# A Methodology for the Experimental Performance Evaluation of Access Control Enforcement Mechanisms based on Business Processes

Stefano Berlato[a,b], Roberto Carbone[b], Silvio Ranise[c,b]

[a] *DIBRIS, University of Genova, Genoa, Italy*
[b] *Center for Cybersecurity, Fondazione Bruno Kessler, Trento, Italy*
[c] *Department of Mathematics, University of Trento, Trento, Italy*

**Abstract**

While the security analysis of Access Control (AC) policies has received a lot of attention, the same cannot be said for their enforcement. As applications become more distributed, centralized services a bottleneck, and legal compliance constraints stricter (e.g., the problem of honest but curious Cloud providers in the light of privacy regulations), the fine-tuning of AC enforcement mechanisms is likely to become more and more important. This is especially true in scenarios where the quality of service may suffer from computationally heavy security mechanisms and low latency is a prominent requirement. As a first step towards a principled approach to fine-tune AC enforcement mechanisms, this paper introduces a methodology providing the means to measure the performance of such mechanisms through the simulation of realistic scenarios. To do so, we base our methodology on Business Process Model and Notation (BPMN) workflows — that provide for an appropriate abstraction of the sequences of requests (e.g., access a resource, revoke a permission) sent toward AC enforcement mechanisms — to evaluate and compare the performance of different mechanisms. We implement our methodology and use it to evaluate three AC enforcement mechanisms representative of both traditional centralized — i.e., the Open Policy Agent (OPA) and the eXtensible Access Control Markup Language (XACML) — and decentralized AC — i.e., the `CryptoAC` tool.

*Keywords:* Access control, access control enforcement, BPMN, workflows, experimental evaluation, cryptographic algorithms

## 1. Introduction

Access Control (AC) is a fundamental component of any cyber-physical application [3]. Samarati and De Capitani di Vimercati [42] defined AC as "the process of mediating every request to resources maintained by an application and determining whether the request should be granted or denied" and divided it into three levels:

- *AC Policy* (or "policy"): this abstract level consists of statements declaring what users can perform what actions on what resources. The policy is usually defined by the owner of the resources or the administrator of the application — the administrator is a user with special privileges;
- *AC Model* (or "model"): this intermediate level is a formal representation of the policy giving the semantics for granting or denying users' requests. The model is usually chosen from a list of known and consolidated models, e.g., Role-Based Access Control (RBAC) [43], Attribute-Based Access Control (ABAC) [26];
- *AC Enforcement Mechanism* (or "mechanism"): this concrete level is the hardware and software enforcing the policy based on the chosen model. Hereafter, we consider software-only AC enforcement mechanisms. The concept of AC enforcement mechanism is equivalent to the concept of *AC mechanism* defined in the Special Publication (SP) 800-162 of the National Institute of Standards and Technology (NIST) [26] as the component that receives users' requests and produces (and sometimes enforces) an access decision.

As these levels are independent of each other, there may exist several mechanisms for the same model (and policy). This flexibility allows organizations to select the best mechanism for enforcing their policy under the chosen model, based on their requirements and those of the underlying scenario. For instance, a large international company operating in the healthcare sector may need a robust mechanism capable of enforcing AC policies following a rigid structure. Therefore, such a company may select one of the several implementations[1,2,3] of the eXtensible Access Control Markup Language (XACML) standard,[4] an OASIS standard comprising a language, a reference architecture and a dedicated protocol to express and evaluate AC policies based on an extension of the ABAC model. Differently, a small private clinic with no internal infrastructure may want to avoid the complexity of XACML and favor a more portable and easy-to-use AC enforcement mechanism. Therefore, such a clinic may choose the Open Policy Agent (OPA),[5] a

[1] `https://github.com/authzforce`
[2] `https://is.docs.wso2.com/en/latest/learn/access-control/`
[3] `https://axiomatics.com/`
[4] `https://docs.oasis-open.org/xacml/`
[5] `https://www.openpolicyagent.org/`

cloud native open-source AC enforcement mechanism providing a high-level declarative language (called Rego) to specify AC policies (both RBAC and ABAC). However, both XACML and OPA expect AC policies to be enforced in a centralized fashion, an approach which comes with both benefits (e.g., ease of deployment and maintainability) but also drawbacks. For example, the use of XACML by a large number of users in the aforementioned international company scenario could lead to a degradation of the user experience due to the intrinsically limited scalability of centralized services and network saturation [1]. Similarly, the offloading of AC policy decisions to OPA when operated by (honest but curious) Cloud providers could threaten the privacy of the clinic patients' data [5]. Therefore, both the company and the clinic may consider the use of Cryptographic Access Control (CAC) [7, 21], which employs cryptography as a means to guarantee confidentiality and integrity of sensitive resources while enforcing AC policies in a decentralized fashion, hence being suitable for scenarios lacking a fully-trusted central entity [5]. In CAC, data are encrypted, and the permission to access the encrypted data is embodied by the secret decryption keys, which are distributed to authorized users only by a single administrator. Hence, in CAC the administration of the policy is still centralized — avoiding synchronization and consistency issues — while the enforcement is pre-compiled and embedded into the (distribution of the) secret decryption keys — solving the problems affecting centralized services. On the other hand, CAC might incur non-negligible overhead due to the cryptographic computations involved [6], an aspect that an organization should carefully assess. Generalizing, an organization may need to compare different mechanisms through the evaluation of *subjective* conditions such as commercial and open-source license terms, monetary costs, integration with existing software — which, by definition, cannot be measured easily — and *objective* metrics such as scalability, response time and throughput. These 3 metrics — grouped under the term "performance" hereafter — are fundamental to evaluate the suitability of a mechanism in a specific scenario. This is especially true in emerging application fields such as Intelligent Transportation System (ITS) where, e.g., low latency is often a strict requirement [13] and, in general, any Internet of Things (IoT) scenario where, due to the presence of constrained devices and unreliable networks, the quality of service may suffer from computationally demanding or network-heavy mechanisms. Another crucial aspect for evaluating mechanisms is assessing their logical security — that is, the security of the AC model they implement (e.g., in terms of soundness). We highlight that this aspect is out of the scope of our work, as we focus on the enforcement level of AC; the interested reader may refer to the rich literature already existing on the topic of AC model security (e.g., see [31, 18, 54]).

### *1.1. Performance Evaluation of Access Control Mechanisms*

Performance evaluation is therefore a fundamental step to compare mechanisms and assess their suitability for a given scenario. Unfortunately, mechanisms both commercially available and proposed in the literature often lack a performance evaluation, or they just present a theoretical analysis without running any concrete experiment (see Section 2.2). In some cases, the performance of these mechanisms is measured through synthetic micro-benchmarks, which consist in making mechanisms evaluate single requests (e.g., read a resource) repeatedly, often while ranging, e.g., the number of roles (e.g., [6]) or attributes (e.g., [38, 34]) in the AC policy. Simply put, micro-benchmarking evaluates a mechanism focusing on its functionalities, i.e., from the developers' point of view. However, in this work we argue that micro-benchmarks are not representative of a realistic use of AC enforcement mechanisms, as they cannot assess the overall user experience in the use of the application, i.e., from the users' point of view. Instead, we believe that a more realistic approach would be to evaluate the performance of a mechanism under a specific sequence of requests derived from a realistic (business) process. Moreover, this sequence should include both users' (e.g., access a resource) and administrative (e.g., assign a permission) requests. The rationale is that organizations use AC enforcement mechanisms to evaluate users' and administrative requests while performing a specific business process. For instance, the request of a warehouse worker to read the store inventory database (e.g., to check the availability of a product) could be part of a (much larger) order fulfillment process which includes several other activities (e.g., remove an article from the catalog, create an order procurement request, send an invoice to the employees of the billing office, update the catalog with a new article). Therefore, while a micro-benchmark would evaluate the performance of an AC enforcement mechanism in relation to, e.g., the store inventory read activity alone, a more comprehensive evaluation would consider the whole order fulfillment process. Even more realistically, the performance of a mechanism should be evaluated through the concurrent and intertwined execution of more instances of different business processes of the same organization (e.g., order fulfillment, shipment, payment) to simulate a representative scenario. Indeed, we claim (as we later empirically show in Section 7.4) that the usage of an AC enforcement mechanism — and, consequently, its performance — largely depends on the business processes defining the kind and number of requests sent to the mechanism, i.e., on the specific scenario; unfortunately, micro-benchmarking cannot capture this fundamental aspect, as we further discuss in Sections 4.3 and 7.4. While it is reasonable that no mechanism is better than the others in all scenarios — and that performance should be carefully evaluated on a case-by-case basis — to the best of our knowledge, no alternative to micro-benchmarking is available to evaluate the performance of AC enforcement mechanisms.

