

## Research article

Exploiting usage control for implementation and enforcement of security by contract<sup>☆</sup>Marco Rasori<sup>a</sup> , Paolo Mori<sup>a</sup> , Andrea Saracino<sup>a,b</sup> , Alessandro Aldini<sup>c</sup> <sup>a</sup> Institute of Informatics and Telematics, National Research Council of Italy, Via G. Moruzzi 1, Pisa, 56124, Italy<sup>b</sup> Department of Excellence in Robotics, AI, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, 56127, Pisa, Italy<sup>c</sup> Università degli Studi di Urbino Carlo Bo, Via Aurelio Saffi 2, Urbino, 61029, Italy

## ARTICLE INFO

## Keywords:

Internet of Things  
Usage control  
Security by contract  
Access control  
Model checking

## ABSTRACT

The widespread adoption of IoT-based smart home technologies has transformed how people interact with their living spaces, offering greater control over everyday tasks. However, this increased connectivity introduces significant security challenges, particularly in managing applications that can control devices within the smart home. Users need effective ways to define and enforce security policies that permit or deny specific behaviors of these applications. Such policies should allow users to control what actions applications can perform, ensuring that they comply with security and privacy preferences. This paper proposes a hybrid framework that combines Security by Contract (SxC) and Usage Control (UCON) to address these challenges and provide a comprehensive security solution with low impact on system performance. SxC ensures verification of the application behavior, described formally as a contract, against predefined XACML-based policies. UCON enables continuous monitoring and enforcement of security policies during application execution. The theoretical foundations of the methodology combining these frameworks are based on labeled state/transition systems and their model-checking-based verification. Through experimental validation on a real testbed, we explore the feasibility of the proposed approach by evaluating its performance across various test campaigns, offering insights into its ability to manage policy enforcement and revocation processes with low overhead.

## 1. Introduction

The rapid adoption of smart home technologies has transformed modern living spaces into interconnected ecosystems, offering unprecedented convenience, efficiency, control, and energy-saving capabilities. Smart devices, such as thermostats, security cameras, lights, and appliances can be controlled remotely via mobile applications, supported by frameworks like Home Assistant, OpenHAB, Web of Things, and Matter, creating a seamless and integrated home environment. However, this proliferation of smart home applications also introduces significant security challenges. The complexity of managing numerous distinct applications, each

<sup>☆</sup> This work was partially funded by the European Union – NextGenerationEU within the framework of PNRR Mission 4 – Component 2 – Investment 1.1 under the Italian Ministry of University and Research (MUR) programme “PRIN 2022” – AsCoT-SCE (Assessing Compliance of IoT API for Security Critical Environments) – grant number 2022598LMZ – CUP: H53D23003430006, and the Horizon Europe project MEDIANE (Multi faceted Implementation of a mixed software/hardware-based zero-trust framework for the computing continuum), grant agreement 101168465.

\* Corresponding author.

E-mail addresses: [marco.rasori@iit.cnr.it](mailto:marco.rasori@iit.cnr.it) (M. Rasori), [paolo.mori@iit.cnr.it](mailto:paolo.mori@iit.cnr.it) (P. Mori), [andrea.saracino@santannapisa.it](mailto:andrea.saracino@santannapisa.it) (A. Saracino), [alessandro.alдини@uniurb.it](mailto:alessandro.alдини@uniurb.it) (A. Aldini).

<https://doi.org/10.1016/j.iot.2025.101697>

Received 20 February 2025; Received in revised form 12 June 2025; Accepted 25 June 2025

Available online 16 July 2025

2542-6605/© 2025 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

interacting with a large set of devices and accompanied by its own set of permissions and potential vulnerabilities, poses a substantial risk to user privacy, user safety, and overall data security.

In the smart home context, ensuring the secure installation and execution of applications is paramount. Each application must be thoroughly examined to ensure it complies with user-defined security policies before it can be integrated into the smart home ecosystem. Additionally, during execution, all invoked functionalities must adhere to these security policies. This is crucial to prevent unwanted behavior, unauthorized access and usage of devices, and other cyber threats that could jeopardize the safety, privacy, and overall security of the smart home environment.

Controlling whether smart home applications have the appropriate access rights to operate on devices, features, and functionalities is crucial for mitigating security and safety risks. Traditional security measures, which often rely on static permissions and infrequent security checks, are insufficient in the highly dynamic and interconnected world of smart homes. Instead, a more flexible and comprehensive approach is needed, one that continuously monitors and controls the behavior of applications to ensure they adhere to the defined security policies throughout their lifecycle. Thus, security in smart homes involves establishing robust frameworks that can dynamically assess and enforce security policies. The required flexibility, policy expressiveness, and effectiveness of enforcement can be achieved using the Attribute Based Access Control (ABAC) paradigm. ABAC leverages standardized architectures, policy languages, and well-established decision and enforcement frameworks, making it highly versatile and easily integrable virtually in any environment. Still, policy evaluation performed at runtime imposes a significant performance overhead, particularly when multiple requests must be handled in parallel, such as in the heterogeneous and multi-device settings of a smart home.

This paper explores the combined usage of Security by Contract (S×C) and Usage Control (UCON) paradigms to control the access rights of smart home applications. Security by Contract is a model that assesses an application's compliance with predefined security policies by leveraging formal contracts. These contracts specify the security-relevant behaviors an application may exhibit during execution. The S×C model relies on three cornerstone elements: the application code ( $\mathcal{A}$ ), the contract ( $\mathcal{C}$ ), and the policy ( $\mathcal{P}$ ). The primary objective is to verify that the application code adheres to the contract (formally expressed as  $\mathcal{A} \leq \mathcal{C}$ ), and that the contract complies with the policy (formally expressed as  $\mathcal{C} \leq \mathcal{P}$ ), thereby ensuring that the application behaves securely according to the policy (namely,  $\mathcal{A} \leq \mathcal{C} \leq \mathcal{P}$ ).

On the other hand, the Usage Control model extends traditional access control models by incorporating continuous monitoring and control of resource usage. Unlike conventional models that make a single decision when access to a resource is initially requested, UCON continuously enforces policies throughout the entire duration of the resource's use, thereby considering dynamic changes in context and ensuring ongoing compliance with security requirements. This is particularly relevant in IoT environments where conditions and contexts can change rapidly, necessitating real-time policy evaluation and re-evaluation.

### 1.1. Contribution

The contribution of this paper is twofold. First, it formalizes the relationship between the two paradigms, S×C and UCON, by demonstrating that the matching operation between a UCON policy and a UCON request is equivalent to a model-checking operation expressed in the same logic used for the S×C's contract-policy matching operation. Second, it proposes an integrated framework that harmonizes S×C and UCON to guarantee enhanced yet efficient security in smart home systems. In particular, this framework enhances security of the smart home by ensuring that applications are installed and executed only if they fully comply with user-defined UCON policies. In addition, leveraging the S×C paradigm, the framework reduces the overhead imposed by UCON at execution time by enforcing execution policies only on those applications which have the potential to violate them. Indeed, when the contract of an application is fully compliant with the UCON policy at installation time, the application is executed without further checks, removing any access control overhead at runtime. Instead, if an application's contract only partially complies with the UCON policy at installation time, the framework provides the users with the option to install it with certain functionalities monitored at runtime or to forgo the installation entirely. Specifically, the application's functionalities that have been found compliant directly at installation time will be executed at runtime without any further check. Instead, for functionalities found non-compliant during installation, our framework enables corresponding monitors, which are then used during execution to check whether these functionalities comply with the security policies at runtime, based on both static and dynamic (context-dependent) conditions. To this end, we leverage a set of controlled, secure, and reliable APIs for interacting with smart home devices, along with a trustworthy software development kit. This kit enables untrusted third-party developers to write applications that can interact with and control smart home devices and services through the provided APIs. The behavior of these applications is defined by S×C contracts, which are automatically generated by the trustworthy software development kit and directly derived from the set of secure APIs used within the application's source code. On the other hand, either the user or a security home administrator will define a set of Usage Control policies. These policies regulate application access rights, including the right to install applications and the actions they can or cannot perform once installed and active. Building on the integration of the S×C and UCON paradigms, we exploit the UCON paradigm to verify these policies against the application contracts at installation time, and to perform runtime monitoring on partially compliant applications.

Through the proposed framework, we define a security model which is generalizable to any IoT framework involving third-party services or applications, where the service providers, i.e., the manager of the IoT frameworks, have control on the application development APIs. Then, we demonstrate how integrating two existing paradigms can ensure a fine-grained and flexible workflow for controlling applications behavior, effectively addressing security and safety issues in an efficient manner.

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

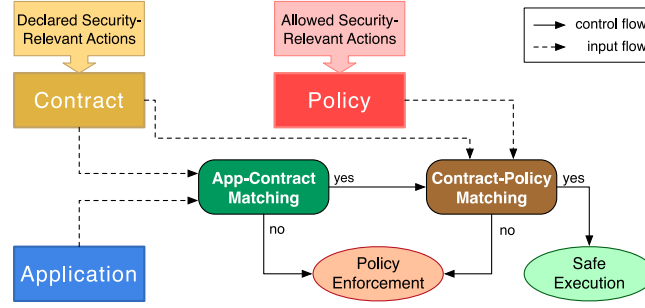


Fig. 1. Security-by-Contract components and workflow.

- We introduce an on-system framework to control the behavior of third-party IoT applications running on that system, ensuring they abide to specific user-defined policies regulating the usage of resources and possible operations.
- We propose for the first time the integration of Security by Contract and Usage Control paradigms to ensure comprehensive security in smart home ecosystems. This integration combines S×C's formal verification of application behavior with Usage Control's continuous monitoring capabilities providing a Usage Control-based implementation of the S×C elements.
- We formalize the representation of application contracts and policies using labeled state/transition systems, enabling precise verification and ensuring that application behavior aligns with security policies.
- We propose a mechanism to reduce runtime overhead, caused by policy enforcement by leveraging the S×C paradigm, to ensure that runtime enforcement is only applied when compliance checks at installation time identify a potential policy violation.
- Through experimental validation on a realistic testbed, we evaluate the performance of the proposed approach across various test campaigns, offering insights into its ability to manage policy enforcement and revocation processes.

## 1.2. Paper structure

The rest of the paper is organized as follows. Section 2 reports background concepts on Security by Contract and on Usage Control. Section 3 describes the reference scenario and system architecture. Section 4 outlines the proposed methodology. Section 5 presents models and verification techniques. Implementation details and experimental evaluation are discussed in Section 6 and Section 7, respectively. Section 8 discusses related work. Finally, Section 9 draws our conclusive remarks.

## 2. Background

This section introduces the background and the concepts used in the rest of the paper, namely the Security by Contract model and the Usage Control framework.

### 2.1. Security by Contract (S×C)

The Security by Contract (S×C) model [1] is based on three cornerstone elements: the app *code*  $\mathcal{A}$ , the app *contract*  $\mathcal{C}$ , and the client *policy*  $\mathcal{P}$ . Given an application, its *contract* is a formal specification of the security-relevant behavior that the application can exhibit during execution on a particular device. Such a behavior is related, e.g., to the API calls or the critical system calls performed by the app. A *policy* is a formal specification of the acceptable security-relevant behavior that the app is allowed to execute on the device in which it is installed. The basic idea of the contract-based approach consists in employing the contract to verify that the security conditions imposed by the policy are actually satisfied by the app. To formalize this, let  $\leq$  denote the *compliance relation* between any pair of elements in the S×C model. The contract-based approach relies on the satisfaction of the following transitive relationship:

$$\mathcal{A} \leq \mathcal{C} \leq \mathcal{P} \Rightarrow \mathcal{A} \leq \mathcal{P}. \quad (1)$$

Fig. 1 reports the S×C components and workflow, which can be described as follows. As an initial step, the verification process assesses whether the contract matches the application, determining if the contract accurately represents the behavior of the app ( $\mathcal{A} \leq \mathcal{C}$ ). This operation is named *app-contract matching* and can be based on several methodologies, depending on the adopted contract model, and spanning from proof-carrying code [2] to the application of trust relations towards either the application developer or the certification authority issuing the contract. Assumed that the application and the contract match, as a second step, the S×C flow verifies if the contract and the security policy defined by the system administrator match as well ( $\mathcal{C} \leq \mathcal{P}$ ). If both the contract and policy are formally expressed using compatible models, this operation, known as *contract-policy matching*, can employ automated formal verification techniques, such as model checking [3] to be carried out.

If both the app-contract matching and the contract-policy matching are verified, the app can be safely executed. This is because, based on the previous checks, it is demonstrated that the app behavior complies with the policy ( $\mathcal{A} \leq \mathcal{P}$ ).

Conversely, if either the app-contract matching or the contract-policy matching fails, the desired relation in Eq. (1) cannot be verified. Nonetheless, the app could still be executed. A *monitor* is attached to the app to ensure that its execution adheres step-by-step to the specified policy. Specifically, if the app attempts to execute an action that violates a policy, the monitor *enforces* the policy by halting the execution of that action. Note that this enforcement introduces overhead, which is entirely avoided in the case of safe execution.

## 2.2. The usage control framework

The *Usage Control (UCON) framework* is based on the UCON model [4] and extends the XACML reference architecture and language [5] to regulate the exercise of rights on resources by subjects, following the principles of Attribute-Based Access Control (ABAC). XACML provides a specification for implementing ABAC frameworks, where access decisions are made based on the attributes of subjects and resources. In the context of a smart home, relevant attributes might include the physical location of a subject or the temperature of a resource, such as a smart oven. In addition, UCON extends traditional access control models by introducing the concept of *attribute mutability* and ensuring the *continuity of policy enforcement*. However, it also considers two critical factors that are relevant in ABAC: *environmental conditions* and *obligations*. Environmental conditions refer to factors not directly tied to the subject or resource, such as date and time, outside temperature, or weather forecasts, which can dynamically change during access, influencing both access and usage decisions in real time. On the other hand, obligations impose mandatory requirements on the subject, such as signing an agreement, paying a fee, or uploading a file, before or during access.

Unlike traditional access control systems which check permissions only at request time, the UCON framework continuously verifies that access rights hold during the whole duration of an access to a resource (*usage*). Access grants may be revoked if the value of some subject's, resource's, or environmental attributes change. Attributes whose value can change over time are called *mutable attributes*. As an example, consider a scenario where 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 only checks conditions at the time of request, allowing the operation if the room temperature is within range. In contrast, UCON not only performs this initial check but also continuously monitors the condition throughout the access period, revoking the grant if the temperature goes out of range.

*Usage Control Policies (UCPs)* define the access strategies using the U-XACML language [6], an extension of XACML that introduces time-based conditions. In particular, to implement continuous policy enforcement, each rule in a U-XACML policy includes three types of conditions and obligations: *pre-*, *ongoing-*, and *post-*. *Pre-conditions* and *pre-obligations* are enforced at access request time, determining whether the access should be granted. *Ongoing-conditions* and *ongoing-obligations* are enforced while the access is in progress, ensuring that the access rights remain valid, or deciding if the access must be interrupted, i.e., revoked. *Post-obligations* are enforced after the access has concluded, requiring the execution of specific tasks once access has ended.

A UCP consists of multiple rules, each returning one of four possible outcomes: Permit, Deny, NotApplicable, or Indeterminate. The final authorization decision is determined by properly combining the outcomes of the policy rules according to a *rule-combining algorithm*. For instance, the *permit-overrides* rule-combining algorithm returns permit if the evaluation of at least one of the rules of the policy returned permit. Additionally, when multiple policies are grouped into a *policy set*, a *policy-combining algorithm* is used to determine a final authorization decision.

Defining access and usage control policies is a complex task, particularly for users without technical expertise. Although several tools have been developed to assist in generating access control policies using the XACML language—such as the AI-based pipeline by Paratore et al. [7], the policy view method by Lang et al. [8], the Islandora XACML Editor [9], the UMU-XACML-Editor [10], and the Axiomatics ALFA Eclipse plugin [11]—no such tools currently exist for the U-XACML language. Nevertheless, with suitable adaptation, existing XACML tools could potentially be extended to support U-XACML policy creation.

### 2.2.1. The UCON workflow

The UCON workflow begins when a subject  $s$  requests access to perform an action  $a$  on a resource  $r$ . The *Policy Enforcement Point (PEP)* initiates this process by issuing a *tryAccess* message, which includes a *UCON request*, encoded in XACML. A typical UCON request contains the attributes *subject-id*,<sup>1</sup> which identifies the subject, the attribute *resource-id*, which identifies the resource, and the attribute *action-id*, which specifies the action that such a subject wants to perform on the resource. This request is purely in XACML format since it does not require any UCON-specific features.

The *tryAccess* message reaches the *Usage Control System (UCS)*, and the UCON request is dispatched to the *Context Handler (CH)* for evaluation. Fig. 2 illustrates the UCON framework architecture, highlighting its main components and their interactions. Upon receiving the UCON request, the CH retrieves relevant attributes by querying *Policy Information Points (PIPs)*, which are configured to interact with the appropriate attribute providers, called *Attribute Managers (AMs)*. For instance, in the smart home scenario, the Wi-Fi access point can be queried to determine whether the (device of the) subject is connected to the Wi-Fi network, thus indicating that the subject is at or near home. Similarly, the current temperature of a smart oven can be retrieved by contacting the oven itself.

Once the required attributes are obtained, the CH constructs an *enriched UCON request* by adding these new attributes to the original request, and forwards it to the *Policy Decision Point (PDP)* to identify an applicable UCP. The PDP then evaluates the

<sup>1</sup> For readability, in the text we use a short version of the actual XACML attributes' names in a way to avoid ambiguity. Table A.2 in Appendix A reports the name used within the text (short name) and the related full name, or XACML AttributeId.

