušević, T. Lindström, H. Hansson, and S. Punnekkat, "Simulation environment for modular automation systems," in *IECON 2022–48th Annual Conference of the IEEE Industrial Electronics Society*, pp. 1–6, IEEE, 2022.
[39] B. Leander, A. Causevic, and H. Hansson, "A recipe-based algorithm for access control in modular automation systems," tech. rep., September 2020.
[40] D. Ferraiolo, S. Gavrila, and W. Janse, "Policy Machine: Features, Architecture and Specification," white paper, NIST, October 2015.
[41] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC)," pp. 13–24, 2016.
[42] U. Lang and R. Schreiner, "Proximity-based access control (pbac) using model-driven security," in *ISSE 2015* (H. Reimer, N. Pohlmann, and W. Schneider, eds.), (Wiesbaden), pp. 157–170, Springer Fachmedien Wiesbaden, 2015.

APPENDIX A
RUNNING EXAMPLE OF ALGORITHM AND DECISION FUNCTION

In the following we describe a running example from part of the recipe orchestration for the ice-cream factory use case, highlighting how the access token sets are compiled, using Algorithm 1, and how the decision function $d(req)$ uses the token claims to make local policy decisions in the resource server.

In this example we only look at one module, the mixer module from the use case. The mixer module has resource interfaces according to Fig. 6[8]. Furthermore, there are two roles defined for the mixer module, $T = \{Observer, Operator\}$:

$$Observer = \{CleanupDone.read, EmptyDone.read, FillMixDone.read, Level.read, LevelPercent.read\},$$

and

$$Operator = \{Cleanup, Empty, EmptyAmount, FillAndMix\}.$$

Let us assume a recipe is being executed and the current step of the recipe is related to filling the mixer module. When the orchestrator (being the client in this example) requests an access token from the AS, the following list of permissions could be the result of the policy decision in the AS:

$$P_{ACL} = \{CleanupDone.read, EmptyDone.read, FillMixDone.read, Level.read, FillAndMix\}$$

Now, all the data needed to run Algorithm 1 is available. In line 1, the working sets are initiated:

$$T_{id} = \emptyset, Z_X = \emptyset, P_X = P_{ACL}$$

[8]Screen capture from the Unified Automation UaExpert tool (www.unified-automation.com).

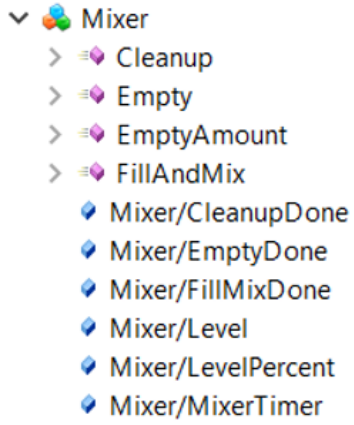


Fig. 6: Interfaces for Mixer module.

At line 2, l_i is evaluated for all the roles, with $l_{Observer}$ being the smallest.

$$l_{Observer} = \mathbf{max}(1,1), l_{Operator} = \mathbf{max}(4,4)$$

At line 3, $l_{Observer}$ is compared with the cardinality of P_X, which is 5, greater than 1. At line 4, 5 and 6 the working sets are updated so that:

$$T_{id} = \{Observer\}, Z_X = \{LevelPercent\}, P_X = \{FillAndMix\} \quad (2)$$

Line 7 directs execution back to line 2, which evaluates smallest l_i over the changed working sets:

$$l_{Observer} = \mathbf{max}(1,5), l_{Operator} = \mathbf{max}(4,4)$$

Smallest value is $l_{Operator} = 4$ which is greater than $\mid P_X \mid$, and therefore the algorithm ends here. The resulting sets from Equation 2 are used to populate the claims in the access token which is returned to the orchestrator as follows.

```
{
  "sub": "Orchestrator_X",
  "aud": "MixerModule",
  "name": "Ice Cream Factory Orchestrator X"
  "iat": <issuing time>,
  "exp": <expiry time>,
  "roles": ["Observer"],
  "entitlements": ["FillAndMix"],
  "restrictions": ["LevelPercent.read"]
}
```

The orchestrator uses the token to activate a session toward the Mixer module. Using the sets in the token, the decision function $d(req)$, running at the mixer module, can be evaluated. Assume e.g., that the orchestrator attempts a request: $req' = Level$, then $d(req') = \mathbf{allow}$, as $Level \in t_{Observer}.P_t$. For $req' = Cleanup$, $d(req') = \mathbf{deny}$, as $Cleanup \notin P_X, Cleanup \notin t_{Observer}.P_t$, etc.