### *1.2. A Methodology for Realistic Performance Evaluation*

In this paper — which derives from research conducted in Berlato's PhD thesis [4] — we propose a methodology allowing organizations to evaluate and compare the performance (i.e., scalability, response time, and throughput) of AC enforcement mechanisms for their scenarios. We report a graphical representation of our methodology in Figure 1.[6] To ensure a realistic

[6] Icons by SBTS2018, mj, IconHome, Vignesh Oviyan, Freepik from `flaticon.com`

evaluation, we rely on Business Process Model and Notation (BPMN), which is the standard most widely adopted by organizations to carry out business processes modeled as *workflows* [8, 40, 33] — a workflow is, essentially, a series of activities. In case an organization does not use BPMN, we remark that there is an extensive body of literature on process mining for BPMN — that is, extracting BPMN workflows, usually from event logs (e.g., see [29]). Besides traditional business processes, we note that BPMN is also used to model (even automated) processes in emerging application fields (e.g., IoT-based scenarios) [35, 28].

In detail, our contributions can be described as follows:

- we propose a procedure called AC state-change rule extraction procedurE (ACE) to derive realistic sequences of requests (e.g., revoke a permission, access a resource) from workflows automatically. Following a well established formal semantics (see [15]), ACE maps an input workflow specified in BPMN to a particular class of Petri nets, called WorkFlow (WF) nets (step 1 in Figure 1). Then, ACE computes the topological ordering of the WF net — represented as a graph — to find all executions, i.e., all possible ways in which it is possible to achieve the business goal modeled in the corresponding workflow (step 2). Finally, ACE decorates the executions with relevant information extracted from the workflow (step 3) to then derive sequences of requests (step 4);

- we design a simulator tool called Access Control Mechanisms Evaluator (ACME) to evaluate and compare the performance of AC enforcement mechanisms on the sequences of requests derived by ACE. More precisely, ACME allows an organization to specify which types of these requests to consider during the performance evaluation (i.e., the *Types of Requests* in Figure 1); for instance, an organization may be interested in evaluating a mechanism based on users' but not administrative requests. ACME is composed of two modules, i.e., the initializer and the engine. The initializer allows setting the initial AC policy state (step 5) — where the AC policy state (or just "state") defines the elements present in the policy (e.g., users, roles, attributes, resources, permissions) — by combining the output of ACE and the types of requests chosen by the organization with a *State Blueprint* (we explain the concept of blueprint in detail in Section 3.3.1). Alternatively from what is represented in Figure 1, the state can also be directly specified by organizations as representative of their internal structure — e.g., number of employees and qualifications. Then, the engine executes the sequences of requests (limited only to those types specified by the organization) derived by ACE — reflecting the execution of workflows — in the given AC policy state — determined by either the initializer or the organization — to evaluate the performance of the mechanism under evaluation (step 6). The engine also supports the concurrent and intertwined execution of more sequences of requests corresponding to different business processes;

- to demonstrate that our methodology — composed of ACE and ACME — provides a robust basis for evaluating and comparing the performance of AC enforcement mechanisms realistically, we use it to evaluate three mechanisms: the OPA and the AuthzForce Server (Community Edition) open-source implementation of XACML — representative of traditional centralized AC — and the `CryptoAC`[7] [6] implementation of CAC — representative of decentralized AC. We also compare the results of our methodology against those obtained from micro-benchmarks, confirming our claim that the latter may not always suggest the best mechanism for a given scenario.

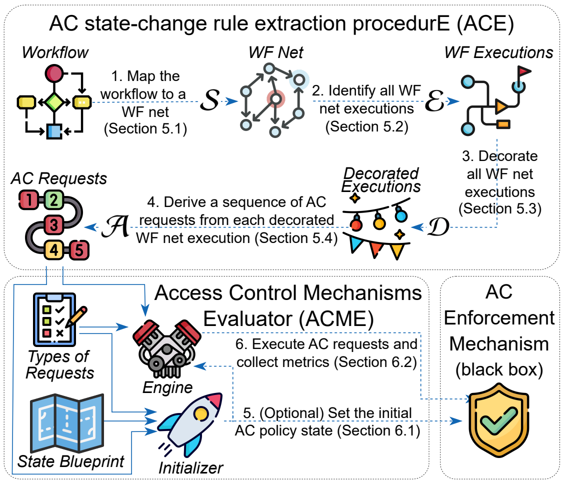


Figure 1: High-Level Graphical Representation of our Methodology

The rest of the paper is organized as follows. In Section 2, we discuss related work, while in Section 3 we introduce the background. In Section 4, we give an overview of our methodology, and present ACE and ACME in detail in Sections 5 and 6, respectively. Then, we use our methodology to evaluate OPA, XACML and `CryptoAC`, and discuss the obtained results, comparing them against micro-benchmarking in Section 7. We conclude the paper with final remarks and future work in Section 8.

## 2. Related Work

Below, we first review proposals in the literature to infer information concerning AC policies and requests from workflows modeling business processes (Section 2.1) — thus, related to ACE. Then, we investigate how performance evaluation is conducted in works presenting novel AC enforcement mechanisms and analyze the solutions proposed to evaluate the performance of such mechanisms (Section 2.2) — thus, related to ACME.

### 2.1. Infer Access Control Information from Workflows

Domingos et al. [16] propose a technique to infer (parts of) RBAC policies from workflows represented as Unified Model-

[7] `https://cryptoac.readthedocs.io/`

ing Language (UML) activity diagrams. The authors identify roles by assuming the presence of UML supply objects defining what qualification (i.e., role) is required to perform a certain activity. Consequently, each role has permission to carry out the activity to which it is connected through the UML supply object. Regarding resources, the authors assume that all activities have generic input and output by default. The authors in [53] propose a model to express RBAC policies in workflows. The model takes as input a workflow encoded as a partially ordered set of activities and a set of roles with permission to carry them out. Each activity (e.g., prepare a purchase order) already includes what information is required (e.g., amount of items and price per item) and what is produced (e.g., expense report). In other words, the authors assume to receive as input the roles and the resources, from which they infer the corresponding permissions. In [20], the authors propose an enhanced RBAC model specific for business workflows. Their approach takes as input a workflow represented as a UML-like activity diagram, along with a textual description of the expected activities. The authors infer RBAC policies manually, extracting roles from the description (i.e., the semantics) of the activities. At the same time, they consider the activities themselves to be the "resources" over which to distribute permissions. Also Uddin et al. [48] propose an enhanced RBAC model for workflows. As the focus of the authors is on the dynamicity of the authorizations and not on deriving AC information, they consider a single workflow and manually extract roles, permissions and resources from the semantics of the activities. Then, they complement the RBAC policy by deciding what and how many users are present and the assignments of users to roles. The authors expect their policy to be enforced by a XACML-based enforcement mechanism.