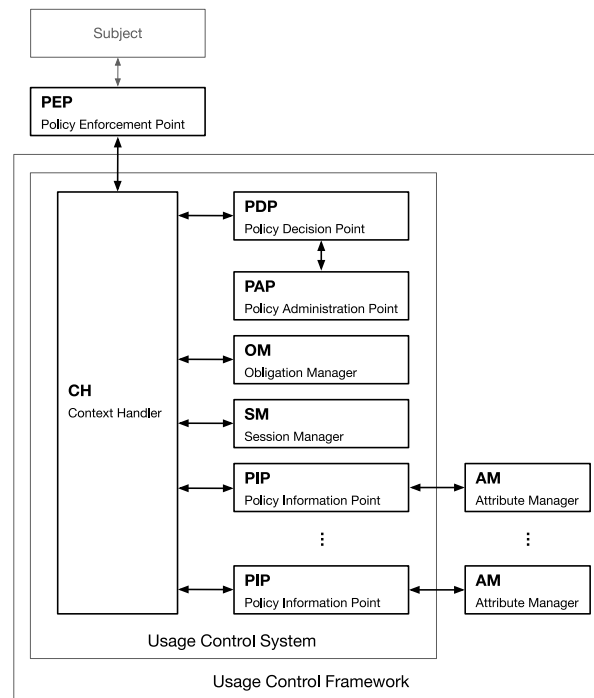


Fig. 2. UCON framework architecture.

*pre-condition* of the UCP's pre-section. Notably, the PDP can use a standard XACML engine, such as WSO2 Balana,<sup>2</sup> for evaluation, since only one condition (pre-, ongoing-, or post-) is evaluated at a time. If the result of the evaluation is Deny, the UCS responds to the PEP with a *denyAccess* message, which enforces the decision by denying access to the resource. If Permit is returned, the CH creates a *session* at the *Session Manager* (SM), thus storing relevant information such as the session identifier, the original request, and policy used. Next, the UCS sends a *permitAccess* message to the PEP, which then grants access to the resource.

Once access begins, the PEP sends a *startAccess* message to the UCS. The CH retrieves the session details, re-enriches the request with current attribute values, and evaluates the *ongoing-condition* of the UCP's ongoing-section. If the result is Deny, access is immediately revoked via a *revokeAccess* message. If Permit, the session status is updated, and the CH subscribes to relevant PIPs to monitor mutable attributes, ensuring continuous evaluation. Moreover, the UCS sends a *permitAccess* message to the PEP to notify it that the access can continue.

If an attribute involved in the ongoing-condition changes, the CH is notified by a PIP, and this event triggers a *policy re-evaluation*. This means that the PDP reassesses the enriched request and ongoing-condition. If the ongoing-decision is Deny, access is revoked, and the PEP is notified through a *revokeAccess* message. If Permit, access continues uninterrupted.

Finally, after access to the resource is terminated, the PEP sends an *endAccess* message to the UCS, which triggers the evaluation of the *post-condition* in the post-section, and the session is deleted from the SM, concluding the workflow.

At any stage of the workflow, the PDP includes any associated obligation (pre-, ongoing-, or post-) in its response to the CH.

### 3. Reference architecture and application scenario

Smart homes can be considered critical assets due to their integration of IoT devices, which, while offering numerous benefits, also present significant risks. These risks include high energy consumption costs, potential damage to the property, physical harm to occupants and their possessions, and possible harm to neighboring properties.

For example, a smart window or roller shutter might open during a storm, leading to rain infiltration and potential water damage within the smart home. Additionally, if the cooling system operates when no one is home, or if it runs simultaneously with open windows or an active heating system, it can lead to unnecessarily high energy costs. Furthermore, the simultaneous activation of multiple high-energy consumption devices (e.g., smart washing machines, ovens, and hairdryers) can result in excessive power usage, potentially triggering a disconnection of the smart home from the electrical grid. Given these concerns, regulating the use of IoT devices in a smart home is crucial for minimizing energy consumption, enhancing security, protecting privacy, and ensuring overall safety.

<sup>2</sup> <https://github.com/wso2/balana>

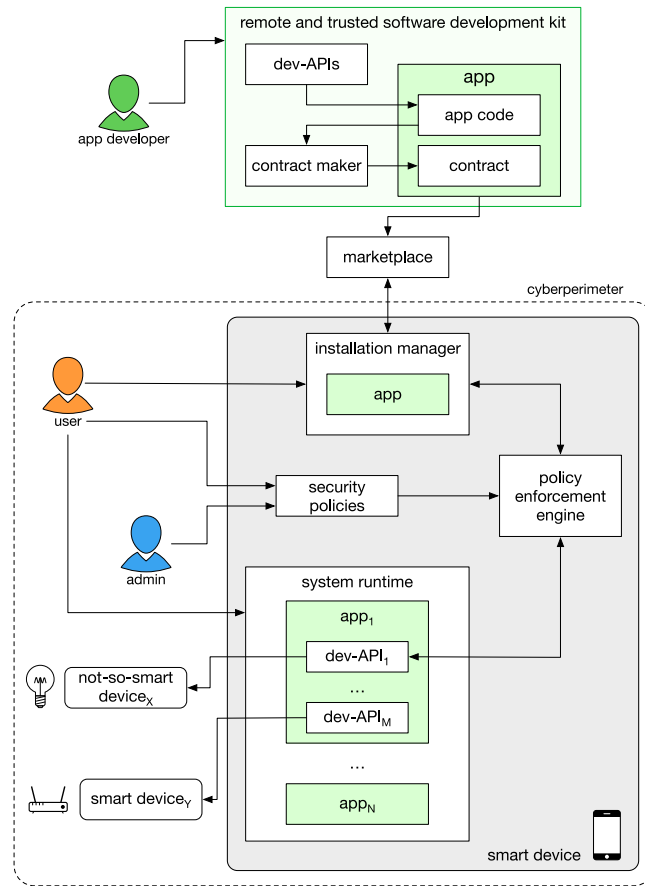


Fig. 3. System Architecture.

The UCON framework (described in Section 2.2) is highly suitable for controlling access to smart home devices. It enables precise control over the right to perform tasks on a device, both at the time of the request and during the device's operation, until the task is completed. This means that an ongoing task can be suspended or even halted if the usage control policy, initially satisfied, is later violated.

The rest of this section describes the reference architecture and provides examples of policies that could be used in a smart home, while the following sections illustrate how the UCON model is employed to safeguard smart homes while minimizing related computational overhead.

### 3.1. System architecture

Fig. 3 illustrates the actors and components constituting our reference architecture. The *cyberperimeter* (depicted as a dashed line) delineates the smart home premises, distinguishing between activities occurring outside and inside the smart home. On the outer side, third-party *app developers* utilize a software development kit, provided as a remote service, to deploy applications (*apps*). This service includes an integrated development environment (IDE) and a build system running on trusted remote servers, ensuring that the app development is securely handled. An app consists of the *app code*, implementing the functionalities of the app, and a *contract*, i.e., a manifest specifying the security-relevant operations the app intends to perform. The contract is derived from the app code using a tool called *contract maker*. Having the development, the compilation process, and the contract generation in such a controlled and trusted environment ensures the contract's correctness and guarantees that it cannot be counterfeited.

In this work, we assume that we have control over the smart home framework and the methods available to app developers and used to write the apps, referred to as *dev-APIs*. In particular, we assume that we can modify the implementation of the dev-APIs, as explained later in this section. Open-source smart home frameworks, like Web of Things and SIFIS-Home, offer this flexibility, making them suitable frameworks to be adopted. Furthermore, we assume that an app can interact with IoT devices in the smart home solely by invoking these methods.

Each dev-API is a specialized, trusted function accomplishing a specific functionality on a particular type of IoT device. A dev-API is characterized by (i) the *action* ("dev-API action") to perform, (ii) the *type* ("dev-API type") of the resource the action will be

performed on, and, optionally, (iii) one or more parameters (“dev-API parameter  $x$ ”) to specialize the behavior, where  $x$  indicates the position of the parameter in the parameter list. Let us suppose to have the SIFIS-Home dev-API `set_lamp_brightness(string id, uint value)`, which involves the action “set\_lamp\_brightness” and targets a resource of type “lamp”. It includes a parameter, `id`, that specify the identifier of the lamp resource to operate on, and an additional parameter, `value`, which is an unsigned integer within a certain range representing the brightness level. Based on the dev-API structure, the contract maker creates the contract by extracting these pieces of information from the invocations of the dev-APIs within the app code.

The functionality of a dev-API can either require exclusive access to the resource, or allow shared access where multiple apps or processes can access the resource simultaneously. Functionalities that require exclusive access modify the state of the resource and are typically classified as “set” operations. In contrast, functionalities that allow shared access are designed to read the state of the resource without altering it. Throughout this paper, we refer to *set* operations as those that necessitate exclusive access to the resource, and *get* operations as those that do not require exclusive access. In the earlier example, the “set\_lamp\_brightness” action is of type *set*, as it modifies the brightness level of the lamp.

When an app developer releases an app, the app is uploaded to an app repository, namely, the *marketplace*. A *user* utilizes a specialized app on their device, the *installation manager*, to search, download, and install apps from the marketplace that best suit their needs for controlling their *IoT devices*. This paradigm is common in the smart home ecosystem, exemplified by frameworks like IFTTT, Web of Things, and SIFIS-Home.

Both users with administrative privileges and external administrators are referred to as *policy makers*, and they have the authority to define security policies concerning the smart home and the use of IoT devices. For instance, policy makers can implement security policies that restrict functionalities for specific apps or prevent the installation of apps that exhibit undesired behavior.

Running apps operate within the *system runtime* component, which manages the apps’ lifecycle and runs on a smart device in the smart home, as shown in Fig. 3. Our scenario is inspired by the one proposed in the SIFIS-Home project (described in [12]), which defines two classes of IoT devices: *Smart Devices* and *Not-So-Smart Devices*. Smart devices are characterized by medium to high computational capabilities, support for one or more connectivity interfaces, and the ability to be customized through the installation of third-party applications. Representative examples of smart devices are Raspberry Pi boards, Android or iOS devices, including routers, smartphones, tablets, general-purpose embedded systems, and even desktop or laptop computers. Not-So-Smart Devices (NSSDs) are typically low-power devices equipped with one or more connectivity interfaces. They are primarily employed to sense environmental parameters (i.e., sensors) or to interact with the physical environment (i.e., actuators). Differently from smart devices, NSSDs lack the ability to be customized. They usually run simplified operative systems—often closed or with limited configurability—and do not support the installation of third-party applications. Instead, NSSDs are configured to communicate with a designated smart device, which is responsible for issuing commands or retrieving data from them. Within the execution flow of a running app, when a dev-API is invoked, a request is generated and sent to the *policy enforcement engine* to request access permission for performing a functionality. In our architecture, the UCON framework described in Section 2.2 serves as the policy enforcement engine, and it is deployed on smart devices as well.

The policy enforcement is facilitated by integrating a PEP within the dev-API code (*inlining*), enabling runtime requests to the UCS (see Section 3.2 for further details). If the UCS denies access, the rest of the dev-API code is skipped, preventing execution of the functionality. Conversely, if the UCS permits access, the dev-API’s functionality is executed as intended.

### 3.2. Security policies and reference examples

In our methodology, security policies are logically divided into *installation policies* and *execution policies*. In the following, we explain both types of policies and provide reference examples that will be used throughout the paper.

#### 3.2.1. Installation policies

Installation policies are specifically designed to enforce constraints during the app installation process. They define the conditions under which an app can be installed and, according to the UCON model, the criteria for an app to remain installed. If an installation policy is not satisfied, the app should not be installed because its code violates some constraint declared in the policy. The constraints are directly tied to the dev-APIs invoked in the app code and should not take into account the value of those attributes that describe the access context and are meaningful only at runtime, when the dev-API is actually invoked. Examples of such attributes are the current state of a window (e.g., open or closed) or the instantaneous energy consumption of the smart home. However, policy makers can define installation policies that include mutable attributes, provided they are relevant at installation time.

**Reference example 1: “allow-fast-charge-if-power-above-threshold”.** This installation policy is defined for apps that operate on resources of type “charger”, such as Level 2 electric vehicle chargers, which are power-hungry devices. The policy reads: “Allow installation of an app that uses the dev-API `fast_charge(string id)` and maintain the app installed as long as the maximum power capacity of the electricity supply contract is above 5 kW”. This policy consists of a single rule, in which both the pre-condition and the ongoing-condition include a predicate requiring the *maximum power capacity* attribute to not exceed the threshold value of 5 kW. To illustrate this policy, consider an app that uses the `fast_charge(string id)` dev-API. The app is initially installed since the electricity supply contract specifies a maximum power capacity of 6 kW, meeting the policy’s threshold. Later, the maximum power capacity is downgraded to 3 kW, which falls below the required threshold. As a result, the app is automatically uninstalled to ensure that the policy’s constraints are enforced. This mechanism guarantees that apps remain compliant with the available energy capacity, preventing potential power management issues and maintaining system reliability.



*Reference example 2: “allow-low-brightness-lights-apps”.* Consider an installation policy that reads: “Allow installation of apps that set the lamp brightness to a value no greater than 50, ensuring that lights are not too bright”. If the user tries to install an app such as *smartLightingControl*, and the app’s code contains the SIFIS-Home dev-API `set_lamp_brightness(lamp_ref, value)`, and the app sets the brightness to a value of 50 or below, the installation policy would be satisfied, and the app could be installed, as it adheres to the brightness limit. Conversely, if the app code contains the `set_lamp_brightness(lamp_ref, value)` dev-API and sets the brightness to a value higher than 50 (e.g., 75), or if it is not clear from the app code what value will be set (for instance, if the brightness is determined dynamically by user input or an external factor), the policy would not be satisfied. In such cases, the app should not be installed, as it either exceeds the maximum allowed brightness or the value cannot be determined, which leaves uncertainty about whether the brightness might exceed the allowed limit.

### 3.2.2. Execution policies

Execution policies, on the other hand, are designed to permit or deny the execution of dev-APIs at runtime, especially when policies conditions depend on contextual information, such as mutable attributes.

*Reference example 3: “restrict-loud-volume-at-night”.* Consider the dev-API `play_audio(string id, string track, uint volume)`, which plays an audio track on a smart speaker (identified by `id`) at the desired volume level. If the policy maker defines an execution policy stating “audio playback is always allowed, but volume must not exceed 50% between 11 PM and 8 AM”, the constraint of this policy will be enforced during runtime. The current time and volume level are mutable attributes, meaning their values change over time and can only be known and evaluated when the dev-API is invoked. If the time is between 8 AM and 11 PM, audio playback is fully permitted at any volume level. However, if the time is between 11 PM and 8 AM, any attempt to set the volume above 50% will be denied. Additionally, if audio is already playing when 11 PM arrives and the volume level is above 50%, the system will revoke access and immediately stop playback to enforce compliance with the policy.

*Reference example 4: “forbid-ac-if-any-window-open”.* Consider a different scenario in which the app *smartHVAC*, which optimizes indoor temperature control by acting on the Heating, Ventilation, and Air Conditioning (HVAC) system based on user preferences and environmental conditions, has been installed. In this case, we define an execution policy with one rule which states “the air conditioning system can be activated and remain operational only as long as all the windows are closed”. When the `turn_HVAC_on(string id)` dev-API is invoked by the *smartHVAC* app, the inlined PEP sends a request to the UCS, which retrieves the current status from the window sensors to evaluate whether the air conditioning can be activated. If the policy evaluation returns a Deny decision—because one or more windows are open—the rest of the dev-API’s code is skipped, so the functionality is prevented from being executed in accordance with the execution policy. Conversely, if a Permit decision is initially returned, the air conditioning is activated. Subsequently, the UCS continuously monitors the attributes representing the current status of the windows, retrieved from the related sensors. If a window is then opened, the policy—specifically, its ongoing-condition—is re-evaluated. In this case, the re-evaluation results in a revocation decision, leading to the automatic deactivation of the air conditioning system.

*Reference example 5: “allow-economy-or-night-wash”.* Consider the dev-API `washing_machine(string id, string washing_program)`, which enables an app to start a wash cycle on a smart washing machine identified by `id` using the specified washing program. Suppose an execution policy with two rules is in place. The policy states: “The washing machine may be used if (rule i) the selected program is economic OR (rule ii) the cycle is initiated between 8 PM and 6 AM”. This policy enforces restrictions on when and how the washing machine can be operated, based on the time of day and the type of washing program. When an app attempts to invoke the `washing_machine` dev-API, the inlined PEP checks the current time and the specified `washing_program` parameter. The current time is fetched at runtime, while the program name is extracted from the API call. If the selected washing program is economic, access is always granted regardless of the time. Similarly, if the time falls between 8 PM and 6 AM, access is permitted for any washing program. However, if the program is not economic and the request is made outside the allowed nighttime window (i.e., between 6 AM and 8 PM), the UCS returns a Deny decision, and the API invocation is halted.

Beyond technical policy enforcement, our approach applies to diverse real-world smart home scenarios where these policies offer significant benefits. For example, in energy-aware households, execution policies can limit appliance usage to off-peak hours or enforce efficient modes. In multi-user environments, such as families with children, role-based policies can control which users may install or operate certain apps. In homes equipped with smart HVAC systems, context-aware policies prevent wasteful operation (e.g., disabling HVAC when windows are open). For short-term rentals, owners can enforce strict installation policies to prevent guests from altering system behavior. Similarly, in assisted living contexts, policies can ensure critical automation (e.g., motion-triggered lighting) remains active and compliant with safety requirements. Together, these scenarios show how our framework can support a broad spectrum of needs in modern smart homes.