Summarizing, all these approaches expect workflows to be represented either as UML activity diagrams or in a custom format, while in our work we address workflows in BPMN — the standard most widely adopted for modeling business processes [8, 40, 33]. Moreover, these approaches expect (part of) AC information to be either provided as straight input (i.e., [53]) or expressed through ad-hoc artifacts (e.g., UML supply objects in [16]) or embedded in the textual description of the activities [20, 48]. Instead, ACE infers information concerning AC policies and requests from workflows automatically based on their syntax and that of BPMN symbols. Moreover, ACE also enumerates all possible executions of workflows with the goal of benchmarking AC enforcement mechanisms.

### 2.2. *Performance Evaluation of Access Control Mechanisms*

Researchers usually measure the performance of AC enforcement mechanisms with micro-benchmarks which, however, are not representative of a realistic scenario, although some works propose techniques to evaluate (more or less generic) mechanisms. Below, we discuss the most relevant of these works.

#### 2.2.1. *Micro-benchmarks*

In [38], the authors propose an AC enforcement mechanism based on XACML for smart energy grids to protect safety-relevant settings from either (unauthorized) malicious or (authorized) accidental changes. The authors measure the average response time of their mechanism for 1,000 requests to change safety-relevant settings with different combinations of CPUs and RAM, varying the AC policy by considering two sizes of attribute sets (10 and 100 attributes). Martinelli et al. [34] propose a mechanism for OPC-UA-based industrial control applications allowing to define complex AC policies with continuous evaluation of requests, i.e., with the possibility of stopping previously authorized actions when the policy conditions are not satisfied anymore. To measure the response time of their mechanism, the authors put a bound on the computational resources available, and then run experiments while varying the number of attributes in the policy. In [6], the authors propose the `CryptoAC` tool to enforce RBAC policies through cryptography. The authors measure the response time of `CryptoAC` by invoking all `CryptoAC`'s APIs separately while ranging the number of roles, resources, and assigned permissions. Ahmad et al. [1] investigate Cloud- and Edge-based ABAC enforcement mechanisms for distributed applications (e.g., smart home). Then, the authors propose a novel mechanism and evaluate its response time in Amazon Web Services (AWS) through micro-benchmarks, measuring the impact of Cloud vs. Edge deployment of the mechanism along with different attribute storage and retrieval strategies.

Summarizing, we find that the performance of AC enforcement mechanisms proposed in the literature is usually (either not measured or) evaluated through micro-benchmarks, i.e., by measuring the response time of single requests while varying specific elements of the AC policy like the number of roles (e.g., [6]) or attributes (e.g., [38, 34]). Moreover, the response time is often the only metric evaluated, while scalability and throughput are not considered. ACME, instead, aims at measuring the performance (i.e., scalability, response time, and throughput) of a mechanism throughout the intertwined execution of more instances of several workflows representative of the business processes of a specific scenario, thus considering the overall user experience from the users' point of view rather than focusing on single functionalities of the mechanism from the developers' point of view (as also discussed in Section 1.1).

#### 2.2.2. *Dedicated Techniques*

In [47], the authors evaluate the response time of 3 open-source XACML implementations with different combinations of XACML policies. In the experiments, the authors send single users' requests and measure the loading and evaluation time of the XACML implementations. Ilhan et al. [27] study possible optimizations for caching in XACML. Similarly to [47], the authors measure the time for loading, filtering, and evaluating policies for a XACML-based mechanism of their choice. The authors consider 4 combinations of policies and measure the response time and the RAM consumption of the mechanism with single users' requests. In [9] (and earlier work [10, 11, 22]), the authors present a framework, `ATLAS`, to measure the response time of AC enforcement mechanisms. First, they generate large numbers of XACML-based policies representative of a scenario, starting from a set of (manually) specified domain

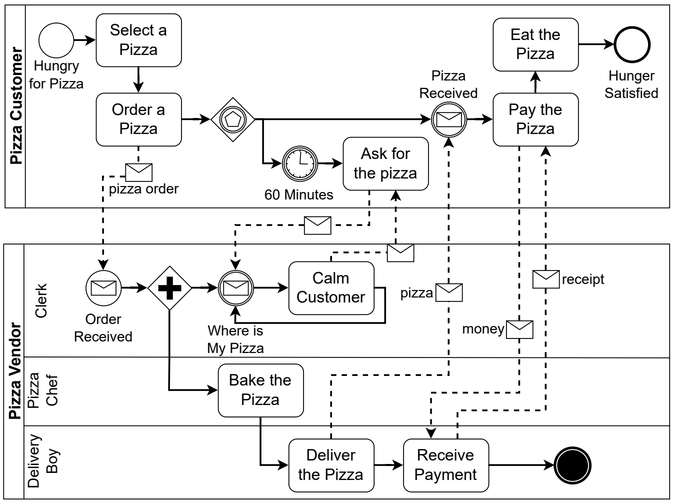


Figure 2: BPMN Workflow - The Pizza Collaboration

models and templates. Then, they evaluate the generated policies by sending requests derived from the policies themselves to XACML-based AC enforcement mechanisms. In `ATLAS`, each measurement is associated with predetermined settings, such as the policy set, the (single) users' request, and the computational resources available (i.e., CPU and RAM). In line with our thinking, the authors underline that performance evaluations must be based on realistic conditions to obtain significant results. However, the authors tackle this issue from the point of view of the policies (whose generation requires manual effort) but not of the realistic sequence of requests, as we do instead. Similarly to [9], also the authors in [2] address the generation of realistic (XACML) policies. To this aim, they propose `XACBench`, a tool to generate synthetic XACML policies by extracting, modeling, and generalizing statistical properties of a given input policy. `XACBench` allows generating XACML policies of any size that model the characteristics of real-world policy sets.

Summarizing, the available literature for performance evaluation of AC enforcement mechanisms either targets specific standards (e.g., XACML in [47, 27]), thus restricting the applicability of the proposed approaches, or focus on the generation of realistic AC policies (e.g., in [9, 2]). Instead, the generation of realistic sequences of requests to measure the performance of generic mechanisms is, to the best of our knowledge, still unaddressed. Finally, all the proposed approaches focus on the evaluation of standalone single users' requests (i.e., evaluating whether a request of a user to perform a particular action on a resource should be granted or denied), while ACME evaluates (sequences of) both users' and administrative requests.

## 3. Background

Below, we provide background information on BPMN and WF nets (Sections 3.1 and 3.2). Then, we introduce relevant AC concepts and present state blueprints — later used in our evaluation — derived from real-world datasets (Section 3.3).

### *3.1. Business Process Model and Notation*

BPMN is an Object Management Group (OMG) standard[8] to model business processes intuitively through workflows readily understandable by all stakeholders (e.g., managers, developers). BPMN uses a common set of symbols meticulously described in the standard [24]; the unambiguousness of these symbols and their independence of any particular implementation environment enable organizations to employ workflows developed in different scenarios to achieve the same business goals. Similarly to programming patterns, workflows can be reused to implement recurring business processes (e.g., project administration, HR management), reducing costs and improving efficiency. Consequently, organizations often reuse consolidated and well-established BPMN workflows instead of developing new ones from scratch. For instance, the ITSM Process Library is a commercial collection of BPMN workflows "for sustainable and value-creating process-oriented IT operations".[9]

#### *3.1.1. BPMN Symbols*

A workflow represents a business process through (several instances of) 3 kinds of *flow objects*:

- *events*: represented as circles, events are triggers that start (circle with thin line), occur during (circle with double line) or end (circle with thick line) a workflow;
- *activities*: represented as (usually rounded) rectangles, activities are tasks to be performed during the execution of the workflow;
- *gateways*: represented as diamonds, gateways are decision points defining conditional behavior to regulate the flow of activities and events. A gateway can be, e.g., exclusive ("x"), parallel ("+"), inclusive ("o") or event-based ("⬠").

Two flow objects can then be linked together with *connecting objects*, the most common of which are:

- *sequences*: represented with solid arrows, sequence connecting objects express the order in which activities are performed;
- *messages*: represented with dashed arrows, message connecting objects represent the exchange of information between two participants;
- *data associations*: represented with dotted arrows, data association connecting objects put data elements — such as data stores and data objects — in relation to activities.