While this work focuses on a smart home reference scenario, we point out that this technology is generic enough to be easily tailored and applied to other domains that follow a similar application and services deployment models (see Fig. 3), such as smart manufacturing, smart vehicles, and smart cities.



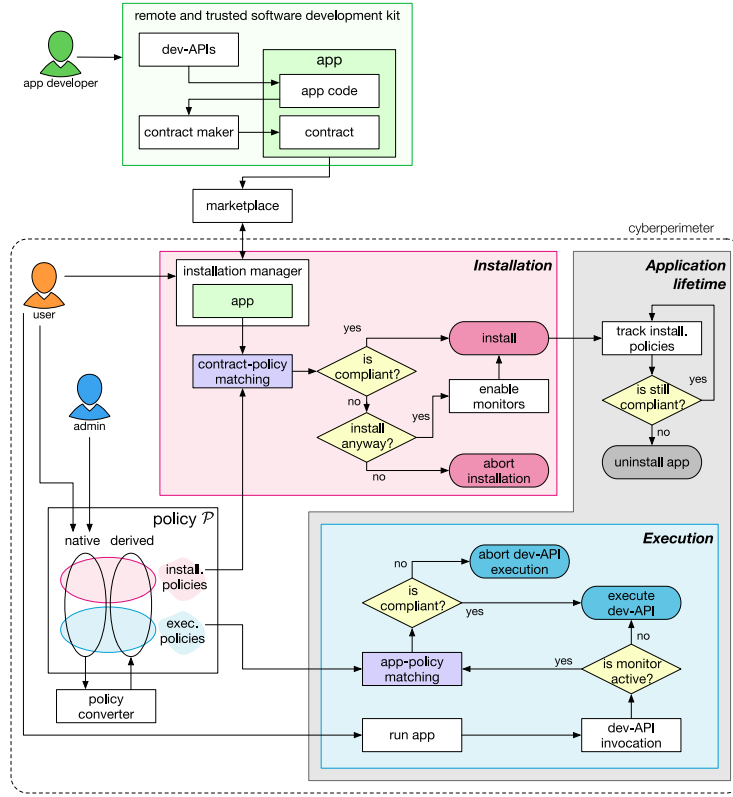


Fig. 4. Overall workflow.

#### 4. Proposed methodology

The framework proposed in this paper aims at exploiting the advantages of both the S×C paradigm and UCON to effectively and efficiently enforce security policies on applications executing in IoT systems. More in details, the expressiveness of ABAC policies, combined with the availability of effective tools for policy verification and continuous monitoring within the UCON framework, ensures precision and certainty of the enforcement at all stages: during installation, during execution, and throughout the period the application remains installed in the controlled system (*application lifetime*). It is important to note that during its lifetime, an application may be executed multiple times. On the other hand, by exploiting the S×C paradigm, it is possible to reduce the overhead imposed by UCON at execution time, by enforcing execution policies only on those applications which have the potential to violate them.

Our approach exploits Security by Contract to perform an effective and efficient control of the behavior of applications. The Security by Contract paradigm provides correctness guarantees as, for an application  $\mathcal{A}$  correctly modeled in the form of a contract  $C$ , the verification of the compliance  $\mathcal{A} \leq C$  and  $C \leq \mathcal{P}$  is aimed at checking whether the application  $\mathcal{A}$  can indeed violate the defined policy  $\mathcal{P}$ . Additionally,  $\mathcal{A} \leq C$  and  $C \leq \mathcal{P}$  compliance checks are conducted statically, i.e., before and during application installation, respectively, and, whenever the outcome is positive, they avoid the overhead due to the enforcement of  $\mathcal{P}$  at runtime, i.e., the  $\mathcal{A} \leq \mathcal{P}$  check is not performed, with a consistent gain in efficiency. However, the correct modeling of application behaviors and policies can prove to be a challenging task, especially in IoT and mobile systems, which are not based on functional programming languages, which normally resonate with the S×C paradigm and behavior verification through formal models.

Our approach consists in using a representation of the application behavior based on the ABAC model, together with the corresponding policies, to rigorously and consistently implement the three operations of the S×C paradigm. More specifically, our implementation of the S×C paradigm employs the UCON authorization workflow, with requests and policies defined as described in Section 2.2. The application ( $\mathcal{A}$ ) is the source code of the app, which consists of the set of dev-APIs included in the app code. The contract ( $C$ ) is represented as a set of UCON requests extracted from these dev-APIs, specifying the operations that the application can perform. Moreover, the policy ( $\mathcal{P}$ ) is represented as the set of all defined UCPs within the specific instance, e.g., a specific smart home.

The proposed workflow, along with the main architectural components, is depicted in Fig. 4. The figure extends the architecture presented in Fig. 3, and it represents the actors, the interaction between the defined components, the decision factors, and decision results of the compliance evaluations related to the contract, the policy, and the application.

The proposed approach aims to reduce the overall overhead required for access control checks during app's execution, by exploiting the effectiveness of the S×C paradigm. As outlined in the figure, this is achieved at three distinct moments of the application's lifecycle, where the usage control workflow is applied:

- At installation time, where the contract-policy matching is performed by batch testing all the UCON requests in the contract against the installation policies.
- During the application lifetime, since the installation policies could include ongoing conditions whose enforcement must be continuously performed during the whole application lifetime according to the UCON model. The violation of ongoing conditions causes the application removal.
- At execution time, only for the dev-APIs requiring runtime policy enforcement (*monitored dev-APIs*), by generating a UCON request and testing it against execution policies before performing the actual dev-API's functionality.

The proposed approach translates execution policies defined by policy makers into installation policies and vice versa. For clarity, *native* policies refer to those defined by the policy maker, while *derived* policies are the result of translating native policies. The translation from native execution policy to derived installation policy allows us to avoid the runtime enforcement of execution policies when it is not necessary, thus reducing UCON temporal overhead during applications execution. In particular, when an execution policy is defined, the framework automatically translates it into the corresponding derived installation policy, and each time an application is installed, the framework assesses whether the contract complies with the derived installation policy as well as the native installation policy. If the contract is compliant with the derived installation policy, we can be confident that the corresponding native execution policy will not be violated. This is because the derived installation policy is more restrictive than the original execution policy (insights into the derivation process are provided in Section 6.6). The reverse translation—from native installation policy to derived execution policy—is required for a different reason. Specifically, if an installation policy (pertaining to the dev-API  $d$ ) is violated but the user chooses to install the application anyway because it offers other functionalities, the dev-API  $d$  should still be monitored at runtime to prevent execution. This is achieved by activating a monitor on the dev-API, with an associated execution policy that can be evaluated when dev-API  $d$  is invoked during execution.

It is worth specifying that the policy  $\mathcal{P}$  includes both native and derived policies.

As shown in Fig. 4, the contract-policy matching process is conducted during installation. In this phase, the framework evaluates the superset composed of both native installation policies and derived installation policies. Similarly, during execution, but only when required—specifically, when the invoked dev-API is monitored—the app-policy matching process is performed. In this case, the framework evaluates the superset consisting of native execution policies and derived execution policies. It is important to note that once an app is installed, the installation policies that evaluated to permit during installation and include ongoing-conditions are continuously monitored throughout the application's lifetime, specifically in the context of these ongoing-conditions (see the box labeled “track install. policies” in the figure).

This methodology enables us to perform an accurate  $C \leq \mathcal{P}$  check at installation time, which, if evaluated as compliant, ensures that the app will never violate any of the installation policies in  $\mathcal{P}$  (see the box labeled “install” in the figure). As a result, it becomes safe to run the app without additional runtime checks, reducing the overhead caused by evaluating the policy for each request made for accessing a resource, i.e., at each dev-API invocation. On the contrary, if the  $C \leq \mathcal{P}$  compliance is not satisfied, (i.e., the evaluation related to some dev-API returns a Deny decision), but the app is installed anyway (see the box labeled “enable monitors” in the figure), the  $A \leq \mathcal{P}$  operation is enforced at runtime by using the standard UCON workflow. Consequently, during execution, the application  $A$  has the policy  $\mathcal{P}$  enforced through the PEPs inlined into each dev-API.

In the following sections, we demonstrate the soundness of the proposed approach by providing a formal representation of the UCON requests and policies, based on an extension of Kripke structures called Labeled State/Transition Systems. This formal modeling enables UCON requests to be evaluated against UCPs as a model-checking problem. Following this, we will present the actual implementation of the workflow using the UCON architecture outlined in [6], with specific adaptations to the overall workflow.

## 5. Models and verification

Automated verification through formal methods provides the means to reliably ensure that the application behavior meets the requirements of access control policies [13]. The guarantees are even stronger if both the application behavior and the policy specification can be modeled in the same formal framework. In this section, we present the formal model behind the representation of the contract  $C$  and the policy  $\mathcal{P}$ , as well as the techniques for the  $A \leq C \leq \mathcal{P}$  verification. In particular, as a common practice in the formal modeling and analysis of the behavior of systems, we employ transition systems (see, e.g., [14] and the inspiring models of [15–17]), which are directed graphs where nodes represent states (labeled with, e.g., propositions describing information about attributes) and edges model transitions between states (labeled with, e.g., security-critical actions causing state changes). We call these models Labeled State/Transition Systems (LSTSs).

**Definition 1.** A labeled state/transition system (LSTS) is a tuple  $(S, Act, Prop, T, L)$ , where  $S$  is a finite set of states,  $Act$  is the set of transition labels,  $Prop$  is the set of propositions,  $T \subseteq S \times Act \times S$  is a transition relation, and  $L : S \rightarrow 2^{Prop}$  is a state-labeling function mapping each state to the subset of propositions holding in the state.

In the following, we show how LSTSs are used to model contracts and policies, thus providing a unifying formal framework for the model-checking-based verification of the compliance relation behind the S×C model.

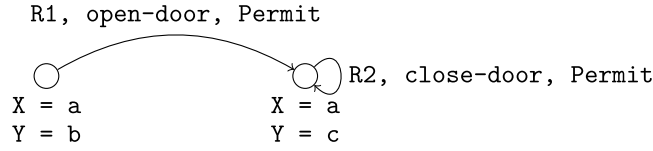


Fig. 5. LSTS representing the XACML policy in Listing 1.

### 5.1. Contract model

For the LSTS modeling a contract, the transition labels in *Act* represent the security-critical actions, i.e., the functionalities, whose execution may influence the values of subjects, resources, and environmental attributes; the propositions in *Prop* represent conditions over such attributes. Formally, propositions are given a classical first-order logic interpretation based on predicates over the variables representing the attributes of interest. Let  $V$  be the set of variables, where each  $v_i \in V$  ranges over a domain  $D_i$ . Then, a proposition over  $v_i$  is a predicate of one of the following forms:

- $v_i = d_i$ , with  $d_i \in D_i$ ;
- $v_i \neq d_i$ , with  $d_i \in D_i$ ;
- $v_i = \star$ , denoting that  $v_i$  may be mapped to *any* value of its domain  $D_i$  (classical interpretations that can be given are *unspecified* or *unknown* value).

Therefore, a state  $s \in S$  can be described as a valuation mapping every  $v_i \in V$  to a value in  $D_i \cup \{\star\}$  and, equivalently, the labeling  $L(s)$  is the set of propositions defined by such a valuation. In logical terms,  $L(s)$  can be viewed as the conjunction of these propositions (we will use interchangeably both notations). By default, in the following, we assume that, if not specified, the predicate associated with any variable  $v$  is  $v = \star$ .

With this interpretation of  $S$  in view, it holds that a transition  $(s, act, s') \in T$  expresses that action  $act$  is executed in a state in which the value of each variable  $v$  is  $s(v)$  and whose effect is leading to a state where the predicate associated with each variable  $v$  is determined by  $s'(v)$ .

### 5.2. Policy model

Similarly to the case of contracts, we formally model the policies in terms of LSTSs, which we automatically derive from XACML specifications. The policy  $\mathcal{P}$  can contain one or more collections of policies, aggregated through the XACML *policyset* element. This element defines the concept of policy aggregation, allowing multiple policies to be grouped and evaluated collectively. The SxC policy  $\mathcal{P}$  is intended as one or more policysets, each one containing a collection of policies among all the defined policies for a specific instance of the system. Without loss of generality, in the following, we consider the policy  $\mathcal{P}$  composed of only one policyset containing  $n$  policies and denoted as  $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ .

To illustrate the translation from XACML to LSTS, let us consider the following policy example (see Listing 1), where we provide pseudocode instead of XACML since the latter is not human-readable.

**Listing 1** Pseudocode of a portion of XACML policy with two rules. The first rule applies to the action-id `open-door`, the triggering condition is `X == a AND Y == b`, and its effect is `Permit`. Moreover, if the action is permitted, the value of attribute `Y` is updated to `c` (obligation). The second rule applies to the action-id `close-door`, and its effect is again `Permit`, provided that the condition is satisfied, while no obligations are defined.

```
...
target: action-id == open-door
rule R1:
  effect: Permit
  condition: X == a AND Y == b
  obligation:
    update: Y = c

target: action-id == close-door
rule R2:
  effect: Permit
  condition: X == a AND Y == c
...
```

In this example, each rule is mapped to a transition in the LSTS. The departing state is labeled with the conjunction of predicates expressing the rule condition. The arrival state is labeled with the same predicates updated on the basis of the rule obligation. The transition is labeled with the rule-id, the action-id, and the rule effect. Hence, regarding rules R1 and R2, the result of this mapping is given in Fig. 5.

To formally define the mapping from XACML to LSTS, we consider XACML conditions that are propositional logic formulas, where we assume that each proposition is of the form  $v = d$  or  $v \neq d$ . Hence, as such, they can be expressed in disjunctive normal form (DNF), i.e., as a disjunction of conjunctions of propositions. Given a rule  $r$ , we use the notation  $condition(r)$  to denote its condition (we extend the same notation to the other components of the rule, such as *obligation* and *effect*) and  $C \in condition(r)$  to denote a conjunction  $C$  of propositions that belongs to the DNF of the condition, e.g.,  $(X == a \text{ AND } Y == b)$  in Listing 1.

Formally, from an XACML policy  $P_i$ , a LSTS  $(S, A, Prop, T, L)$  can be derived by applying the following algorithm:

```

For each rule  $r \in P_i$ 
  For each  $C \in condition(r)$ 
     $S := S \cup \{s, s'\}$  such that  $L(s) := C \wedge L(s') := L(s)[obligation(r)]$ 
     $T := T \cup \{s, act, s'\}$  such that  $act := \langle r, action-id(r), effect(r) \rangle$ 

```

Let us comment on such an algorithm. Every conjunction  $C$  occurring in the DNF expressing a rule condition  $condition(r)$  is mapped to a transition of the LSTS. Such a transition departs from the state  $s$  associated with the conjunction  $C$ , i.e., the labeling  $L(s)$  is equal to  $C$ , and leads to the state  $s'$  that is obtained by updating the labeling of  $s$  on the basis of the (potential) obligation associated with the rule. Indeed, the expression  $L(s)[X]$  returns the propositions of  $L(s)$  (except those regarding the variables in  $X$ ) joined with the propositions of  $X$ , thus modeling the effect of an obligation  $X$ . The label of the transition is a triple, containing the rule-id, the action-id, and the effect of the rule. Note that during the execution of the algorithm, new states are added only whenever necessary, so that the state space  $S$  and the transition relation  $T$  are generated incrementally. It is also worth observing that, since a rule condition is a DNF potentially composed of several conjunctions and since each of these conjunctions generates a transition, a rule may be associated with several transitions, each one departing from a different state and depending on a different conjunction of literals in the DNF representing the rule condition.

We conclude by observing that XACML policies are accompanied by a *rule-combining algorithm* that is used to resolve potential conflicts among multiple rules that apply simultaneously. Hence, in the following, we assume that a policy  $P_i$  is actually a pair  $(S_i, RCA_i)$ , where  $S_i$  is the LSTS modeling  $P_i$ , and  $RCA_i$  is the specific rule-combining algorithm governing the policy. Similar to individual policies, policysets are governed by a *policy-combining algorithm*. This algorithm combines the single decisions derived from the policies of the policyset in an overall authorization decision.

### 5.3. $A \leq C \leq P$ verification

Since  $C$  is built automatically from  $A$ , we can assume that  $A \leq C$  is satisfied by construction. In the following, we present the formal verification  $C \leq P$ , which is based on a simulation check between  $C$  and  $P$ , and the formal enforcement  $A \leq P$ , which is activated at runtime whenever  $C$  does not meet the former validation.

#### 5.3.1. Model checking $C \leq P$

Given an application contract  $C$ , a set of policies  $\mathcal{P} = \{P_1, \dots, P_n\}$ , where  $P_i = (S_i, RCA_i)$ ,  $1 \leq i \leq n$ , and a policy-combining algorithm PCA governing the  $n$  policies, the objective of the  $C \leq P$  verification is to check the compliance of the contract with respect to the policyset. Such a matching between contract and policy requires checking every transition in the LSTS representing the contract against the policies in  $\mathcal{P}$ , if any, which refer to the request modeled by the transition.

Hence, on the one hand, it is worth identifying the subset of policies in  $\mathcal{P}$ , if present, that must be checked. The selection process depends on the labeling of the transition (and of the related departing state) associated with the contract request under analysis. Given a transition  $t = (s, act, \_)$  in the LSTS modeling the contract, let  $ps(t, \mathcal{P})$  be the policy selection function returning the set  $I$  of indexes referring to the policies  $P_i$ , with  $i \in I$ , that apply to the request modeled by  $t$ .