In other words, sequences establish dependencies among flow objects, while data stores, data objects and messages define the flow of information. Messages and data objects are transient (i.e., they exist only during the execution of the workflow), while data stores are persistent (i.e., they already exist before the execution of the workflow and continue to exist after

[8] `https://www.omg.org/spec/BPMN/`

[9] `https://www.signavio.com/reference-models/itsm-process-library/`

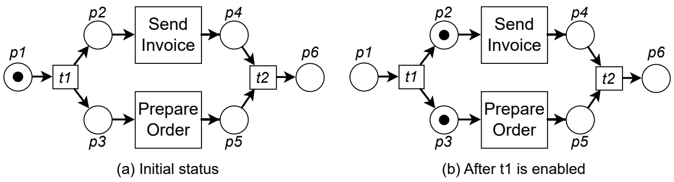


Figure 3: Sample WF Net

the execution of the workflow). Flow and connecting objects are often grouped in a *lane* representing an active agent (either a user or software) in charge of carrying out a portion of the workflow. More lanes (e.g., different agents) may be logically grouped under a *pool* (e.g., an organization). Finally, workflows are usually encoded in XML documents. For more details on BPMN, we refer the interested reader to [24].

### *3.1.2. The Pizza Collaboration Workflow*

We report in Figure 2 an example workflow from an official OMG report [23]. The workflow models the interaction between a pizza vendor and a pizza customer. The entry point is the "Hungry for Pizza" event, which leads the pizza customer to select and order a pizza with the "Select a pizza" and "Order a pizza" activities, respectively. Afterward, the pizza customer reaches an event-based gateway; this gateway indicates that the pizza customer waits for one of two events that could happen next: either the pizza is received or there is no delivery within 60 minutes, i.e., after one hour the customer calls the pizza vendor and asks for the pizza. In response to this call, the clerk promises that the pizza will be delivered soon, and the customer goes back to waiting. After the "Order a pizza" activity, a message "pizza order" is sent toward the start event "Order Received" activating the "Clerk" lane. The parallel gateway splits the process into concurrent activities, one answering the pizza customer's call and the other one reaching the "Pizza Chef" lane. After the chef baked the pizza, the delivery boy delivers the pizza and receives the payment, which includes giving a receipt to the customer, and reaches an intermediate end event (bold circle with a black dot). Finally, the customer can eat her pizza, reaching the end event (circle with a thick line).

## *3.2. Petri and Workflow Nets*

Petri nets are widely used to model the flow of objects or information in (concurrent) applications [15]. A Petri net is a tuple $\langle P, T, F\rangle$ representing a (possibly cyclic) directed graph whose vertices correspond to a set of places $P$ (represented as circles) and a set of transitions $T$ (represented as rectangles), while edges correspond to a set of arcs $F \subseteq (P \cup T) \times (P \cup T)$. Petri nets are bipartite graphs, meaning that edges connect places to transitions and vice-versa only, i.e., $F \cap (P \times P) = F \cap (T \times T) = \emptyset$. In other words, a transition is preceded by one or more input places and followed by one or more output places; formally, a transition $t \in T$ has a pre-set $^{\bullet}t = \{p \mid (p, t) \in F\}$ and a post-set $t^{\bullet} = \{p \mid (t, p) \in F\}$. A place can contain one or more tokens (represented as full black circles) modeling the flow of objects or information. A marking $m$ is a mapping from $P$ to the set of natural numbers $\mathbb{N}$, i.e., defining how many tokens each place of the Petri net contains. A transition $t \in T$ can be enabled (i.e., visited) only if there is at least one token in each of its input places, i.e., $m(p) > 0\ \forall p \in {}^{\bullet}t$ (e.g., *t1* in Figure 3a). When a transition $t \in T$ is enabled in a marking $m$ — denoted with $m[t\rangle$ — we remove one token from each $p \in^{\bullet} t$ and add one token to each $p \in t^{\bullet}$ (see Figures 3a and 3b) deriving a marking $m'$ such that:

$$m'(p) = \begin{cases} m(p) - 1 & \text{if } p \in {}^{\bullet}t \\ m(p) + 1 & \text{if } p \in t^{\bullet} \\ m(p) & \text{otherwise} \end{cases} \tag{1}$$

We denote with $[m\rangle$ the set of all markings that can be recursively (i.e., not only immediately) derived starting from $m$.

WF nets (see Figure 3) are a subclass of Petri nets typically used to model control — as well as data [50] — flow in (e.g., BPMN) business processes modeled as workflows. Differently from (more generic) Petri nets, WF nets always have a unique source place $p_{source}$ — i.e., that is not the target of any edge — and a unique sink place $p_{sink}$ — i.e., that is not the source of any edge. Moreover, in a WF net, every place and transition is on (at least one) directed path from the source to the sink place. The marking $m_0$ of a WF net $\langle P, T, F\rangle$, which assigns one token to the source place and zero tokens to all other places — i.e., $m_0(p_{source}) = 1 \wedge m_0(p) = 0\ \forall p \in P \setminus \{p_{source}\}$ — is called the *canonical marking*. Conversely, the marking $m_s$ assigns one token to the sink place and zero tokens to all other places — i.e., $m_s(p_{sink}) = 1 \wedge m_s(p) = 0\ \forall p \in P \setminus \{p_{sink}\}$. WF nets with the canonical marking are *1-safe*, meaning that there cannot be more than one token in a given place at a time [15]. However, WF nets may have multiple transitions that can be enabled simultaneously (e.g., "Send Invoice" and "Prepare Order" in Figure 3b); in this case, the choice of which transition to enable first is nondeterministic. Given a WF net $\langle P, T, F\rangle$ with canonical marking $m_0$, we define as a *WF net execution* a sequence of transitions $x = \langle t_1, ..., t_n\rangle \mid (t \in T\ \forall t \in x)$ enabled by following the aforementioned marking conditions — that is, $m_0[t_1\rangle \wedge (m'[t_i\rangle \implies m' \in [m\rangle \wedge m[t_{i-1}\rangle\ \forall t_i \in x \setminus \{t_1\})$.

*Soundness.* A WF net $\langle P, T, F\rangle$ with canonical marking $m_0$ satisfies the *soundness* criterion if it has the following two properties [49, 45]:

- *proper termination*: the workflow represented by the WF net can terminate for any case. In other words, for every marking $m$ derived from the canonical marking $m_0$, there exists a transition enabling sequence leading from marking $m$ to marking $m_s$; formally, $m_s \in [m\rangle\ \forall m \in [m_0\rangle$. Moreover, $m_s$ is the only marking that can be derived from $m_0$ that assigns one token to the sink place. Formally, $m \in [m_0\rangle \wedge m(p_{sink}) = 1 \implies m = m_s$;
- *quasi-liveness*: the WF net does not have any *dead transition*. In other words, any transition can possibly be enabled; formally, $\exists m \in [m_0\rangle \mid m[t\rangle\ \forall t \in T$.

A WF net having the *proper termination* property — but not the quasi-liveness property — is said to satisfy the *weak* *soundness* criterion [45]. Intuitively, the soundness criterion implies the weak soundness criterion. A WF net satisfying weak soundness is guaranteed to always have at least one *complete WF net execution* $x = \langle t_1, ..., t_n \rangle$, that is, a WF net execution where exactly one token reaches the sink place; formally, $m[t_n\rangle \implies [m\rangle = \{m_s\}$. A complete WF net execution reflects the execution of the workflow modeled by the WF net; we discuss WF net executions in more detail in Section 5.2.