On the other hand, before performing the model checking of  $t$  against each  $P_i$ , with  $i \in I$ , we point out that the contract must be refined with the information of interest for each specific  $P_i$ . Indeed, the LSTS modeling the contract is defined on a reduced set of propositions compared to the policy, as the contract has no view on the attributes used by the policies in  $\mathcal{P}$ . Thus, to perform the model checking for each request, the departing state of the corresponding transition is extended with the attribute assignments that are relevant to the policy  $P_i$  of interest for the request.

Building on Reference Example 1, to evaluate the “allow-fast-charge-if-power-above-threshold” policy, a proper request concerning the dev-API `fast_charge()` needs to include the attribute referring to the “maximum power capacity”, since this attribute is essential for assessing the policy condition, i.e., the home maximum capacity is above the threshold of 5kW specified within the policy. However, the request within the contract does not include this attribute, as the contract is generated without knowledge of the specific security policies deployed in a given smart home instance. As a result, in order for the policy to be evaluated correctly, the request must be extended with the additional attributes required by the policy.

After this step, the model checking can be performed according to the following steps applied to every transition  $t = (s, act, \_) \in T$  of the LSTS  $(S, Act, Prop, T, L)$  modeling the contract refined as specified above.

First, the LSTS  $S_i$  of the policy  $P_i = (S_i, RCA_i)$ , such that  $i \in ps(t, \mathcal{P})$ , is projected over the states and transitions of  $S_i$  that are relevant for the evaluation of the contract transition  $(s, act, \_)$ . Second,  $RCA_i$  is applied to such a projection to determine the result of the evaluation, which can be one among permit, deny, not-applicable, or indeterminate.

To formally define the notion of relevance mentioned above, we introduce the following compatibility relation, which intuitively relates states with non-contradictory valuations.

**Table 1**  
Examples of XACML rule-combining algorithms.

RCA	Behavior
deny-overrides	if $(\_, \_, \_, \text{deny}) \in T'$ then deny else if $(\_, \_, \_, \text{permit}) \in T'$ then permit else not-applicable
permit-overrides	if $(\_, \_, \_, \text{permit}) \in T'$ then permit else if $(\_, \_, \_, \text{deny}) \in T'$ then deny else not-applicable
deny-unless-permit	if $(\_, \_, \_, \text{permit}) \in T'$ then permit else deny
permit-unless-deny	if $(\_, \_, \_, \text{deny}) \in T'$ then deny else permit

**Definition 2.** Given  $u, w \subseteq 2^{Prop}$ , we say that  $u$  and  $w$  are *compatible*, denoted  $u \cong w$ , if and only if whenever  $(v_i \text{ op } d_i) \in u$  and  $(v_i \text{ op } d'_i) \in w$ , with  $\text{op} \in \{=, \neq\}$ ,  $v_i \in V$ , and  $d_i, d'_i \in D_i$ , then  $d_i = d'_i$ .

Now, we are ready to define the operation of LSTS projection with respect to the valuation of a given contract state  $s \in S$  and an *act*-labeled transition in  $T$  departing from  $s$ .

**Definition 3.** Let  $s$  be a state of the LSTS modeling the application contract and *act* the label of a transition departing from  $s$ . Let  $S_i = (S_i, A_i, Prop, T_i, L_i)$  be the LSTS of the policy selected for evaluating the execution of *act* from  $s$ . The projection  $\text{Proj}(S_i, s, \text{act})$  of  $S_i$  with respect to the pair  $(s, \text{act})$  is the LSTS defined as  $(S' \subseteq S_i, A_i, Prop, T' \subseteq T_i, L_i)$ , which contains all and only the transitions  $(s', \langle r, \text{act}, \text{effect} \rangle, s'') \in T_i$  (and the related departure and arrival states  $s'$  and  $s''$ ) such that  $L_i(s) \cong L_i(s')$ .

Intuitively, given an application request modeled by the action *act* departing from a given state  $s$  of the LSTS modeling the application contract, the projection operation takes the LSTS of the selected policy  $P_i$  and extracts the portion related to those rule conditions that must be applied to the request. Notice that the projection could, in certain cases, be an empty LSTS. In any case, such a projected LSTS goes through the rule-combining algorithm  $\text{RCA}_i$  to determine the result of the request evaluation. In Table 1, we show some examples taken from the XACML standards.

The set of the results returned by the rule-combining algorithms of the selected policies are then combined by the policy-combining algorithm PCA to determine the final decision. Note that such a set could be empty if no policy is applicable to the application request.

Finally, the overall condition to verify is:

$$\forall t = (s, \text{act}, \_) \in T : \text{PCA}\{\text{RCA}_i(\text{Proj}(S_i, s, \text{act})) \mid i \in \text{ps}(t, \mathcal{P})\} = \text{permit}.$$

If all the transitions are permitted, then the contract  $C$  is compliant with the policy  $\mathcal{P}$  and no enforcing mechanism will be necessary at run time. On the other hand, if some transitions are not permitted, only the requests of the contract  $C$  related to such transitions will be subject to enforcement at run time.

### 5.3.2. Enforcing $\mathcal{A} \leq \mathcal{P}$

The runtime enforcement follows the same roadmap as discussed above. At each step, the action requested and the current configuration of the system attributes determine the labelings used to match each applicable policy. Formally, if such labelings are represented by the pair  $(s, \text{act})$  and assuming  $\text{ps}((s, \text{act}), \mathcal{P}) = I$ , then the action request is satisfied if

$$\text{PCA}\{\text{RCA}_i(\text{Proj}(S_i, s, \text{act})) \mid i \in I\} = \text{permit}.$$

If permitted by a specific policy  $P_i$ , all the obligations associated with the rules that permit *act* in  $\text{Proj}(S_i, s, \text{act})$  must be applied. More precisely, for each  $r$  such that  $(s, \langle r, \text{act}, \text{permit} \rangle, s') \in T'$ , then the labeling of  $s'$  is applied to the current state; in other words, this means that *obligation*( $r$ ) is applied to update  $s$ .

## 6. Implementation of S×C

In the former sections, we demonstrated the validity of implementing the Security by Contract workflow, by exploiting the Usage Control model. In the following, we discuss the steps performed for implementing the S×C model through the UCS.

### 6.1. Application, contract, and policy

Before describing the steps of the  $\mathcal{A} \leq C \leq \mathcal{P}$  verification, we define the concepts of application  $\mathcal{A}$ , contract  $C$ , and policy  $\mathcal{P}$  in our approach.

The application  $\mathcal{A}$  is the source code of the app, which consists of the set of dev-APIs  $\{d_i \mid i = 1, 2, \dots, n\}$  potentially invoked when the app is running. As discussed in Section 3.1, we assume that the apps use only the dev-APIs to define their smart-home related functionalities.

The contract  $C$  is automatically generated from the app code using the *contract maker* tool, included in the trusted software development kit provided as a remote service. The resulting contract is a manifest specifying the app's expected behavior, which is represented as a set of UCON requests corresponding to the dev-APIs included in the app code. Since the contract generation is performed in a controlled and trusted environment, this ensures the contract's correctness and guarantees that it cannot be counterfeited (see Section 6.3).

The contract maker generates the UCON requests (*installation requests*) to be embedded in the contract. We recall from Section 2 that a typical UCON request contains the attribute `subject-id`, which identifies the subject, the attribute `resource-id`, which identifies the resource, and the attribute `action-id`, which specifies the action that such a subject wants to perform on the resource.

The UCON requests generation proceeds as follows. Suppose the app code includes an invocation to the dev-API  $d_i$ . The UCON request  $R_j$ , derived from  $d_i$ , contains the attribute `subject-id` with value `marketplace`, the attribute `resource-id` with value `system`, and the attribute `action-id` with value `install`. Additionally, each request contains at least two other resource-related attributes: `device:device-type` and `device:action:action-id`, which correspond to the “dev-API type” and the “dev-API action” of  $d_i$  defined in Section 3.1, respectively, with values  $r_i$  and  $a_i$ . Optionally, parameters that specialize the behavior of  $d_i$ , such as “dev-API parameter  $x$ ”, may be included in the request, defined by an `AttributeId` specific to the case and with the value set to the specific value used during the invocation within the app code, if explicitly indicated. Thus, a UCON request of this type can be read as: the subject `marketplace` wants to install on the system an app that will perform the action  $a_i$  with parameter(s)  $p_i^x$  on the resource  $r_i$ . Referring to the earlier example of the SIFIS-Home dev-API `set_lamp_brightness(string id, uint value)` in Section 3, the values  $r_i$  and  $a_i$  are “lamp” and “set\_lamp\_brightness”, respectively, and  $p_i^2$  might be “90”. Note that the value for  $p_i^1$ , which represents the `string id`, would not be included, as the specific identifier of the lamp deployed in a particular smart home is unknown during development and, hence, cannot be present in the app code.

The policy  $\mathcal{P}$  describes the accepted and forbidden behaviors for all the apps running on the system, together with specific constraints on the apps that can be and cannot be installed. The policy  $\mathcal{P}$  is structured as two XACML *Policy Sets*: *installation policy set* and *execution policy set*, both including the UCPs defined by the policy makers to express their preferences on privacy, security, and safety. The installation policy set contains all the installation policies and is used at installation time to decide whether to install an app or not, depending on its functionalities included in the contract. The execution policy set contains all the execution policies and is used at execution time to either allow or deny specific app’s functionalities in real time.

The two XACML policy sets are regulated by the `deny-unless-permit` policy-combining algorithm. This ensures that if the evaluation result of the whole set of policies is either `NotApplicable` or `Indeterminate`, i.e., there is no rule defined for a specific request, the requested action will be denied. Moreover, each policy is regulated by the `permit-overrides` rule-combining algorithm. Each policy can include zero or more rules with a `Permit` effect and must have one unconditional `Deny` rule (*default-deny*). This ensures that, if the policy is applicable, the evaluation result will always be either `Permit` or `Deny`.

## 6.2. Application vs. Contract

In our architecture, we assume that the contract is automatically generated using a developer tool called the contract maker (as proposed in [12]), which operates within a trusted environment. More precisely, the contract maker generates the contract by processing the app’s source code, as discussed in Section 3.1. Considering that the contract is generated directly from the app, we notice that the compliance of the  $\mathcal{A} \leq C$  operation is complete and by construction [17]. Hence, the relationship between  $\mathcal{A}$  and  $C$  can actually be expressed with the stronger binary satisfiability relation  $\mathcal{A} \models C$ .<sup>3</sup>

According to the provided definition of  $\mathcal{A}$ , from which the list of dev-APIs potentially used by the app can be inferred, we can assert that the  $\mathcal{A} \models C$  operation is always verified by construction. Moreover, by employing hash digests and digital signature techniques, we can verify that the contract is not altered after it has been produced, thereby maintaining the consistency of the  $\mathcal{A} \models C$  operation.

## 6.3. Contract vs. Policy

The contract-policy matching ( $C$  vs.  $\mathcal{P}$ ) operation is normally performed at installation time of the app within a specific system. If the contract does not match the policy, the app should not be installed. However, if the user decides to install it anyway, the app’s behavior must be monitored to ensure compliance with the policy during execution, performing the  $\mathcal{A} \leq \mathcal{P}$  verification.

In the proposed architecture, the contract-policy matching operation is performed at installation time by evaluating the set of actions listed in the contract that the app intends to perform against the installation policies, which include both native and derived policies. Specifically, each UCON request in  $C$  is sent by the installation manager, acting as a PEP, to the UCS for evaluation against the installation policies. If any evaluation results in a `Deny` decision, then the  $C \leq \mathcal{P}$  is considered failed, meaning that the contract  $C$  does not match the policy  $\mathcal{P}$ . Consequently,  $\mathcal{A} \leq \mathcal{P}$  will be enforced at runtime. On the contrary, if all evaluations return a `Permit` decision, it means that the contract is compliant with the policy, and the app can be installed and subsequently executed without further runtime checks when dev-APIs are invoked.

The process of deriving installation policies from user-defined execution policies must ensure that if an app complies with a derived installation policy, it will also comply with the corresponding execution policy. Conversely, the fact that an app violates a derived installation policy does not imply that it violates the corresponding execution policy too. This is because, by construction, a derived installation policy is equivalent to or more restrictive than the native execution policy from which it originates. Specifically, when deriving the installation policy, rules based on mutable attributes (in the native execution policy) are deliberately excluded. These rules typically include conditions that enable permissions only under certain dynamic context. Therefore, by omitting them,

<sup>3</sup> The  $X \models Y$  operator is the base binary model-checking operator expressing satisfiability of predicates in  $Y$ , from the predicates  $X$ .



the installation policy becomes more conservative—effectively denying access in cases where the execution policy might have allowed it under favorable runtime conditions.

Building on Reference Example 5, we illustrate how an app that satisfies the derived installation policy also satisfies the native execution policy. Conversely, we show that an app violating the derived installation policy may still comply with the corresponding native execution policy. Let us suppose that the policy “allow-economy-or-night-wash” has just been defined by the policy maker. In such a case, this native execution policy is promptly transformed into a derived installation policy. During derivation, the rule concerning the time when the wash cycle can be started is omitted, as it relies on a mutable attribute—specifically, the current time—which can only be evaluated at runtime. Instead, the rule that allows the execution of the `washing_machine(string id, string washing_program)` dev-API when the `washing_program` is equal to `economic` is retained. Consequently, an app invoking such a dev-API with the `economic` washing program would satisfy the derived installation policy at installation time and the native execution policy at execution time. In contrast, an app invoking the dev-API with the `heavy duty` washing program would violate the derived installation policy at installation time, although it may still comply with the native execution policy at runtime. Specifically, at execution time, the same dev-API invocation with the `heavy duty` washing program at 9 PM would comply with the native execution policy, since the time-dependent rule would then be satisfied.

Given the rationale explained above, in the scenario where the app complies with a derived installation policy  $\mathcal{A} \leq \mathcal{P}$  can be skipped, as we can be confident that invoking any dev-API in  $\mathcal{C}$  will not violate the policy  $\mathcal{P}$ . Instead, in the scenario where the app violates a derived installation policy, if the user decides to install the app despite the violation,  $\mathcal{A} \leq \mathcal{P}$  is enforced at runtime.

During the contract-policy matching, for each UCON request in the contract, both a `tryAccess` and a `startAccess` message are issued, following the standard UCON workflow. In particular, the `tryAccess` message is sent to trigger the evaluation of the pre-conditions in the installation policy, to decide whether the app can be installed or not. Once the app is installed, the `startAccess` message is sent to trigger the evaluation of the ongoing-conditions in the installation policy, to verify that the policy is still satisfied while the app is installed. In this way, we model both the authorization to install an app and the authorization for the app to remain installed. Since attributes can change over time, if some specific attribute changes, then the app’s right to remain installed is re-evaluated by the UCS. In case of a Deny decision at re-evaluation time, the authorization is *revoked*, and the app is removed.

#### 6.4. Application vs. Policy

If  $\mathcal{C} \leq \mathcal{P}$  is not verified at installation time, the user will be notified that the application does not conform to the defined policies. In this situation, the user faces two choices: they can either choose to abort the installation process entirely, thus preventing the app from being installed on their system, or they can decide to proceed with the installation despite the compliance issues. If the user opts to continue with the installation, the policy will be directly enforced on the application by enabling active monitors, which will ensure that the app’s behavior aligns with the policies during its execution.

In the proposed approach, active monitors are implemented through PEPs integrated within the dev-APIs. Each dev-API is associated with a specific list of apps, accessible to the runtime environment, and to the related PEP. These lists define the apps for which the specific dev-API is monitored, ensuring that the monitoring mechanism operates at the level of individual dev-APIs. When a dev-API is invoked, it first checks whether the invoking app is in its monitored list. If the app is not in the monitored list, it means that the app has passed the  $\mathcal{C} \leq \mathcal{P}$  check for this dev-API. This check imposes a negligible overhead, as the list of monitored apps is based on hashmaps, with a complexity search of  $\mathcal{O}(1)$ . On the other hand, if the app is on the monitored list, we need to verify whether the specific operation is allowed by the execution policies in the current context.

This is achieved by tasking the PEP with structuring and sending a UCON request (*execution request*) for the requested action. Such a request contains the attribute `subject-id` with value the app’s name, the attribute `resource-id` with value the target device’s identifier, the attribute `action-id` with value the requested action, and possibly other attributes characterizing the parameters of the dev-API. This UCON request is then matched against the execution policy set and enriched with all the attributes needed for a meaningful evaluation, gathered through the PIPs. Practically, this workflow is handled by the runtime environment and the policy enforcement engine, which includes the Usage Control System. If the evaluation returns a `Permit` decision, the action is allowed, and the dev-API will proceed with its flow, executing its functionality. Conversely, if a `Deny` decision is returned, the execution of the dev-API will be aborted, thus preventing the functionality from being executed.

It is worth noting that, at execution time, it is possible to evaluate conditions based on real-time values of physical measures, such as temperature, humidity, or the number of people in a video frame. As a result, the authorization decision may vary at different moments, and the authorization can even be revoked after it has been initially granted if attribute mutability triggers a policy re-evaluation leading to a `Deny` decision.

Requests are performed each time a specific dev-API is invoked by the application, and the related action will not be performed until authorization is granted, i.e., a `Permit` decision is returned by the UCS. This implies a time overhead, which has to be minimized to avoid impacting the user experience. At installation time, even if a consistent number of UCON requests are evaluated in bursts, the delay caused by this is likely to be minor, given that the installation itself is typically a time-consuming operation. On the other hand, delays in executing a functionality expected to be real-time (e.g., a lamp turning on in response to a button press) might negatively impact the user experience by reducing the responsiveness of the expected service. This motivates the  $\mathcal{C}$  vs.  $\mathcal{P}$  evaluation, which, if successful, can eliminate this overhead.

### 6.5. Implementation of security policies

As discussed in Section 3.2, security policies are divided into two types: installation policies and execution policies. Both types are UCPs, hence written in U-XACML, and include pre-, ongoing-, and post-sections, in compliance with the UCON model.

In our approach, each policy follows a fixed structure. In simple terms, the policy's target is defined as a logical AND between two predicates involving the values of `subject-id` and `resource-id`, e.g., (`subject-id` equal to `marketplace`) AND (`resource-id` equal to `system`). This means the policy applies only when both the specified subject and resource are involved. Additionally, each rule within the policy targets specific actions, so the rule's target is implemented as a logical OR between `action-ids`. This ensures that the rule applies to any of the specified actions.

In XACML terms, the implementation details are as follows: the `<Target>` element of the policy consists of a conjunctive sequence of two `<AnyOf>` elements identifying the `subject-id` and the `resource-id`. Each rule's target includes an `<AnyOf>` element that contains a disjunctive sequence of `<AllOf>` elements. Each `<AllOf>` element encloses one `<Match>` element containing an attribute that identifies an `action-id`. For further details on the specific tags and elements used in XACML, please refer to the OASIS standard documentation [5].

We recall that each policy follows the `permit-overrides` rule-combining algorithm, meaning that a `Permit` decision takes precedence over a `Deny`. Moreover, in our approach, each policy must contain zero or more rules with a `Permit` effect along with an unconditional `Deny` rule (`default-deny`). This structure ensures that if the policy is applicable, the evaluation result will always be either `Permit` or `Deny`, preventing ambiguous decisions.

In the following, we present the peculiarities of both types of policies.

#### 6.5.1. Installation policies structure

The target of every installation policy contains `marketplace` as the `subject-id` and `system` as the `resource-id`. Moreover, the target of each rule within the policy focuses on a single action, i.e., it specifies `install` as the `action-id`. This results in a degenerate case with respect to the previously described rule target structure, where there is no effective use of OR in the `action-id`.

Installation policies differ from each other based on the attributes used to define the target app, resource type, and the action performed on that resource. These key attributes include the `app-name`, which identifies the app to which the policy applies, `device:device-type`, which specifies the type of resource (e.g., a lamp or thermostat), and `device:action:action-id`, which represents the action to be performed on the resource. Additionally, to further characterize the policy, installation policies can include attributes referring to the parameters of the dev-API to which the policy applies. This representation ensures that the requests contained in the contract, which represent the behavior of the dev-APIs, can match the installation policies.

An installation policy can target a specific app and a specific device type, such as `app-name = app-1` and `device:device-type = lamp`. Note that the `app-name` can be omitted, meaning the policy applies to all apps interacting with the specified device type, such as `device:device-type = lamp`, providing flexibility in the policy's scope.

#### 6.5.2. Execution policies structure

An execution policy grants authorization to a subject *S* to access a resource *R* and continue using it as long as certain conditions are met. In this context, the attribute `subject-id` identifies the specific app (e.g., `app-1`) while the attribute `resource-id` specifies the particular resource being accessed (e.g., `lamp-1`).

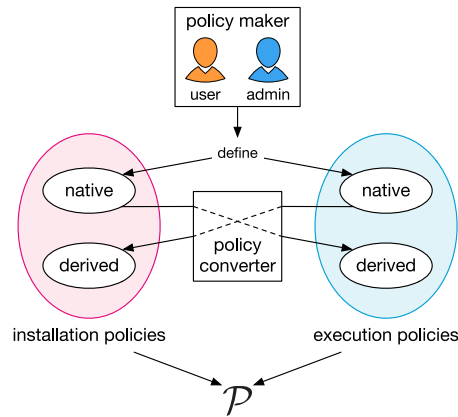
We assume that each subject and resource are associated with an attribute (`subject:priority` and `resource:priority`, respectively). The `subject:priority` indicates the priority of the subject, while `resource:priority` indicates the priority level of the subject that is currently utilizing the resource at a given moment. This priority is crucial for determining access situations, especially in cases where multiple subjects may contend for the same resource. In this context, a “handover” occurs when a subject with a higher priority than the currently controlling subject attempts to access the resource. The execution policy must be able to check if a handover condition exists, which involves comparing the priorities of the subjects involved.

Additionally, each resource is associated with the attribute `resource:busy`, which indicates whether the resource is in use for actions that require exclusive access for modification or is currently idle.

As previously described, rules are characterized within the `<Target>` element by a Boolean OR operation between actions. Specifically, for execution policies, all `action-ids` used within a rule must be either of type `set`, which refers to operations that modify the resource's state and necessitate exclusive access, or type `get`, which involves reading the resource's state without requiring exclusive access. The structure of each rule varies depending on the type of action it governs, and we explain these differences below.

For a rule of type `set`, the `ongoing-condition` is structured as a Boolean AND operation that combines the user-defined criteria with the value of the integer attribute `resource:priority`. This structure is designed to check whether the resource *R* is in a handover state, indicating that another subject with higher priority than the one currently using the resource is trying to acquire it. Furthermore, the rule includes a post-condition that always results in a `Permit` decision. This decision is coupled with an obligation that updates the Boolean attribute `resource:busy` to `false`, indicating that the resource *R* is now available and no longer in use.

Further details on how these attributes are used, along with the practical mechanisms employed to enforce these types of policies at runtime, are provided in Appendix B. This section elaborates on the underlying processes and tools that ensure policy compliance during the execution of dev-APIs, including mechanisms that ensure exclusive access to a resource.



**Fig. 6.** Policy types. The policy maker, either the user or the admin, defines native policies. From these native policies, the policy converter generates derived policies. All the installation policies and execution policies together compose the policy  $\mathcal{P}$ .

In contrast, rules of type *get* do not involve the `resource:busy` attribute or the post-obligation since these additions are only required to manage exclusive access to resources. Get operations simply read the state of a resource and thus do not require exclusive control over it.

The following listing illustrates the pseudocode for a rule of an execution policy. This example shows the structure of a rule of type *set*, including the necessary components for managing exclusive access, as well as the structure of a rule of type *get* when those components are omitted.

**Listing 2** Pseudocode for a rule of an execution policy.

```

Rule
  Effect: Permit
  Target: action-id = id_1 OR ... OR action-id = id_n

  pre-condition
    ...

  ongoing-condition
    {user-defined criteria}
    AND resource:priority != HANDOVER

  post-condition
    {always return Permit}

  post-obligation
    update: resource:busy = false

```

Removing the parts in red, we obtain a rule of type *get*. Conversely, including them gives us a rule of type *set*. Note that for a rule of type *set*, all `action-ids` from 1 to  $n$  must be of type *set*; similarly, for a rule of type *get*, all `action-ids` must be of type *get*.

## 6.6. Derivation of security policies

Security policies defined by the policy maker (either the user or the admin) are called native security policies. In our framework, we have both native installation policies and native execution policies. Once a native security policy is defined, the policy converter component can derive the corresponding derived policy when required. Specifically, the policy converter transforms native installation policies into derived execution policies and native execution policies into derived installation policies, as shown in Fig. 6.

### 6.6.1. Derived installation policies

The process of deriving installation policies from native execution policies is designed to ensure that if an app violates a native execution policy, the related derived installation policy is also violated. Therefore, this derivation process guarantees that the derived installation policy is at least as restrictive as the corresponding native execution policy. The derivation involves systematically examining each rule in the native execution policy to determine its inclusion in the derived installation policy. Rules containing one or more mutable attributes are excluded from the derived installation policy. Their dynamic nature makes them unsuitable for

enforcing stricter controls during installation, as the values of such attributes may not be available at installation time or, if available, they may differ from those at execution time. Note that in our approach, which employs rules with effect *Permit*, excluding such rules from the derived policy inherently results in a more restrictive policy.

Building on Reference Example 4, we now apply the derivation process to the execution policy associated with the *smartHVAC* app. The execution policy includes a rule with effect *Permit* for the action *turn\_HVAC\_on*, whose ongoing-condition specifies that all windows must remain closed for the action to be allowed. Since this condition depends on a mutable attribute—namely, the current status of the windows—the rule is excluded during the derivation of the installation policy. Consequently, the derived installation policy becomes more restrictive, containing only the default-deny rule for the action *turn\_HVAC\_on*, thereby imposing an unconditional denial on turning the HVAC system on. Hence, during the installation of an app that uses the *turn\_HVAC\_on()* dev-API, this results in a *Deny* decision. As a result, a monitor is activated for the dev-API. At runtime, the original execution policy will therefore be enforced, allowing the HVAC system to be turned on only if all the windows are closed.

The derivation process is triggered immediately after the native execution policy is defined and proceeds as follows. The target of the derived installation policy specifies that the policy is applicable when the values of the attributes *subject-id* and *resource-id* are equal to *marketplace* and *system*, respectively. These fixed values are independent of those specified in the original execution policy.

Rules with effect *Permit* that do not include mutable attributes in the native execution policy are selected to be included in the derived installation policy. For each selected rule, a corresponding rule is created in the derived installation policy according to the following steps:

1. The target specifies that the rule is applicable when the value of *action-id* is equal to *install*.
2. The pre-condition of the rule combines predicates involving the attributes *app-name*, *device:device-type*, and *device:action:action-id* using a logical AND. Additional attributes, e.g., dev-API parameters, are also included, maintaining the logical structure of the native policy. The predicates on such attributes are defined as follows:
  - *app-name*: The predicate compares the value of the attribute *app-name* with the value taken from the *subject-id* attribute of the native execution policy.
  - *device:action:action-id*: The predicate compares the value of the attribute *device:action:action-id* with the value representing the action performed on the device or resource, which is directly mapped from the *action-id* attribute in the native execution policy.
  - *device:device-type*: The predicate compares the value of the attribute *device:device-type* with the value representing the generalized resource (e.g., a specific type of device such as a lamp), which is determined by retrieving the related resource type using the *resource-id* attribute in the execution policy.
3. If the rule in the execution policy refers to an *action-id* of type *set*, the *resource:priority* attribute and the post-obligation are removed, as installation policies do not require mechanisms to ensure exclusive access to resources (see Listing 2).

To follow the policy structure defined in Section 6.5, a default-deny rule is then included, with target *action-id* set to *install*.

Note that if the rule's target in the execution policy specifies *n* *action-id*s, the derivation process creates *n* derived installation policies, with the generalization steps outlined above applied to each one.

In the following, we show that a derived installation policy maintains the same level of restriction or imposes stricter controls than the native execution policy it is based on. We use the notation  $rule(effect, decision\_factors)$  to denote a rule *rule* with the effect that can be either *permit* or *deny*, and with the Boolean expression *decision\_factors* that triggers the effect whenever evaluated to true.

**Definition 4.** For each rule  $e-rule(permit, decision\_factors)$  in execution policy *e-P*, the corresponding rule *i-rule* in the derived installation policy *i-P* will be

$$i-rule = \begin{cases} e-rule'(permit, decision\_factors) & \text{if } derivable(decision\_factors) = true \\ null & \text{otherwise,} \end{cases}$$

where *derivable* is a predicate that evaluates to true if *decision\_factors* satisfies the requirements allowing the rule to be derived, and  $e-rule'(permit, decision\_factors)$  is the installation rule derived from  $e-rule(permit, decision\_factors)$  according to the derivation mechanism described above.

**Definition 5.** Let *e-req* be an execution request (see Section 6.4), and let *i-req* be an installation request (see Section 6.1). We say that *i-req* is the corresponding installation request of the execution request *e-req* if both pertain to the same dev-API. In this context, the former refers to an access request performed for installing an app that includes the dev-API, while the latter refers to an access request made at runtime for executing the same dev-API.

**Theorem 1 (Derived Installation Rule).** Let *e-P* be an execution policy, and let *i-P* be the installation policy derived from *e-P*. Let *e-rule* be a rule of *e-P*, and let  $i-rule \neq null$  be the corresponding rule of *e-rule* in *i-P*, derived according to Definition 4. Let us consider any execution request *e-req* evaluated by *e-P* and the corresponding installation request *i-req*, evaluated by *i-P*. If *e-rule* evaluates to true on *e-req*, then *i-rule* will evaluate to true on *i-req*.

**Proof.** By virtue of [Definition 4](#), we have that *decision\_factors* and effect of *i-rule* are the same as that of *e-rule*, and by virtue of [Definition 5](#), the correspondence between *i-req* and *e-req* guarantees that the result of the evaluation of *i-rule* is the same as that of *e-rule*, from which the result immediately follows.  $\square$

**Theorem 2 (Derived Installation Policy).** Let *e-P* be an execution policy consisting of *N* rules *e-rule<sub>j</sub>*, with  $j \in [1, N]$ , each having the effect permit. Let *i-P* be the installation policy derived from *e-P*, which, according to [Definition 4](#), contains  $M \leq N$  rules *i-rule<sub>k</sub>*, with  $k \in [1, M]$ . Let us consider any execution request *e-req* that is evaluated to deny by *e-P*. Then, the corresponding installation request *i-req* will also be evaluated to deny by *i-P*.

**Proof.** Let us consider the execution request *e-req* and its corresponding installation request *i-req*. In order to receive a permit evaluation for *i-req*, the policy *i-P* should include at least one rule with effect permit that is applicable. Since, according to [Definition 4](#), each rule in *i-P* is obtained by derivation from a rule of *e-P*, the subset *X* of rules in *i-P* which can return a permit decision for *i-req* is a subset of the subset *Y* of rules in *e-P* which can return a permit decision for *e-req*. If, by theorem hypothesis, *e-req* is evaluated to deny by *e-P*, then *Y* is empty, from which it turns out that *X* is empty too and, as a consequence, also *i-req* is evaluated to deny by *i-P*.  $\square$

### 6.6.2. Derived execution policies

The following describes the derivation process, performed by the policy converter, to generate a derived execution policy from a native installation policy. This process is not triggered immediately after the native policy is defined; rather, it occurs only if the native installation policy returns a Deny decision during the contract-policy matching, and the user opts to proceed with the installation, thereby enabling the monitor and enforcing  $\mathcal{A} \leq \mathcal{P}$  at runtime.

The derived execution policy includes Permit rules extracted from the native installation policy to express the precise conditions under which a dev-API may be allowed at runtime. Otherwise, the derived policy would be limited to exclusion through Deny rules only, preventing the dev-API's functionality from ever being executed. Therefore, for each rule with effect Permit in the native installation policy, a corresponding rule is created in the derived execution policy as follows:

1. The value of the attribute `action-id` in the rule's target is compared with the value that the attribute `device:action:action-id` was compared with in the rule of the installation policy. If the `action-id` is of type *set*, two additional elements are incorporated into the rule of the derived execution policy, as shown in the pseudocode in [Listing 2](#):
  - In the ongoing-condition, the condition that checks whether the `resource:priority` attribute differs from the special value `HANDOVER`.
  - In the post-obligation, the update statement that sets the `resource:busy` attribute to `false`.
2. The rule's pre-condition is created by combining, via a Boolean AND operation, the full expression of both the pre-condition and the ongoing-condition of the native installation policy. The rule's ongoing-condition is created by replicating the full expression of the ongoing-condition of the native installation policy. During this operation, the attributes `device:device-type` and `app-name`, if present in any of the conditions, are not reported in the derived execution policy since they are later used to define the policy target, as explained below. These modifications preserve the logical structure of the rule while ensuring that the ongoing-condition is applied before granting access to the resource, as well as when usage is ongoing. By checking both conditions before granting access, the derived execution policy ensures that the access is only granted when both conditions permit it.

To follow the policy structure defined in [Section 6.5](#), a default-deny rule is then included, with target `action-id` matching the value of the attribute `device:action:action-id` in the native policy.

Regarding the policy target, if the installation policy includes a predicate that specifies the value of the `app-name` attribute, this value is directly used to define the `subject-id` the derived execution policy is applicable to. If `app-name` is not present in the installation policy, this does not pose a problem, as the derivation process occurs during an ongoing app installation.

Finally, the `device:device-type` attribute from the installation policy is used to retrieve the list of identifiers for all currently deployed devices of that type, represented by their corresponding `resource-id` attributes. For each of these `resource-ids`, a derived execution policy is created by following the steps outlined above. As a result, multiple derived execution policies are generated, each containing a `<Target>` element that specifies the `resource-id`, identifying a specific resource of type `device:device-type`, and the `subject-id`, indicating the app currently being installed.

Building on [Reference Example 1](#), the native installation policy “allow-fast-charge-if-power-above-threshold” is subject to derivation if, during the installation phase, it evaluates to Deny (e.g., because the contract includes a request for the `fast_charge()` dev-API, but the electricity supply contract supports 3kW only) and the user chooses to proceed anyway—possibly motivated by other functionalities offered by the app. Specifically, the derivation process generates one execution policy for each deployed device of type “charger”, embedding the corresponding device identifier in the policy's target. For simplicity, consider a smart home scenario where only one such device exists. The corresponding derived execution policy will include a rule targeting the action `fast_charge`, through the attribute `action-id`. The resulting execution policy includes a pre-condition that ensures the electricity supply contract supports at least 5 kW. The ongoing-condition of the native installation policy is preserved as-is and replicated directly into the ongoing-condition of the derived execution policy. Thus, both the pre- and the ongoing-condition in the derived execution policy will check that the electricity supply contract supports at least 5 kW.

## 7. Experimental evaluation

In this section, we outline the workflow considered in our experiments, detail the performance metrics measured, and describe the specific setup employed for conducting the experiments. The purpose of our experiments is to quantify the temporal overhead introduced by our approach both during installation and execution, and to determine whether and how this cost affects the user experience.

In Section 7.2, we assess the overhead introduced during installation, which results from the contract-policy matching. This series of experiments investigates how the temporal overhead varies with the complexity of an app, specifically the size of the set of UCON requests within its contract.