### 3.3. Access Control Concepts

As in [31], we define an *AC scheme* as a state transition system $\langle \Gamma, Q, \vdash, \mathbf{\Psi} \rangle$ where $\Gamma$ is a set of states, $Q$ is a set of queries, $\mathbf{\Psi}$ is a set of state-change rules and $\vdash$: $\Gamma \times Q \rightarrow \{true, false\}$ is the entailment relation determining whether a query is true in a given state. We can instantiate an AC scheme into an *AC system* which consists of a pair $\langle \gamma, \Psi \rangle$ where $\gamma \in \Gamma$ and $\Psi \subseteq \mathbf{\Psi}$. Given an AC system $\langle \gamma, \Psi \rangle$, a sequence of state-change rules $\langle \psi_1, ..., \psi_{n-1} \rangle \mid \psi_i \in \Psi$ for $0 < i < n$ induces an *AC execution* $\langle \gamma_1, \psi_1, \gamma_2, ..., \gamma_{n-1}, \psi_{n-1}, \gamma_n \rangle$ where $\gamma_1 = \gamma$ and $\gamma_i \mapsto_{\psi_i} \gamma_{i+1}$ — i.e., $\psi_i$ transforms $\gamma_i$ into $\gamma_{i+1}$ — for $0 < i < n$.

In the rest of the paper, we consider policies specified in RBAC because of its wide adoption in both academia and industry [12] and the fact that business workflows are usually complemented with RBAC policies (see Section 2.1). As such, we now describe an AC scheme for RBAC. In RBAC, a state $\gamma \in \Gamma$ is described as a tuple $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$, where $\mathbf{U}$ is the set of users, $\mathbf{R}$ is the set of roles, $\mathbf{P}$ is the set of resources, $\mathbf{UR} \subseteq \mathbf{U} \times \mathbf{R}$ is the set of user-role assignments and $\mathbf{PA} \subseteq \mathbf{R} \times \mathbf{PR}$ is the set of role-permission assignments, being $\mathbf{PR} \subseteq \mathbf{P} \times \mathbf{OP}$ a derivative set of $\mathbf{P}$ combined with a fixed set of actions $\mathbf{OP}$ (e.g., read, write). Both $\mathbf{OP}$ and $\mathbf{PR}$ are not part of the state, as $\mathbf{OP}$ remains constant over time and $\mathbf{PR}$ is derivative of $\mathbf{P}$ and $\mathbf{OP}$. We note that role hierarchies can always be compiled away by adding suitable pairs to $\mathbf{UR}$. The set of all queries is $Q = \mathbf{U} \times \mathbf{PR}$, thus a query $q = \langle u, p \rangle$ means whether the user $u \in \mathbf{U}$ can use the permission $p \in \mathbf{PR}$, where $p = \langle f, op \rangle$ with $f \in \mathbf{P}$ and $op \in \mathbf{OP}$. We note that, when considering $\mathbf{OP} = \{$`read`, `write`$\}$, there exist two types of queries, i.e., to read and write over a resource, respectively. We report the complete set $\mathbf{\Psi}$ of state-change rules for RBAC — according to the NIST [19] — and the resulting states in Table 1. For the sake of simplicity, we consider the entailment relation $\gamma \vdash (\langle u, p \rangle)$ as a special state-change rule for RBAC that does not transform the state $\gamma$ to which it is applied — i.e., $\gamma \mapsto_{\gamma \vdash (\langle u,p \rangle)} \gamma$ — but has the side effect of evaluating the query $\langle u, p \rangle$. In other words, the entailment corresponds to users' requests, while the other state-change rules for RBAC correspond to administrative requests. Hereafter, when mentioning state-change rules, we refer to state-change rules for RBAC only.

Table 1: The Set $\mathbf{\Psi}$ of State-Change Rules for a Generic RBAC Scheme [19]

| State-Change Rule $\psi_i$ | Resulting AC Policy State $\gamma_{i+1}$ |
|---|---|
| addUser($u$) | $\langle \mathbf{U} \cup \{u\}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| deleteUser($u$) | $\langle \mathbf{U} \setminus \{u\}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| addRole($r$) | $\langle \mathbf{U}, \mathbf{R} \cup \{r\}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| deleteRole($r$) | $\langle \mathbf{U}, \mathbf{R} \setminus \{r\}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| addResource($f$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P} \cup \{f\}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| deleteResource($f$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P} \setminus \{f\}, \mathbf{UR}, \mathbf{PA} \rangle$ |
| assignUserToRole($u, r$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR} \cup \{(u, r)\}, \mathbf{PA} \rangle$ |
| revokeUserFromRole($u, r$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR} \setminus \{(u, r)\}, \mathbf{PA} \rangle$ |
| assignPermissionToRole($r, p$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \cup \{(r, p)\} \rangle$ |
| revokePermissionFromRole($r, p$) | $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \setminus \{(r, p)\} \rangle$ |
| $\gamma \vdash (\langle u, p \rangle) : true \iff \exists r \in \mathbf{R} \mid (u, r) \in \mathbf{UR} \wedge (r, p) \in \mathbf{PA}$ | |

#### 3.3.1. Role-based Access Control Policy State Blueprints

A *state blueprint* for RBAC consists of a tuple $\langle |\mathbf{U}|, |\mathbf{R}|, |\mathbf{P}|, |\mathbf{UR}|, |\mathbf{PA}| \rangle$ — expressing the cardinality of each set of an RBAC state — along with a set of integers representing the maximum ($M$) and minimum ($m$) number — but not the identifiers — of roles assigned to each user ($r \rightarrow u$), users assigned to each role ($u \rightarrow r$), permissions assigned to each role ($p \rightarrow r$) and roles assigned to each permission ($r \rightarrow p$). Ene et al. [17] present some RBAC state blueprints mined from real-world datasets which we report in Table 2. For instance, the *healthcare* state blueprint expects an RBAC policy with 46 users, 13 roles, and 46 resources, where (in total) there are 55 assignments between users and roles and 359 assignments between roles and permissions. However, each user can be assigned to at least 1 role and at most 5 roles (i.e., $r \rightarrow u$), while each role can be assigned to at least 1 user and at most 17 users (i.e., $u \rightarrow r$). Finally, each role can be assigned to at least 7 permissions and at most 45 permissions (i.e., $p \rightarrow r$), while each permission can be assigned to at least 1 role and at most 12 roles (i.e., $r \rightarrow p$). We show how a state blueprint can be used to generate a state in $\Gamma$ in Section 6.1.

#### 3.3.2. Functional Points in Access Control

We frame our methodology within the theoretical foundations outlined in the seminal work of [42] and the well-established NIST SP 800-162 [26].[10] AC enforcement mechanisms [42] — which are equivalent to the concept of AC mechanisms [26] (see Section 1)— are a central subject of our work. However, we note that the exact nature of a mechanism may vary depending on the supported AC models (e.g., RBAC vs. ABAC) and the developers' design choices concerning the (configuration and deployment of the) hardware and the software composing the mechanism. In this regard, it is important to report the concept of functional point as defined in

[10] While the well-known RFC 2904 ("AAA Authorization Framework") [44] provides a relevant perspective on AC, it was published in August 2000 and has since been largely superseded by the more comprehensive guidance in NIST SP 800-162.