In Section 7.3, we assess the overhead introduced during execution, which is due to the policy enforcement phase for a monitored dev-API. This series of experiments aims to quantify how the temporal overhead varies with the complexity of the execution policy, specifically focusing on the number of mutable attributes within the ongoing-condition of the policy.

In Section 7.4, we evaluate the overhead associated with the revocation process. Notably, this process is only possible after the UCS has issued a positive response to a *startAccess* request, as it presumes an ongoing usage session. The experiments involve deliberate modifications to attribute values during the usage session to trigger policy re-evaluation and revocation. These experiments are designed to measure the *inconsistency time*, defined as the interval between the moment an attribute value changes and the time when the PEP terminates access to the resource upon receiving the revocation decision from the UCS. In other words, the inconsistency time quantifies the period during which access to a resource is still possible but not authorized due to a change in the attributes governing access.

Finally, please note that no specific experiment has been conducted to measure the performance of the *endAccess* operation, as this action is executed at the end of a resource usage session and does not introduce overhead affecting the user experience. This is because the *endAccess* operation is typically lightweight since it does not include any authorization policies. Consequently, it is rarely the focus of performance evaluations in related literature [6,18,19].

### 7.1. Experimental setup

In our experiments, we use a Raspberry Pi 4 Mod. B Rev 1.4 equipped with 8 GB of RAM as the smart device. Indeed, as described in Section 3, the Raspberry Pi is a representative example of smart device capable of supporting both app installation and execution. This device is capable of downloading apps from the Internet via the marketplace and initiating the installation procedure, which includes the contract-policy matching. Additionally, the device runs apps that invoke monitored dev-APIs, thus enforcing the policy on the application.

On the implementation side, the app is packaged as a Docker image stored in a Docker repository, namely, ghcr.io. In addition to the app code, the Docker image includes the contract—represented as a set of UCON requests—detailing the security-relevant operations the app intends to perform.

The IoT device also runs a Java-based implementation of the UCON framework,<sup>4</sup> which is queried to determine whether a security-relevant operation is allowed or not.

Each test repetition begins with the launch of the UCS, followed by a sequence of 50 warm-up cycles designed to establish a steady-state operational environment. These warm-up cycles simulate a complete UCON workflow to ensure the system is fully initialized and operating consistently before performance measurements are taken. Each warm-up cycle involves the following steps: the PEP sends a *tryAccess* message to the UCS and waits for a response; upon receiving permission, it proceeds to send a *startAccess* message and again waits for the UCS's decision; finally, the PEP sends an *endAccess* message, concluding the interaction and awaiting confirmation from the UCS.

The warm-up process is structured to enable uninterrupted interactions, ensuring that the UCS authorizes all requests from the PEP during this phase. This ensures that the subsequent test results are not influenced by initialization or transient conditions. Following the completion of the warm-up cycles, the specific test under consideration is executed, and the time performance metrics are recorded to evaluate the behavior under the designated experimental conditions.

For each configuration being tested, 50 independent repetitions are performed to ensure statistically sound results. Each repetition includes the warm-up phase and the execution of the test itself. During the test phase of each repetition, the start and end timestamps for all relevant time intervals are recorded, enabling the computation of the duration for each interval.

The time intervals collected from the 50 repetitions are averaged to obtain a representative value for the performance metrics. Additionally, 95% confidence intervals are calculated to provide a measure of the variability and reliability of the results. Notably, in all the tests conducted, the confidence intervals are extremely narrow, indicating low variability in the results. These intervals are present in all the graphs that follow, but in some of them, they are so small to be barely appreciable.

#### 7.1.1. Structure of the tested policies

In our experiments, we focus on a deliberately challenging yet well-defined policy structure to rigorously assess system performance. Specifically, we use policies composed of a single rule, designed to reflect a worst-case scenario in terms of evaluation

<sup>4</sup> <https://sssg-dev.iit.cnr.it/marco-rasori/new-ucs>



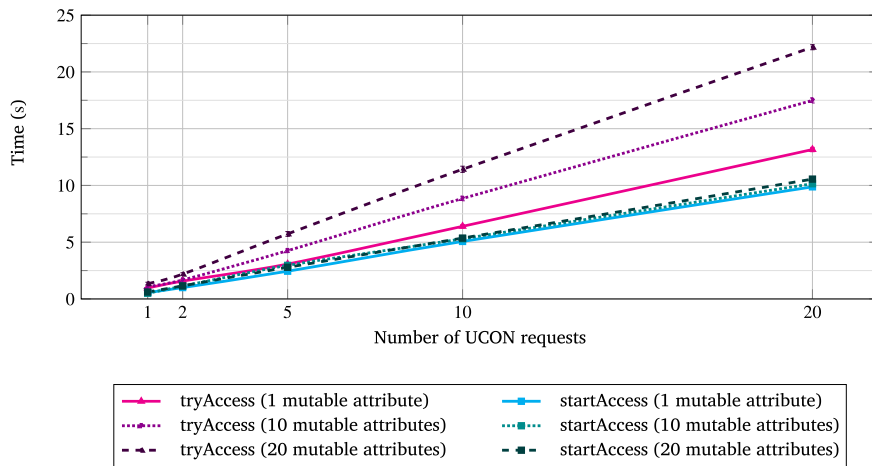


Fig. 7. Installation tests. Average time for the tryAccess and the startAccess phases varying the number of requests and number of mutable attributes. All times are recorded at the PEP.

overhead. Within this rule, we systematically vary the number of mutable attributes included in the ongoing-condition, ranging from 1 up to  $n$ . In particular, the predicates involving these attributes are logically combined using a Boolean AND operator, meaning that all of them must be simultaneously satisfied for the rule to be applicable.

This configuration forces the UCS to evaluate all included attributes in every access decision, thereby maximizing the computational load. It provides a controlled and consistent framework to analyze how performance scales with policy complexity.

To contextualize this choice, consider a more typical policy composed of multiple rules—for instance, 10 rules each involving 5 attributes, summing to a total of 50. If such a policy employs the permit-overrides combining algorithm and all rules have a Permit effect, the UCS can potentially make a decision after evaluating just the first applicable rule. In such a case, only 5 attributes may be evaluated, as opposed to all 50—resulting in significantly lower computational overhead. Conversely, our single-rule policy design requires the UCS to evaluate all attributes, establishing a stringent and consistent performance baseline.

The pre-condition of the tested policies remains constant and involves two attributes. Moreover, the overall structure of the policies mirrors that of the installation policies for the experiments described in Section 7.2, and that of the execution policies for the experiments in Sections 7.3 and 7.4.

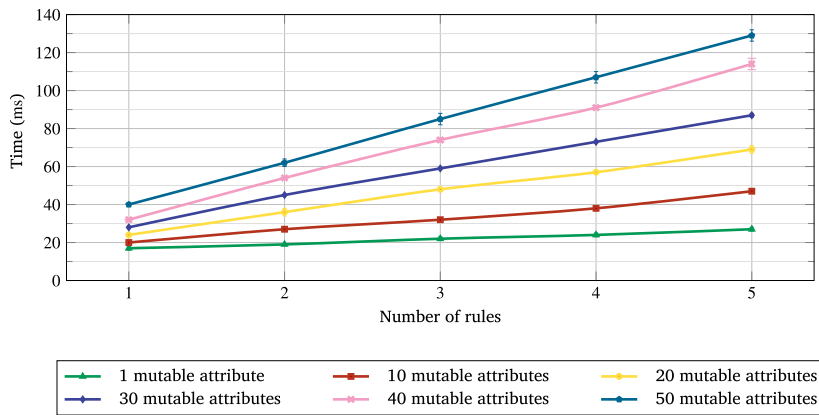
## 7.2. Installation time campaign: Contract-policy matching

This section investigates the time required to determine the contract-policy compliance, which introduces temporal overhead to the total *installation time* of an app. The installation time is defined as the duration from the moment the user clicks the Install button in the installation manager to the moment the app is ready to be executed on the user's device. Typically, this time includes the time for downloading the app (which depends on its size and download speed) and performing checksums to verify its integrity. In standard scenarios, the installation time ranges from a few seconds to several tens of seconds. With the proposed approach, this time also includes the duration required to evaluate the contract-policy matching, which refers to the time taken to evaluate all access requests from the app's contract against the corresponding policies. From a UCON perspective, this is the period between sending the first *tryAccess* request and receiving the response to the final *tryAccess* request.

In the following, we outline the assumptions made during these tests. First, we assume there is at least one applicable policy for each access request sent by the app. Second, we assume all *tryAccess* requests are granted permission (Permit), and similarly, all the consecutive *startAccess* requests aimed at carrying on the UCON workflow are also successful (Permit). To understand how the contract size and policy complexity affect the contract-policy matching time, we define a series of contract and policy configurations. In particular, we created a set of contracts containing 1, 2, 5, 10, and 20 requests, and a set of installation policies in which the number of mutable attributes in the ongoing-condition is set to 1, 10, and 20. Then, for each of these policies, we measured the time required for evaluating the contract-policy compliance for each of the contracts we defined.

Fig. 7 illustrates the outcomes of the installation time campaign, highlighting the contract-policy matching times varying the complexity of both contracts and policies. During each test, the PEP sends all the *tryAccess* messages and waits for all the responses from the UCS. The series labeled “tryAccess” in the figure refer to the time interval starting from when the PEP sends the first *tryAccess* message to when it receives the last response to a *tryAccess* message. This time interval constitutes the actual overhead introduced during the app installation process. Note that the evaluation of a *tryAccess* message determines whether access to a resource can be granted or not, enabling the installation process to proceed.

However, to enforce the conditions that depend on mutable attributes, the UCON workflow continues beyond this step: The PEP sends all the *startAccess* messages and waits for the corresponding responses from the UCS. Note that this part of the workflow can execute in parallel with the installation of the application, mitigating its impact on the overall installation time. The series labeled



**Fig. 8.** Time to convert an installation policy to execution policies, varying the number of rules and number of mutable attributes per rule within the installation policy.

as “startAccess” in the figure refer to the time interval starting from when the PEP sends the first *startAccess* message to when it receives the last response to a *startAccess* message.

As observed in the figure, both the *tryAccess* and *startAccess* times increase linearly with the number of requests, which aligns with the expected behavior as the contract complexity increases. This trend holds consistently across all the policy configurations tested, regardless of their complexity. What becomes more evident, however, is the impact of policy complexity on the per-request evaluation time. As the number of mutable attributes in the ongoing-condition increases, the UCS needs more processing time to evaluate each individual request. These additional delays accumulate with each request, leading to a higher overall overhead. In the most demanding scenario—contracts with 20 requests and policies containing 20 mutable attributes—the cumulative *tryAccess* time reaches approximately 22.2 s.

Nevertheless, we argue that such highly complex policies are rarely encountered in practice. Real-world policies typically involve a smaller number of attributes, favoring configurations that offer a reasonable trade-off between expressiveness and performance. Therefore, the results presented here provide a useful upper-bound reference for the system’s behavior under computationally demanding conditions.

In the graph, we note that the *tryAccess* time increases sub-linearly. For example, in the scenario with 1 mutable attribute, the *tryAccess* time is approximately 900 ms for a single request. However, when multiple requests are sent back-to-back, the total time does not scale as the sum of individual request times; instead, it is lower (e.g., about 13 s for 20 requests). This behavior can be explained by looking at how the *tryAccess* time is composed. It consists of three components: (i) the time for the first request to travel from the PEP to the UCS, (ii) the time required by the UCS to evaluate all requests, and (iii) the time for the final response to travel from the UCS back to the PEP. This is because of pipelining: requests following the first one are processed sequentially at the UCS, where each new request is evaluated as soon as the previous one completes. In the case of a single request, the round-trip communication delay—that is, the time to send the request and receive the response—accounts for a significant portion of the total time. When multiple requests are sent back-to-back, however, the communication time overlaps with UCS’s evaluation time. As a result, the additional time per request stabilizes around 650 ms, primarily reflecting UCS-side evaluation. A similar trend is observed during the *startAccess* phase.

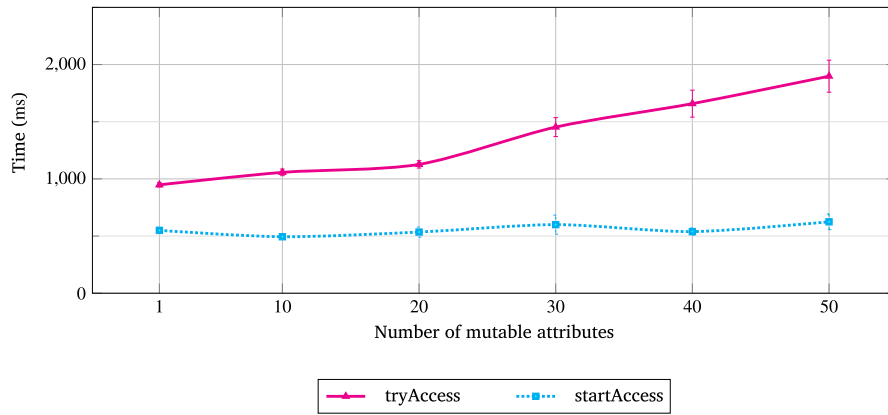
### 7.2.1. Policy derivation overhead

The previous tests focus on fully compliant contracts that do not require policy derivation during installation. As part of the installation time campaign, we also conduct additional tests to estimate the time required to derive execution policies from native installation policies. This may happen during the contract-policy matching when the evaluation of a native installation policy yields a *Deny* decision and the user opts to proceed with the app’s installation.

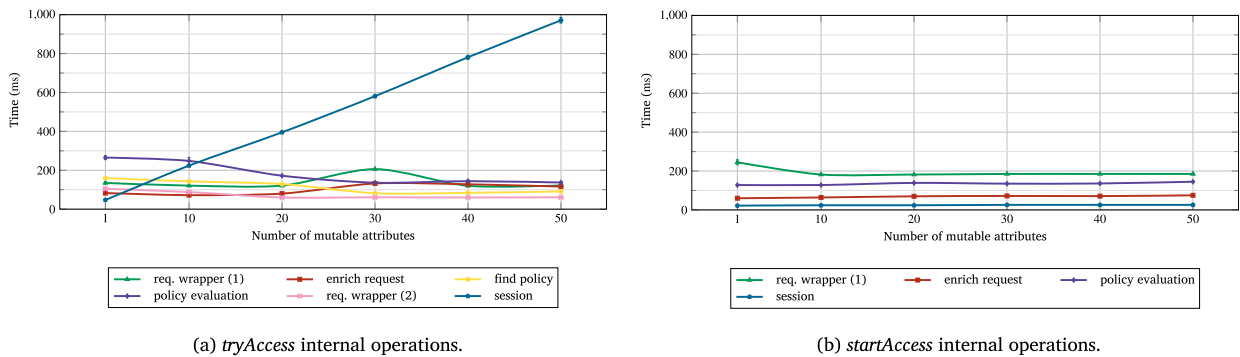
Since these tests are not intended to measure policy evaluation time, we do not adopt the single-rule structure that forces the UCS to consider all the attributes. Instead, we employ installation policies composed of multiple rules, each tailored to a specific device type and action. With this setup, the policy converter generates a number of derived policies equal to the number of rules in the native policy, provided that in our configuration we have one device per type deployed.

Fig. 8 presents the results for native installation policies, with a number of rules ranging from 1 to 5. Each curve in the plot represents a different configuration, where the number of mutable attributes within each rule’s ongoing-condition varies. The most complex configuration tested involved a native installation policy with 5 rules, where each rule contained 50 mutable attributes combined using a Boolean AND operator. This configuration resulted in five distinct derived execution policies after derivation.

Notably, even under these high-complexity conditions, the time required for policy derivation is minimal—approximately 130 ms—which is negligible when compared to the installation times discussed in the previous section. Therefore, in scenarios involving partially compliant applications, the overhead associated with policy derivation does not introduce a significant additional cost.



**Fig. 9.** Execution tests. Average time for the tryAccess and the startAccess phases varying the number of mutable attributes in the policy. All times are recorded at the PEP.



**Fig. 10.** UCS analysis of execution tests. Internal UCS operations varying the number of mutable attributes in the policy. All times are recorded at the UCS.

### 7.3. Execution time campaign: Policy enforcement

This section investigates the temporal overhead introduced by the proposed approach when a monitored dev-API is invoked. When the dev-API is monitored, a UCON request is generated, and a *tryAccess* message is sent to the UCS (for further details, see the pseudocode of a generic dev-API in [Appendix B](#), Listing 3). If the UCS returns Deny as result of the policy evaluation, the actual functionality is not executed. Otherwise, if the response is Permit, the portion of the code initiating the access to the resource begins, and, a *startAccess* message is subsequently sent to the UCS.

In this set of experiments, a Docker image is launched, and the related container runs an app that invokes a monitored dev-API. When the dev-API is invoked, we measure the time it takes to create the UCON request and obtain a response to the *tryAccess* message from the UCS, which includes the time to compute the access decision. Moreover, we measure the time it takes to evaluate the ongoing-condition of the policy after sending the *startAccess* message. We assume that the *tryAccess* request is granted permission (Permit), and the *startAccess* request is also successful (Permit).

[Fig. 9](#) illustrates the outcomes of the execution time campaign, highlighting the evaluation times for single-rule policies of varying size, ranging from 1 to 50 mutable attributes. The number of attributes in the pre-condition of the policies remains constant across all the configurations, and it could be expected that the time interval corresponding to the *tryAccess* phase would remain constant as well. However, this time interval increases slightly in a linear fashion as the number of mutable attributes in the ongoing-condition increases.