Table 2: RBAC Policy State Blueprints from [17]

| State | Set | | | | | $r \rightarrow u$ | | $u \rightarrow r$ | | $p \rightarrow r$ | | $r \rightarrow p$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Blueprint | $\lvert U \rvert$ | $\lvert R \rvert$ | $\lvert P \rvert$ | $\lvert UR \rvert$ | $\lvert PA \rvert$ | $M$ | $m$ | $M$ | $m$ | $M$ | $m$ | $M$ | $m$ |
| domino | 79 | 20 | 231 | 75 | 629 | 3 | 0 | 30 | 1 | 209 | 1 | 10 | 1 |
| emea | 35 | 34 | 3,046 | 35 | 7,211 | 1 | 1 | 2 | 1 | 554 | 9 | 31 | 1 |
| firewall1 | 365 | 60 | 709 | 1,130 | 3,455 | 14 | 0 | 174 | 1 | 617 | 1 | 25 | 1 |
| firewall2 | 325 | 10 | 590 | 325 | 1,136 | 1 | 1 | 222 | 1 | 590 | 6 | 8 | 1 |
| healthcare | 46 | 13 | 46 | 55 | 359 | 5 | 1 | 17 | 1 | 45 | 7 | 12 | 1 |
| university | 493 | 16 | 56 | 495 | 202 | 2 | 1 | 288 | 1 | 40 | 2 | 12 | 1 |

[26]. In detail, a functional point represents a logical function or component of a mechanism that handles specific aspects of policy management and enforcement. The NIST SP 800-162 [26] identifies four functional points:

- the *Policy Decision Point (PDP)*, which produces AC decisions by evaluating users' requests (i.e., the entailment relation) against the policy;
- the *Policy Enforcement Point (PEP)*, which protects resources maintained by an application by enforcing AC decisions produced by the PDP;
- the *Policy Information Point (PIP)*, which serves as the retrieval source for data required by the PDP for evaluating users' requests;
- the *Policy Administration Point (PAP)*, which allows for managing (e.g., creating and modifying) AC policies by evaluating administrative requests (i.e., state-change rules other than the entailment relation).

In Section 7, we consider three mechanisms: the OPA, the AuthzForce Server (Community Edition) open-source implementation of XACML, and `CryptoAC`. The AuthzForce Server (Community Edition)[11] primarily functions as a PDP while also offering PAP capabilities, but it does not encompass the functions of a PEP or a PIP: these components need to be implemented or integrated separately. More precisely, in our context the PIP is not strictly necessary, as we later argue in Section 6.3. Similarly, OPA functions as both a PDP and a PAP, enabling dynamic updates and versioning of policies. On the other hand, OPA does not encompass the functions of a PEP or a PIP. Again, in our context, the PIP is not strictly necessary. Finally, `CryptoAC` functions as both PAP and PIP. Being a decentralized AC enforcement mechanism, `CryptoAC` also functions as a PDP and PEP in the sense that the evaluation of the entailment relation is implicit and the enforcement is pre-compiled and embedded into the (distribution of the) secret decryption keys.

## 4. Methodology Overview

We now give an overview of our methodology (see Figure 1) which consists of a procedure, ACE, to derive representative sequences of requests from workflows (Section 4.1), and a simulator tool, ACME, using these sequences to evaluate the performance of AC enforcement mechanisms (Section 4.2). Below, we keep the discussion at a high level to allow the reader to get a general understanding of our methodology before delving into (more complex) details; we provide a meticulous description of ACE and ACME in Sections 5 and 6, respectively. Finally, we discuss the main issues affecting experimental evaluations using micro-benchmarks to (try to) measure the performance of AC enforcement mechanisms realistically and argue how our methodology solves these issues (Section 4.3). Throughout this section, we formulate 5 assumptions (*A*) which we group and report in Table 3 for ease of reference; we discuss each assumption as soon as we introduce it in the text below. While *A1*, *A2*, and *A5* are key for our methodology, it is possible to circumvent *A3* and *A4* if needed, as we later explain in Section 5.4.

In Table 4, we provide a conceptual map for our methodology as a sort of reading key of this work. In detail, for each step in our methodology, the conceptual map in Table 4 reports where that step is described (at high-level, informally, and in detail), where that step is represented as pseudocode in an algorithm, and what are the concepts involved in that step — as well as where each concept is first introduced or defined.

### 4.1. ACE Overview

As shown in Figure 1, ACE works in 4 steps: map to WF nets, identify WF net executions, decorate WF net executions and derive requests. Below, we briefly present each step, using the workflow in Figure 2 as a running example by reporting in Figure 4 the corresponding transformations.

#### 4.1.1. Map to Workflow Nets - $\mathcal{S}$

Given as input a BPMN-compliant workflow describing a business process (*A1*), we derive a WF net by relying on a well-established stream of work proposing a formal semantics — as pioneered by [15] — whereby flow objects (e.g., activities, gateways) and connecting objects (e.g., sequences, messages, and data associations) of the workflow are mapped to vertices (i.e., transitions and places) and edges of the WF net (see $\mathcal{S}$ in Figure 4). We assume the input workflow to not contain any loop (*A2*) as, intuitively, the sequences of requests we derive need to be finite to be used by ACME; noncyclicality of workflows can be achieved by design or, e.g., by pre-processing workflows to unroll loops by repeating them a specific number of times. The output of this step is therefore an acyclic WF net $\langle P, T, F\rangle$. We discuss the mapping in more detail in Section 5.1.

#### 4.1.2. Identify Workflow Net Executions - $\mathcal{E}$

Given as input an acyclic WF net, we compute its topological ordering to identify all WF net executions (see the definition in Section 3.2) which express all possible ways to achieve the business goal modeled in the corresponding workflow. A topological order of a generic directed acyclic graph — like a WF net — is a linear ordering of its vertices such that for every edge $\langle v, v'\rangle$ from vertex $v$ to vertex $v'$ it holds that $v$ comes before $v'$ in the ordering. In other words, a topological order is a traversal of the graph in which each vertex is visited only after all its dependencies are visited. We highlight that, while computing

Table 3: Assumptions on Workflows and AC Enforcement Mechanisms

| *A* | Description |
|---|---|
| *A1* | Workflows are written in standard BPMN |
| *A2* | Workflows do not contain loops |
| *A3* | Workflows contain messages and data associations objects |
| *A4* | Workflows group activities under pools and lanes |
| *A5* | Mechanisms implement the state-change rules in Table 1 |

[11] https://github.com/authzforce/server

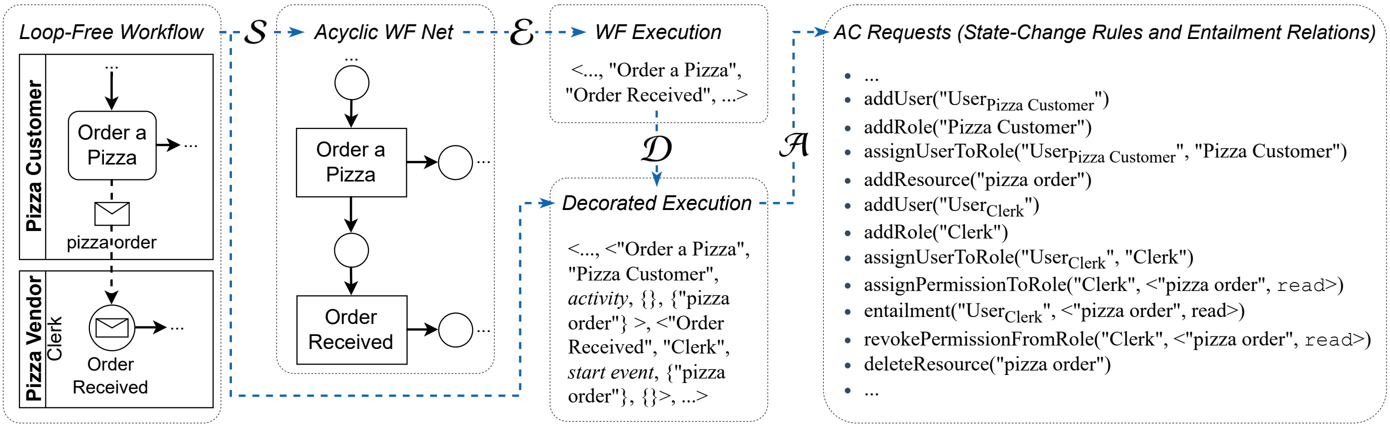


Figure 4: Application of ACE on (a Portion of) The Pizza Collaboration BPMN Workflow

the topological ordering of a WF net, we still have to respect the requirements for enabling transitions (see Section 3.2). Intuitively, this may lead to WF net executions that do not contain all transitions, as it is in the nature of WF nets that not all transitions are enabled; this may happen when, e.g., the original workflow contains exclusive gateways (see Section 3.1). Nonetheless, we are guaranteed that every transition is part of (at least one) WF net execution (recall the definition of WF net in Section 3.2). The output of this step is the list of (unique) WF net executions, where we label each transition with the name of the corresponding BPMN flow object (see $\mathcal{E}$ in Figure 4). We discuss the identification of WF net executions in more detail in Section 5.2.

### *4.1.3. Decorate Workflow Net Executions - $\mathcal{D}$*

We decorate each WF net execution previously found with further information extracted from the original workflow; we later use the information to derive requests. In particular, we associate to each transition — derived from a BPMN flow or connecting object (see [15]) — the pool or lane that contains the object (*A4*) and the type of the object itself. For instance, the transition corresponding to the "Order a pizza" activity is associated with the "Pizza Customer" pool in Figure 2. For flow objects, we also collect the identifiers of incoming and outgoing data (e.g., associations and messages) into two sets (*A3*). The output of this step is the list of decorated WF net executions (see $\mathcal{D}$ in Figure 4). We discuss the decoration of WF net executions in more detail in Section 5.3.

### *4.1.4. Derive Access Control Requests - $\mathcal{A}$*

We translate each decorated WF net execution given as input into a sequence of requests, where each request is a state-change rule (see Table 1) — in other words, requests and state-change rules map to the same concept. We highlight that the state-change rules are not manually inferred from the semantics of the transitions (e.g., the description of the activities of the original workflow), but instead they are automatically derived from the decorated WF net execution. More in detail, given a decorated WF net execution, we consider WF net transitions associated with incoming and outgoing data; the presence of such transitions corresponds to either the creation (e.g., for transient messages and data objects), modification (e.g., for persistent data stores) or reading of resources, for which a permission may be required and checked by an AC enforcement mecha-

Table 4: Methodology Conceptual Map

| Step | Section | | | | | Involved Concepts | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | High-Level Overview | Informal Explanation | Detailed Description | Implementation | Algorithm (If Present) | BPMN workflows (3.1) | BPMN pools/lanes/data (3.1.1) | WF nets (3.2) | WF nets markings (3.2) | WF net executions (3.2) | Topological ordering (4.1.2) | RBAC state-change rules (3.3) | RBAC policy states (3.3) | RBAC blueprints (3.3.1) | AC executions (3.3) |
| 1 — $\mathcal{S}$ | 1.2 | 4.1.1 | 5.1 | 5.5 | - | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 2 — $\mathcal{E}$ | 1.2 | 4.1.2 | 5.2 | 5.5 | 1 | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 3 — $\mathcal{D}$ | 1.2 | 4.1.3 | 5.3 | 5.5 | 2 | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 4 — $\mathcal{A}$ | 1.2 | 4.1.4 | 5.4 | 5.5 | 3 | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| 5 — Initializer | 1.2 | 4.2.1 | 6.1 | 6.3 | 4 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| 6 — Engine | 1.2 | 4.2.2 | 6.2 | 6.3 | - | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |

nism. Since business workflows are usually complemented with RBAC policies (see Section 2.1), we distribute and revoke permissions through roles, which we identify with BPMN pools and lanes. For instance, the "Pizza Customer" pool in Figure 2 corresponds to the "Pizza Customer" role. Consequently, the "Order a pizza" activity can be carried out only by a user assigned to the "Pizza Customer" role. The output of this step is, for each decorated WF net execution, a sequence of state-change rules (see $\mathscr{A}$ in Figure 4). We discuss the derivation of state-change rules from decorated WF net executions more in detail in Section 5.4.

### 4.2. ACME Overview

We assume an AC enforcement mechanism supporting RBAC to be capable of processing (at least) the state-change rules reported in Table 1; hence, a mechanism should allow manipulating the state of the RBAC policy (i.e., administrative requests) and evaluating the entailment relation (i.e., users' requests) — at least for read and write actions over a resource (*A5*). In other words, a mechanism should at least encompass the functions of a PAP (to evaluate administrative requests) and a PDP (to evaluate users' requests). Under this assumption, we note that ACME can interface with different kinds of mechanisms and it is not limited to OPA, XACML, and `CryptoAC`. As shown in Figure 1, ACME is composed of two modules, initializer and engine, which we describe below.

#### 4.2.1. Initializer

This module allows deriving an initial RBAC policy state for the mechanism under evaluation. The initializer takes three inputs: (*i*) the sequences of state-change rules obtained from ACE, (*ii*) the set of state-change rules $\Psi \subseteq \mathbf{\Psi}$ to consider (i.e., the *Types of Requests* in Figure 1), and (*iii*) an RBAC state blueprint (see Section 3.3). As anticipated in Section 1, instead of using the initializer, organizations can autonomously define an RBAC state representative of their internal structure — e.g., number of employees and qualifications.

#### 4.2.2. Engine

This module measures the performance of the mechanism under evaluation by simulating the execution of workflows to obtain representative results. The engine takes three inputs: (*i*) an RBAC policy state $\gamma \in \Gamma$ determined by either the initializer or the organization, (*ii*) the set of state-change rules $\Psi \subseteq \mathbf{\Psi}$ to consider (i.e., the *Types of Requests* in Figure 1), (*iii*) and the sequences of state-change rules produced by ACE (see Section 4.1). We highlight how the first two inputs (i.e., $\langle \gamma, \Psi \rangle$) constitute an RBAC system (recall the definition from Section 3.3). This RBAC system is used by the engine to induce AC executions from the sequences of state-change rules produced by ACE. Finally, the engine runs (possibly in a concurrent and intertwined fashion) the obtained AC executions by invoking the functionalities of the mechanism accordingly; note that the correct configuration and the deployment of the mechanism — which is treated like a black box, as shown in Figure 1 — are assumed to be correct and are not discussed in this paper.

### 4.3. Our Methodology vs. Micro-benchmarking

Micro-benchmarks are traditionally implemented by sending single (either administrative or users') requests to the mechanism being evaluated sequentially, as said in Section 2.2, while possibly scaling specific aspects of the AC policy state, like the number of roles (e.g., [6]) or attributes (e.g., [38, 34]). Although micro-benchmarks are surely useful to evaluate single functionalities of a mechanism, we believe that two fundamental requirements to obtain truly representative results for a specific scenario are to (*i*) consider a realistic AC policy state and (*ii*) expect multiple users to send requests concurrently; our methodology satisfies both. Potentially, also micro-benchmarks can satisfy these requirements; indeed, it is ideally possible to run multiple instances of a micro-benchmark concurrently and fix the AC policy to a realistic state. However, to the best of our knowledge, there are no such examples in the literature.

Nonetheless, we identify two structural issues in micro-benchmarks. First, micro-benchmarks are likely to activate optimizations, even though indirectly, that may alter the significance of the final results. The clearest example is caching, which allows a mechanism to answer more quickly to an (unrealistic) sequence of identical requests of the same type and parameters, as argued for XACML in [27, 9]. Caching applies both at the application level and to, e.g., CPU and RAM. Furthermore, given that some mechanisms may benefit from this kind of (unrealistic) caching more than others, we conclude that results obtained with micro-benchmarks can hardly be representative of a realistic scenario. By considering different kinds of requests (i.e., those discussed in Section 4.2), our methodology is way less likely to activate such optimizations and, in any case, those optimizations would be the same as — or at least similar to — those activated in a realistic scenario. The second issue concerns the typical usage of an AC enforcement mechanism, which largely depends on the business processes (i.e., the workflows) carried out by an organization. For instance, consider an organization that has to decide which AC enforcement mechanism to deploy between two mechanisms, $M_1$ and $M_2$. Now, assume that a micro-benchmark evaluation reports generally better results for $M_1$ over $M_2$, except for a few types of requests. Based on these results, the organization may then choose $M_1$ over $M_2$. However, this choice may be wrong, as the real answer depends on how frequently the types of requests in which $M_2$ performs better are present in the business processes — thus, in the workflows — of the organization. Therefore, contrary to what intuitively results from the micro-benchmark evaluation, $M_2$ may be a better fit for the specific scenario of the organization than $M_1$, even though generally less performing; our methodology is designed to address exactly this issue. We provide a concrete example of this situation in Section 7.4.