We assumed that the communication overhead would remain consistent across configurations because the PEP and UCS operate on the same device (as described in [Section 3.1](#)), and the payload size of their messages does not vary with the test configuration. Therefore, to explain the observed behavior, we conducted a deeper analysis of the UCS's internal operations during the *tryAccess* and *startAccess* phases, which are depicted in [Fig. 10](#).

The analysis revealed that most inner phases—details of which are beyond the scope of this discussion—appear to be independent of the number of attributes in the ongoing-condition for the configurations tested. However, the series labeled “session”, specifically in [Fig. 10\(a\)](#), demonstrated a clear dependency on policy complexity when handling a *tryAccess* message. In this case, the UCS creates a session entry in the database maintained by the Session Manager, recording all the necessary session-related information for future reference. This process becomes increasingly resource-intensive as policy complexity grows. In contrast, when handling

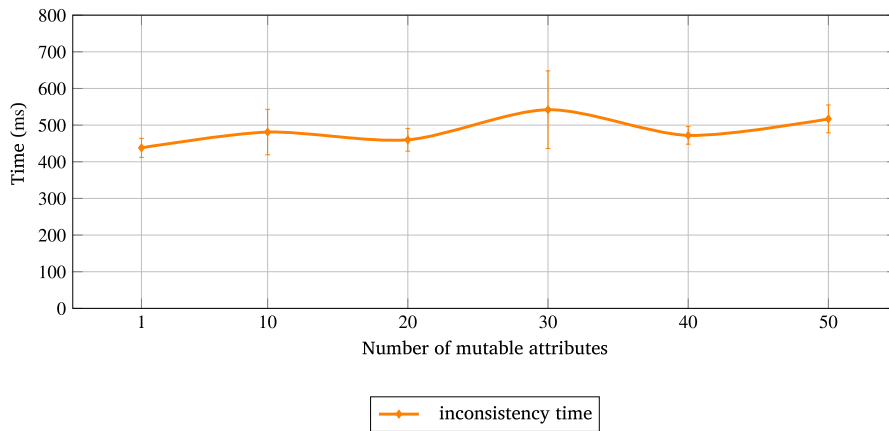


Fig. 11. Revocation tests. Average time for the inconsistency time varying the number of mutable attributes in the policy. All times are recorded at the PEP.

a *startAccess* message, the operations corresponding to the “session” series, shown in Fig. 10(b), involve the UCS retrieving the corresponding session entry from the database and updating its status. Unlike the former case, these operations exhibit relatively stable performance regardless of policy complexity.

With reference to the proposed approach, the critical operation of a monitored dev-API can be executed only after the completion of the *tryAccess* phase. As outlined, this phase introduces a temporal overhead that is directly influenced by the policy complexity. Fig. 9 shows that in the most demanding scenario tested, i.e., a policy with 50 attributes in the ongoing-condition, this overhead reaches approximately 1800 ms. For simpler configurations, the overhead is notably lower, around one second.

#### 7.4. Revocation time campaign: Inconsistency time

This section investigates the time required for the revocation process, focusing specifically on the measurement of the inconsistency time. The experimental workflow mirrors that of the execution time campaign described in Section 7.3, leveraging the same configurations and testbed setup. As part of the workflow, a Docker image is launched, and the associated container runs an application that invokes a monitored dev-API. Then, the PEP sends a *tryAccess* message to the UCS, starts the access to the resource, and sends a *startAccess* message to the UCS. In this campaign, however, the focus shifts to the revocation phase.

During the active usage session, we modify the value of a mutable attribute, recording the initial time  $t_i$ . This change initiates the revocation workflow, which involves the following steps: (i) attribute change detection: the UCS identifies the attribute change and begins re-evaluating the policy; (ii) policy re-evaluation: the UCS assesses the updated attribute values against the policy, producing a new decision (in our case *Deny*, which means revocation); (iii) decision communication: the UCS sends a *revokeAccess* message to the PEP; (iv) access termination: the PEP enforces the decision by terminating access to the resource.

We measure the final time,  $t_f$ , when the access termination is complete. The inconsistency time is then calculated as  $t_{Inc} = t_f - t_i$ . This duration reflects the period during which unauthorized access persists following a change in the governing attributes.

Fig. 11 illustrates the outcomes of the revocation time campaign, highlighting the inconsistency times for policies of varying size, ranging from 1 to 50 mutable attributes. During the active usage session, a mutable attribute is deliberately modified, triggering the revocation workflow. The temporal overhead associated with this process is shown in the figure as the series labeled “inconsistency time”.

For policies with a single mutable attribute, the inconsistency time is approximately 440 ms, while for policies with 50 mutable attributes, it rises to about 520 ms. This slight increase in inconsistency time as the number of mutable attributes grows is attributable to the re-evaluation phase. In this phase, the UCS evaluates the ongoing-condition of the policy against the enriched UCON request. As the number of mutable attributes grows, the UCS must query more PIPs, increasing the time required to gather attribute data. Simultaneously, the policy evaluation performed by the PDP becomes more complex due to the larger number of attributes in the ongoing-condition, further extending the re-evaluation phase.

The observed variability in the inconsistency times is mainly due to the attribute change detection step, i.e., the time the UCS needs to detect that the value of a mutable attribute has changed and to trigger the re-evaluation. In our implementation, PIPs periodically poll the AMs to retrieve fresh attribute values. As a consequence, the detection delay depends on when the attribute change occurs relative to the polling cycle: if the change happens right after a polling round, it may take nearly an entire polling interval to detect it, whereas if it happens just before polling, detection is almost immediate.

Even in the most demanding scenario tested, the temporal overhead introduced by the revocation workflow is still within an acceptable range, remaining under 550 ms.

### 7.5. Discussion: Benefits of the proposed approach

The experimental evaluation demonstrates good scalability in the number of requests during installation, handling complex contracts in approximately 22 seconds, even in the most demanding scenarios. While this operation does affect the user experience—particularly in configurations involving a high number of requests and complex policies—it is performed only once, at installation time. This initial waiting is compensated at runtime, as the contract-policy matching determines which dev-APIs do not require monitoring.

If an app is fully compliant at installation, no runtime monitors are activated, resulting in no overhead for the user during runtime. Conversely, if an app is only partially compliant, monitors are activated selectively for specific dev-APIs. At runtime, these dev-APIs will allow or deny critical operations dynamically, depending on mutable attributes and contextual conditions. Therefore, by shifting much of the policy compliance work to the installation phase, the runtime overhead during execution is minimized.

Regarding partially compliant application scenarios, we note that the contract-policy matching time represents the dominant portion of the evaluation overhead. Although policy derivation introduces an additional step in this case, our experiments have shown that its impact is negligible compared to the time required for contract-policy matching. In the worst case (a policy with 5 rules and 50 mutable attributes per rule), the derivation time was approximately 130 ms. For an average case (3 rules and 30 mutable attributes per rule), this value was about 60 ms. This further confirms that policy derivation does not significantly affect the overall evaluation time.

Experimental validation confirmed the system's ability to handle complex scenarios efficiently. For example, inconsistency times remain under 500 ms even for policies with numerous attributes, enabling rapid revocation of resource access when attribute changes occur. This ensures prompt responses to critical conditions, such as deactivating a space heater if environmental factors become unsafe.

To further characterize the UCS's efficiency, we also evaluated its resource footprint during the installation time campaign, using two representative configurations shown in Fig. 7. Specifically, we considered (i) the simplest case of a single request with one mutable attribute in the policy, and (ii) the most demanding case of 20 back-to-back requests, with 20 mutable attributes in each policy. In the single-request test, average CPU utilization was roughly 25%, and heap memory usage averaged about 400 MB. In contrast, under the most demanding scenario, average CPU utilization increased to about 30%, with peaks reaching up to 55%, and heap memory usage averaged approximately 500 MB.

By combining efficient policy evaluation with practical optimizations, this approach provides a robust and scalable framework while maintaining a good user experience.

## 8. Related work

Investigating the discrepancies among Access Control (AC) policies, application models, and implementation is of paramount importance (see, e.g., [20]) and the automated, formally-correct, and centrally-controlled management of AC policies is the goal of various formal approaches in different IoT-based domains (see, e.g., how solvers for constraint satisfaction problems are used in the context of Software-Defined Networking [21]). In this section, we examine some relevant contributions in this setting.

The authors in [22] propose a hybrid ABAC-RBAC (Role-Based Access Control) model tailored to smart home environments, which can also be generalized to other IoT architectures. Authorization rules are checked by evaluating propositional logic formulas. The paper focuses on finding a scalable method for defining policies that can utilize a large set of attributes to map the complexity of a smart home environment. However, the considered subjects are always human users, which differs from our work, as we focus on the analysis and control of applications behavior. The authors in [23] present an application of the ABAC paradigm based on the MQTT communication protocol for IoT environments. The paper focuses on a specific application, demonstrating how the ABAC paradigm can be used to enforce access control in the MQTT protocol. Differently from our work, this paper is implementation-specific and envisions devices as subjects, instead of applications. Another application of ABAC in IoT environments is discussed in [24], which proposes a framework named RECON. This framework minimizes the collection of attributes by using machine-learning-based predictions of attribute values. The application domain is a generic smart home environment, where the predicted attribute values are physical measures based on LSTM neural networks.

The authors of [25,26] present an implementation of the SxC applied in the IoT domain, presenting also a possible integration with legacy devices. Unlike our work, they are not focused on the authorization and access control aspects. In [27], the SxC paradigm is developed for the Industrial IoT setting, where the devices come equipped with manufacturer usage description (MUD) profiles as a manifest stored on blockchains and verified against SxC-based smart contracts. In recent years, several approaches rely on the MUD standard to cope with security issues in IoT-enabled scenarios, see, e.g., [28] and the references therein. Among these works, it is worth mentioning [29], which applies a formal ad-hoc semantic framework to validate MUD profiles against policies, and [30], which integrates MUD-compliant devices with the SxC framework.

Another approach relying on LTS-like finite-state machines to express state transitions of attributes is presented in [31]. This work is specific to usage control and has been used to model attribute update relationships, as part of the obligations construct. An effort to formalize and extend the expressiveness of the usage control paradigm is presented in [32]. The paper formalizes the relationship between entities of the UCON paradigm and proposes relaxing some hypotheses to express state transitions. Unlike our work, this specification is not based on the XACML standard architecture, but on the ALFA language, an open-source language defined as part of a commercial product provided by Axiomatics. Furthermore, the paper does not present any specific application, while our work demonstrates its applicability to a specific, general environment and is supported by experimental results. An

application of the S×C paradigm to the IoT domain is presented in [33], which discusses how to adapt the S×C paradigm to a sensor-driven application. The work focuses on defining prescriptive actions for services in the IoT and fog domains but does not address access control functionalities. An effort to combine contract-based security and access control is presented in [34], which defines a relationship between the Design by Contract [35] paradigm and RBAC. Our approach, conversely, focuses on more general paradigms such as ABAC, which, being attribute-based, is far more expressive than RBAC.

In [36], a general-purpose, formal framework for the specification and enforcement of obligation policies is proposed, which relies on semantics based on LSTS-like state-transition systems. Finite-state machines are also used in [37] to model generic AC policies that are then model checked through SPIN [38]. Our idea of modeling in a unifying formal framework, both the system model and the policy model, is also common to other general-purpose approaches. This is done, e.g., in [39], which focuses on policy enforcement and employs the higher-level paradigm of process algebra for modeling purposes, without, however, relying on any real-world policy and/or enforcement model. Finally, separate treatments of the policy language (which do not consider the app/contract formalization) are proposed for XACML in a logical framework [40], in the SAT modulo theory (SMT) [41], and with respect to a denotational-style formal semantics [42].

## 9. Conclusion

In this paper, we have proposed a IoT authorization management framework that integrates the two paradigms of Security by Contract and Usage Control to enhance the security of smart home applications. This framework leverages S×C for the formal verification of application behavior against predefined policies and UCON for continuous monitoring and enforcement during application execution, ensuring comprehensive security coverage with low impact on system performance.

Additionally, we have implemented and validated the proposed methodology through formal models based on labeled state/-transition systems, using UCON requests and policies to represent S×C's contracts and policies, respectively. To ensure consistent policy enforcement throughout the application lifecycle, we introduced a policy translation mechanism that aligns installation and execution policies.

Experimental evaluations on a real testbed have demonstrated the feasibility of our approach. These experiments have provided insights into the performance of the framework, showing its ability to manage policy enforcement and revocation processes with low overhead. Our results confirm that the combination of S×C and UCON offers a robust solution for addressing security challenges in existing smart home frameworks.

Future work will extend the framework to consider behaviors and policies including also quantitative aspects, such as time and probabilities, as done, e.g., in [17,43].

## CRedit authorship contribution statement

**Marco Rasori:** Writing – original draft, Validation, Project administration, Data curation, Writing – review & editing, Visualization, Software, Methodology, Conceptualization. **Paolo Mori:** Supervision, Methodology, Conceptualization, Writing – review & editing, Project administration, Investigation. **Andrea Saracino:** Writing – review & editing, Methodology, Conceptualization, Supervision, Funding acquisition. **Alessandro Aldini:** Writing – review & editing, Supervision, Investigation, Formal analysis, Writing – original draft, Methodology, Funding acquisition, Conceptualization.

## Declaration of competing interest

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

## Appendix A. Full attribute identifiers

For readability, throughout the paper we have referred to XACML attributes using short names, avoiding the verbosity of full `AttributeId` strings while ensuring clarity and avoiding ambiguity. Table A.2 reports the mapping between the short names used in the main text and their corresponding full identifiers, including their category and data type. Each row provides: (i) the short name used in the text, (ii) the corresponding full name (i.e., the actual `AttributeId`), (iii) the category (e.g., subject, resource), and (iv) the data type (e.g., string, boolean).

This table is provided for reference to help the reader to interpret the examples and policies discussed in the paper.

## Appendix B. Mechanisms for runtime policy enforcement and resource control in dev-API execution

In this section, we describe the enforcement mechanisms we defined for dev-APIs and their internal structure, in order to support usage control at runtime. We also present a classification of dev-APIs based on their behavior and resource interaction.

Each dev-API can be categorized into one of the following three types: *temporal dev-APIs*, *state-changing dev-APIs*, and *control dev-APIs*.



**Table A.2**

Full name and short name for the AttributeId of the attributes used within the paper.

Short name	AttributeId (full name)	Category	DataType
subject-id	urn:oasis:names:tc:xacml:1.0:subject:subject-id	subject	string
resource-id	urn:oasis:names:tc:xacml:1.0:resource:resource-id	resource	string
action-id	urn:oasis:names:tc:xacml:1.0:action:action-id	action	string
resource:busy	urn:it:cnr:iit:xacml:1.0:resource:busy	resource	boolean
subject:busy	urn:it:cnr:iit:xacml:1.0:subject:busy	subject	string
device:device-type	urn:it:cnr:iit:xacml:1.0:resource:device:device-type	resource	string
device:action:action-id	urn:it:cnr:iit:xacml:1.0:resource:device:action:action-id	resource	string
resource:priority	urn:it:cnr:iit:xacml:1.0:resource:priority	resource	integer
subject:priority	urn:it:cnr:iit:xacml:1.0:subject:priority	subject	integer
app-name	urn:it:cnr:iit:xacml:1.0:resource:app-name	resource	string

- Temporal dev-APIs: These dev-APIs trigger actions that last over time, such as playing an audio file through the `play_audio` dev-API, as in Reference Example 3. The action is sustained, with the software continuously accessing the resource (e.g., the audio playing). These actions can terminate in one of three ways: (i) naturally, when the event completes (e.g., an audio file finishes playing); (ii) through revocation, when a change in an attribute value causes a revocation; and (iii) through the invocation of a *control* dev-API, which explicitly stops the action, such as calling `stop_audio` to halt the audio.
- State-changing dev-APIs: These dev-APIs trigger actions whose execution happens almost instantly from a software perspective, such as turning a light on. However, the action itself is not limited to the moment of execution—it endures over time as long as its effect persists in the physical realm (i.e., the light remains on). Unlike temporal dev-APIs, whose actions may conclude naturally, this type of action does not end by itself. Instead, it persists until it is explicitly revoked or overridden by another command (a control dev-API) that cancels its effect, such as turning the light off.
- Control dev-APIs: These dev-APIs are designed specifically to stop actions initiated by temporal or state-changing dev-APIs. For example, a `stop_audio` dev-API can halt the playback started by a `play_audio` dev-API, and a `turn_off_light` dev-API can reverse the effect of a `turn_on_light` dev-API.

The implementation of control dev-APIs differs slightly from that of other dev-APIs, as their primary function is to terminate or override previously triggered actions.

In the following, we outline a methodology for enforcing execution policies and managing resource control in dev-APIs that implement functionalities subject to usage control. While both *get* and *set* operations necessitate effective usage control, *set* operations demand more stringent regulation to ensure exclusive access. To address these needs, dev-APIs employ a combination of runtime policy checks and resource control mechanisms, particularly for operations that alter the state of a resource, such as turning on a lamp.

The following pseudocode illustrates the structure and functionality of a typical temporal or state-changing dev-API. Control dev-APIs will be discussed in the following section.

### Listing 3 Pseudocode for a generic dev-API.

```

1  dev_API(args) {
2      bool monitored = is_monitored(app.name, dev-API.name);
3      if (monitored) {
4          request = create_UCON_request(args);
5          resp_try = tryAccess(request);
6          if (resp_try.evaluation == "Deny") {
7              return;
8          }
9      }
10     bool success = run_functionality(args);
11     if (monitored && success) {
12         resp_start = startAccess(resp_try.sessionId);
13         if (resp_start.evaluation == "Deny") {
14             terminate_access();
15             endAccess(resp_try.sessionId)
16         }
17     }
18 }

```

In cases where the dev-API is not monitored, the function bypasses any usage control checks and directly executes the `run_functionality()` function, meaning that  $\mathcal{A} \leq \mathcal{P}$  is not enforced at runtime. Conversely, when the dev-API is monitored, two distinct phases of policy enforcement are applied. In the first phase, the conditional block checks if access is allowed by creating a usage control request (via `create_UCON_request()`) and submitting it to the UCS using `tryAccess()`. If the response evaluation from the UCS is *Deny*, the functionality is aborted, and the process terminates early. If access is granted, the second phase is initiated after the `run_functionality()` is executed. This phase checks whether the conditions for continuing access

are valid by issuing a `startAccess()` call. If this ongoing evaluation fails, access is revoked, and the dev-API terminates access to the resource by invoking `terminate_access()` and sending an `endAccess()` message to the UCS.

For dev-APIs that refer to *set* operations, such as turning on a lamp, the internal code of the `run_functionality()` function is more complex than for *get* operations. This complexity arises from the need to ensure exclusive access to the resource, preventing concurrent modification of its state. The following pseudocode illustrates the structure of the `run_functionality()` function:

**Listing 4** Pseudocode for functionalities of type *set* and *get*, enforcing exclusive access for *set* operations. The integer value `HANDOVER` is reserved to signal resource handover and is not used as a priority value.

```

1  run_functionality(args) {
2      if (operation_type == "get") {
3          api_code();
4          return true;
5      } else if (operation_type == "set") {
6          sem_wait(resource:lock);
7          if (subject:priority < resource:priority) {
8              resource:priority = HANDOVER;
9              wait_until (resource:busy == false);
10             resource:busy = true;
11             resource:priority = subject:priority;
12             if (!monitored) {
13                 subscribe(resource:priority);
14             }
15             api_code();
16             sem_post(resource:lock);
17             return true;
18         } else {
19             if (monitored) {
20                 endAccess(sessionId);
21             }
22             sem_post(resource:lock);
23             return false;
24         }
25     }
26 }

```

Since *get* operations—such as reading data from a device—do not require mutual exclusion, their internal code is considerably simpler, consisting only of the invocation of `api_code()`.

For *set* operations, the pseudocode in Listing 4 uses a semaphore on `resource:lock` (line 6) to ensure exclusive access to the resource. In line 7, the system determines if the current process has sufficient priority to access the resource by comparing the priority of the current process (i.e., `subject:priority`) to `resource:priority`. In this context,  $P_h$  represents a process with higher priority trying to acquire the resource, and  $P_l$  is the lower-priority process currently using the resource.

Since it has a higher priority,  $P_h$  sets the `resource:priority` to `HANDOVER` (line 8), signaling to  $P_l$  that it must release the resource. Once  $P_l$  terminates its access, it sets `resource:busy` to `false`, releasing the resource for  $P_h$  to acquire. After the resource becomes available, the higher-priority process  $P_h$  sets `resource:busy` to `true`, signaling that it has acquired the resource. It then updates `resource:priority` to reflect its own priority (line 11), indicating that it is now the process using the resource. Finally, the process proceeds with the functionality (line 15).

The mechanism by which  $P_l$  detects that `resource:priority` has been set to `HANDOVER` depends on whether the dev-API is monitored or non-monitored.

For monitored dev-APIs, an ongoing session is present at the UCS. A re-evaluation of the policy is triggered when  $P_h$  sets `resource:priority` to `HANDOVER`, resulting in revocation of  $P_l$ 's access. The PEP of  $P_l$  receives a *revokeAccess* message, releases the resource, and sends an *endAccess* message to the UCS, which updates the attribute `resource:busy` to `false` through the post-obligation.

For non-monitored dev-APIs, a simpler mechanism is enforced. Upon gaining access to the resource,  $P_l$  monitors `resource:priority` in a separate thread. When  $P_h$  sets `resource:priority` to `HANDOVER`,  $P_l$  detects this change, terminates its access, and sets `resource:busy` to `false`, allowing  $P_h$  to proceed with executing `api_code()` (line 15).

The `api_code()` function spawns a new thread and runs asynchronously, allowing the `run_functionality()` function to return to its caller for further execution. Once back in the caller, if the dev-API is monitored, a *startAccess* message is sent to the UCS to ensure that the access can be maintained (line 11 of Listing 3). If the response from the UCS is *Deny*, the PEP immediately terminates access to the resource by invoking `terminate_access()`, which aborts the `api_code()` execution, and subsequently sends an *endAccess* message to the UCS. Conversely, if the response is *Permit*, the dev-API flow continues, and access to the resource is maintained.

Note that the thread running `api_code()` can also terminate naturally once it has completed its operation, releasing the resource. Even in this scenario, if the dev-API is monitored, the PEP still sends the *endAccess* message to the UCS. This task is handled by the thread running `api_code()`, ensuring that the resource is properly released and the UCS is informed of the operation's completion.

For both monitored and non-monitored dev-APIs, once resource usage has ended, `resource:busy` is set to `false` and `resource:priority` is reset to the special value `IDLE`, indicating that the resource is free and no longer in use.

### B.1. Control dev-API implementation

Control dev-APIs are responsible for terminating an operation initiated by either a temporal or state-changing dev-API. Subsequently, the ongoing session related to such an operation is also closed. Like the other dev-APIs, control dev-APIs can be monitored. However, they do not invoke `startAccess()` since the operations they perform require access control only.

The following pseudocode illustrates the implementation of a typical control dev-API:

**Listing 5** Pseudocode for a generic control dev-API.

```

1  control_dev_API(target_session_id)
2    bool monitored = is_monitored(app.name, control-dev-API.name);
3    if (monitored) {
4      request = create_UCON_request(target_session_id);
5      resp_try = tryAccess(request);
6      if (resp_try.evaluation == "Deny") {
7        return;
8      }
9
10   terminate_functionality(target_session_id);
11   endAccess(target_session_id);
12
13   if (monitored) {
14     endAccess(resp_try.sessionId);
15   }
16 }
```

In this pseudocode, the function first checks whether the control dev-API is monitored. If it is, a UCON request is created and submitted via `tryAccess()`. If access is denied, execution stops. If access is granted, or if the control dev-API is non-monitored, the function proceeds with `terminate_functionality()`, which either deactivates the resource in the physical realm (e.g., switching off a lamp) or terminates the operation in software. Immediately after, `endAccess(target_session_id)` (line 11) is invoked to formally conclude the session at the UCS associated with the terminated functionality, pertaining to a previously invoked temporal or state-changing dev-API.

For monitored control dev-APIs, an additional *endAccess* message

(`endAccess(resp_try.sessionId)`, line 14) ensures proper cleanup of the control dev-API session itself. In contrast, for non-monitored control dev-APIs, the access control steps are skipped, and the termination process, including its associated session closure, is executed directly.

### Data availability

Data will be made available on request.

### References

- [1] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, E. Vetillard, et al., Security-by-Contract (SxC) for software and services of mobile systems, in: *At Your Service – Selected Papers on EU Research on Software and Services*, MIT Press, 2008, pp. 49–54.
- [2] G. Costa, A. Lazouski, F. Martinelli, I. Matteucci, V. Issarny, R. Saadi, N. Dragoni, F. Massacci, Security-by-contract-with-trust for mobile devices, *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 1 (4) (2010) 75–91, <http://dx.doi.org/10.22667/JOWUA.2010.12.31.075>.
- [3] E.M. Clarke, O. Grumberg, D.A. Peled, *Model checking*, 1st Edition, MIT Press, 2001, URL <http://books.google.de/books?id=Nmc4wEaLXFEC>.
- [4] J. Park, R. Sandhu, The UCON<sub>ABC</sub> usage control model, *ACM Trans. Inf. Syst. Secur.* (TISSEC) 7 (1) (2004) 128–174.
- [5] eXtensible Access control markup language (XACML) version 3.0 plus errata 01, 2017, URL <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html>.
- [6] E. Carniani, D. D'Arenzo, A. Lazouski, F. Martinelli, P. Mori, Usage control on cloud systems, *Future Gener. Comput. Syst.* 63 (2016) 37–55.
- [7] M.T. Paratore, E. Marchetti, A. Calabrò, From plain english to XACML policies: An AI-based pipeline approach, in: F. Cicciozzi, L.F. Pires, F. Bordeleau (Eds.), *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2025*, Porto, Portugal, February 26–28, 2025, SCITEPRESS, 2025, pp. 85–96, <http://dx.doi.org/10.5220/0013357200003896>.
- [8] B. Lang, N. Zhao, K. Ge, K. Chen, An XACML policy generating method based on policy view, in: *2008 Third International Conference on Pervasive Computing and Applications*, vol. 1, IEEE, 2008, pp. 295–301.
- [9] Islandora XACML editor, 2016, Online (GitHub / LYRASIS Wiki), Provides a GUI for editing XACML policies for Fedora repository objects, URL [https://github.com/Islandora/islandora\\_xacml\\_editor](https://github.com/Islandora/islandora_xacml_editor).
- [10] P.G. Morcillo, A.J. Lázaro, G.D. Tormo, UMU-XACML-editor, 2009, Software project page, University of Murcia, A Java-based GUI editor for XACML 2.0 policies, URL <https://sourceforge.net/projects/umu-xacml-editor/>.
- [11] Axiomatics, ALFA eclipse plugin, 2012, Press release and documentation, Free plugin for Eclipse IDE to author XACML 3.0 policies using ALFA, URL <https://axiomatics.com/news/press-releases/axiomatics-releases-free-plugin-for-the-eclipse-ide-to-author-xacml3-0-policies>.
- [12] L. Ardito, L. Barbato, P. Mori, A. Saracino, Preserving privacy in the globalized smart home: The SIFIS-Home project, *IEEE Secur. Priv.* 20 (1) (2021) 33–44.
- [13] L. Golightly, P. Modesti, R. Garcia, V. Chang, Securing distributed systems: A survey on access control techniques for cloud, blockchain, IoT and SDN, *Cyber Secur. Appl.* 1 (2023) 100015.
- [14] C. Baier, J.-P. Katoen, *Principles of model checking*, The MIT Press, 2008.

- [15] E. Clarke, O. Grumberg, D. Peled, *Model checking*, The MIT Press, 1999.
- [16] M.H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications, in: S. Leue, P. Merino (Eds.), *Formal Methods for Industrial Critical Systems*, Springer, 2008, pp. 133–148.
- [17] A. Aldini, A. La Marra, F. Martinelli, A. Saracino, Ask a(n)droid to tell you the odds: probabilistic security-by-contract for mobile devices, *Soft Comput.* 25 (2021) 2295–2314.
- [18] A. La Marra, F. Martinelli, P. Mori, A. Saracino, Implementing usage control in internet of things: a smart home use case, in: 2017 IEEE Trustcom/BigDataSE/ICESS, IEEE, 2017, pp. 1056–1063.
- [19] A. La Marra, F. Martinelli, P. Mori, A. Rizos, A. Saracino, Introducing usage control in MQTT, in: *Computer Security: ESORICS 2017 International Workshops, CyberCPS 2017 and SECPRE 2017*, Oslo, Norway, September 14–15, 2017, Revised Selected Papers 3, Springer, 2018, pp. 35–43.
- [20] V.C. Hu, R. Kuhn, D. Yaga, et al., Verification and test methods for access control policies/models, *NIST Spec. Publ. 800 (192)* (2017) <http://dx.doi.org/10.6028/NIST.SP.800-192>.
- [21] D. Brighenti, J. Yusupov, A.M. Zarca, F. Valenza, R. Sisto, J.B. Bernabe, A. Skarmeta, Automatic, verifiable and optimized policy-based security enforcement for SDN-aware IoT networks, *Comput. Netw.* 213 (2022) 109123, <http://dx.doi.org/10.1016/j.comnet.2022.109123>.
- [22] S. Ameer, J. Benson, R. Sandhu, Hybrid approaches (ABAC and RBAC) toward secure access control in smart home IoT, *IEEE Trans. Dependable Secur. Comput.* 20 (5) (2023) 4032–4051, <http://dx.doi.org/10.1109/TDSC.2022.3216297>.
- [23] P. Colombo, E. Ferrari, Access control enforcement within MQTT-based internet of things ecosystems, in: *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies, SACMAT '18*, Association for Computing Machinery, New York, NY, USA, 2018, pp. 223–234, <http://dx.doi.org/10.1145/3205977.3205986>.
- [24] A. Blower, G. Kotonya, A predictive authorization approach for IoT environments, in: *Proceedings of the 11th International Conference on the Internet of Things, IoT '21*, Association for Computing Machinery, New York, NY, USA, 2022, pp. 80–87, <http://dx.doi.org/10.1145/3494322.3494333>.
- [25] A. Giaretta, N. Dragoni, F. Massacci, S×C4IoT: A security-by-contract framework for dynamic evolving IoT devices, *ACM Trans. Sens. Networks* 18 (1) (2021) 12:1–12:51, <http://dx.doi.org/10.1145/3480462>.
- [26] A. Giaretta, N. Dragoni, F. Massacci, IoT security configurability with security-by-contract, *Sensors* 19 (19) (2019) 4121, <http://dx.doi.org/10.3390/S19194121>.
- [27] P. Krishnan, K. Jain, K. Achuthan, R. Buyya, Software-defined security-by-contract for blockchain-enabled MUD-aware industrial IoT edge networks, *IEEE Trans. Ind. Informatics* 18 (10) (2022) 7068–7076.
- [28] J.L. Hernández-Ramos, S.N. Matheu, A. Feraudo, G. Baldini, J.B. Bernabe, P. Yadav, A. Skarmeta, P. Bellavista, Defining the behavior of IoT devices through the MUD standard: Review, challenges, and research directions, *IEEE Access* 9 (2021) 126265–126285.
- [29] A. Hamza, D. Ranathunga, H.H. Gharakheili, M. Roughan, V. Sivaraman, Clear as MUD: Generating, validating and applying IoT behavioral profiles, in: *Proceedings of the 2018 Workshop on IoT Security and Privacy*, in: *IoT S&P'18*, Association for Computing Machinery, 2018, pp. 8–14.
- [30] G.n. Matthiasson, A. Giaretta, N. Dragoni, IoT device profiling: From MUD files to SxC contracts, in: *Open Identity Summit 2020*, in: *Lecture Notes in Informatics (LNI) - Proceedings*, vol. P-305, Gesellschaft für Informatik e.V., 2020, pp. 143–154.
- [31] F. Martinelli, I. Matteucci, P. Mori, A. Saracino, Concurrent history-based usage control policies, in: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELWARD, INSTICC, SciTePress*, 2017, pp. 657–666, <http://dx.doi.org/10.5220/0006232506570666>.
- [32] U. Schöpp, C. Xu, A. Ibrahim, F. Faghih, T. Dimitrakos, Specifying a usage control system, in: *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies, SACMAT '23*, Association for Computing Machinery, New York, NY, USA, 2023, pp. 193–200, <http://dx.doi.org/10.1145/3589608.3593843>.
- [33] A. Giaretta, N. Dragoni, F. Massacci, Protecting the internet of things with security-by-contract and fog computing, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), 2019, pp. 1–6, <http://dx.doi.org/10.1109/WF-IoT.2019.8767243>.
- [34] C.E. Rubio-Medrano, G.-J. Ahn, K. Sohr, Verifying access control properties with design by contract: Framework and lessons learned, in: 2013 IEEE 37th Annual Computer Software and Applications Conference, 2013, pp. 21–26, <http://dx.doi.org/10.1109/COMPSAC.2013.7>.
- [35] C.A.R. Hoare, An axiomatic basis for computer programming, *Commun. ACM* 12 (10) (1969) 576–580.
- [36] Y. Elrakaiy, F. Cuppens, N. Cuppens-Boulahia, Formal enforcement and management of obligation policies, *Data Knowl. Eng.* 71 (1) (2012) 127–147, <http://dx.doi.org/10.1016/j.datak.2011.09.001>.
- [37] V. Karimi, P. Alencar, D. Cowan, A formal modeling and analysis approach for access control rules, policies, and their combinations, *Int. J. Inf. Secur.* 16 (2017) 43–74, <http://dx.doi.org/10.1007/s10207-016-0314-4>.
- [38] M. Ben-Ari, *Principles of the Spin Model Checker*, Springer, 2008.
- [39] M. Langar, M. Mejri, K. Adi, Formal enforcement of security policies on concurrent systems, *J. Symbolic Comput.* 46 (9) (2011) 997–1016, <http://dx.doi.org/10.1016/j.jsc.2011.05.003>.
- [40] C.D.P.K. Ramli, H.R. Nielson, F. Nielson, The logic of XACML, *Sci. Comput. Program.* 83 (2014) 80–105, <http://dx.doi.org/10.1016/j.scico.2013.05.003>, Formal Aspects of Component Software (FACS 2011 selected & extended papers).
- [41] F. Turkmen, J. den Hartog, S. Ranise, N. Zannone, Formal analysis of XACML policies using SMT, *Comput. Secur.* 66 (2017) 185–203, <http://dx.doi.org/10.1016/j.cose.2017.01.009>.
- [42] M. Masi, R. Pugliese, F. Tiezzi, Formalisation and implementation of the XACML access control mechanism, in: G. Barthe, B. Livshits, R. Scandariato (Eds.), *Engineering Secure Software and Systems*, Springer Berlin Heidelberg, 2012, pp. 60–74.
- [43] A. Aldini, F. Martinelli, A. Saracino, D. Sgandurra, Detection of repackaged mobile applications through a collaborative approach, *Concurr. Comput.: Pr. Exp.* 27 (11) (2015) 2818–2838, <http://dx.doi.org/10.1002/cpe.3447>.