## 5. ACE

We now present our procedure, ACE, to extract sequences of state-change rules from workflows. First, we explain how to obtain a WF net from a workflow (Section 5.1). Then, we show how to identify all WF net executions (Section 5.2) and describe how to decorate them with information (Section 5.3) that we later use to extract sequences of state-change rules which are representative of the original workflow (Section 5.4). We represent each transformation as a function in Table 5; their nested invocation — which corresponds to ACE — precisely defines how to derive realistic sequences of state-change rules (i.e., those in Table 1) from a workflow specified in BPMN. Finally, we present the implementation of ACE (Section 5.5).

### 5.1. Map to Workflow Nets - $\mathcal{S}$

Dijkman et al. [15] use WF nets to provide a formal semantics to (the execution of) BPMN workflows. This semantics defines a *WF module* for several common BPMN flow objects (e.g., activities, events, and gateways) and connecting objects (e.g., sequence and messages). For instance, the semantics maps an activity onto a module composed of a transition with one input place and one output place; intuitively, the transition models the completion of the activity. Similarly, the semantics maps a message onto a module composed of a transition with one input place and one output place; intuitively, the transition models the transmission of the message. For further examples and more details on the semantics, we refer the interested reader to [15]. In our work, we use the proposed semantics to map BPMN flow and connecting objects to transitions and places. We denote with $\mathcal{S}$ (see Table 5) the function that takes as input a BPMN workflow $W$ — where loops were either absent or previously unrolled, as discussed in Section 4.1 — and returns an acyclic WF net $(P, T, F)$ using — hence, inheriting the accuracy and completeness of — the formal semantics in [15]; in other words, (the execution of) the WF net $(P, T, F)$ is representative of (the execution of) the business process modeled by $W$.

### 5.2. Identify Workflow Net Executions - $\mathcal{E}$

Given an acyclic WF net $(P, T, F)$, we compute its topological ordering to identify all WF net executions. In detail, we modify an algorithm for topological ordering based on depth-first search [14] according to the peculiarities of WF nets (see Section 3.2) and report it in Algorithm 1. The algorithm takes as input an acyclic WF net $(P, T, F)$ and returns a topological order $o$, which consists of an ordered sequence of $k$ transitions and places, i.e., $o = \langle v_1, ..., v_k\rangle \mid v \in (P \cup T)\ \forall v \in o \wedge v_1 = p_{source}$. As it is possible to see from Algorithm 1, our modification to the original approach in [14] is straightforward, and consists in starting the visit from $p_{source}$ by considering the canonical marking $m_0$ (line 3 in Algorithm 1) and then visiting a transition — and consequently deriving a new marking — only when the transition is enabled (lines 15—22). As mentioned in Section 4.1, a topological order for a WF net may not contain all vertices of the WF net, as not every transition is always enabled. Nonetheless, we are certain that Algorithm 1 always terminates as (*i*) the input WF net is acyclic — that is, it contains no loops — and (*ii*) every vertex of the WF net is on (at least one) directed path *from the source to the sink place* (see Section 3.2), i.e., visiting a vertex always brings the visit closer to the sink place. A corollary of (*ii*) is that Algorithm 1 exhaustively finds all WF net executions, under the assumption that all possible markings are considered. More concretely, given a topological order $o = \langle v_1, ..., v_k\rangle$ of a WF net $(P, T, F)$, we obtain a WF net execution $x = \langle t_1, ..., t_n\rangle$ by considering all and only transitions in $o$ (i.e., $t_i \in x \iff (\exists v \in o \mid v \in T \wedge t_i = v)$ for $0 < i < n$) while preserving the same ordering (i.e., $(t_i, t_j \in x \wedge 0 < i < j \leq n) \iff (\exists v_h, v_l \in o \mid t_i = v_h \wedge t_j = v_l \wedge 0 < h < l \leq k)$.

The fact that Algorithm 1 always terminates does not necessarily imply that each WF net execution is complete, i.e., correctly captures a way to achieve the modeled business goal; indeed, we are not guaranteed that the WF net satisfies the soundness criterion (recall the definition of these notions in Section 3.2). Determining the (even weak) soundness of an acyclic WF net is a co-NP-complete problem [45] that falls within the competence area of (BPMN) workflow designers. Hence, this problem is out of the scope of our paper; we refer the interested

**Algorithm 1:** Workflow Net Topological Ordering

**Input:** $(P, T, F)$
**Output:** $o/\perp$
1 let $A = \emptyset$; `// set of marked vertices`
2 let $o = []$; `// array for the topological order`
3 let $m = m_0$; `// start from the canonical marking`
4 **while** $\exists v \in (P \cup T) \mid v \notin A \wedge (v \in P \vee (m(p) = 1\ \forall p \in {}^{\bullet}v))$ **do**
5     invoke `visit`($v$);
6 **end**
7 **if** $m(p_{sink}) \neq 1$ **then**
8     **return** $\perp$;
9 **end**
10 **return** $o$;

11 **Function** `visit`($v$):
12     **if** $v \in A$ **then**
13         **return**;
14     **end**
15     **if** $v \in T$ **then**
16         **if** $m(p) > 0\ \forall p \in {}^{\bullet}v$ `// enable the transition`
17         **then**
18             $m(p) = m(p) - 1\ \forall(p \in {}^{\bullet}v)$;
19             $m(p) = m(p) + 1\ \forall(p \in v^{\bullet})$;
20         **else**
21             **return**;
22         **end**
23     **end**
24     invoke `visit`($u$) $\forall u \in (P \cup T) \mid \langle v, u\rangle \in F$;
25     add $v$ to $A$ and add $v$ to the head of $o$;

Table 5: Composition of ACE

| Function | Input | Output | Description |
|---|---|---|---|
| $\mathcal{S}$ | $W$ | $\langle P, T, F\rangle$ | Obtain an acyclic WF net $\langle P, T, F\rangle$ from the (loop-free) BPMN workflow $W$ using the formal semantics in [15] |
| $\mathcal{E}$ | $\langle P, T, F\rangle$ | $X$ | Identify the set of all (complete) WF net executions $X$ of the WF net $\langle P, T, F\rangle$ by repeatedly invoking Algorithm 1 |
| $\mathcal{D}$ | $W, X$ | $X^d$ | Derive the set of decorated WF net executions $X^d$ from the WF net executions $X$ of $W$ by repeatedly invoking Algorithm 2 |
| $\mathcal{A}$ | $X^d$ | $C$ | Derive the list of sequences of state-change rules $C$ from the decorated WF net executions $X^d$ by repeatedly invoking Algorithm 3 |
| $\text{ACE}(W) := \mathcal{A}(\mathcal{D}(W, \mathcal{E}(\mathcal{S}(W))))$ | | | |

reader to [45, 50, 49]. Here, we just consider complete WF net executions (lines 7—10). Finally, we denote with $\mathcal{E}$ (see Table 5) the function that takes as input a WF net $(P, T, F)$ and returns the set of all (complete) WF net executions $X$ by repeatedly invoking Algorithm 1 considering all possible markings (i.e., $[m_0\rangle$). We note that we can easily set an upper bound to the number of markings we consider, if needed.

### *5.3. Decorate Workflow Net Executions - $\mathcal{D}$*

After having computed the set of (complete) WF net executions $X$ through the topological ordering of the corresponding WF net $(P, T, F)$, we decorate each WF net execution with further information that we later use to extract sequences of state-change rules. We report the pseudocode for the decoration of a WF net execution in Algorithm 2. Given a WF net execution $x = \langle t_1, ..., t_n \rangle$ of a WF net derived from a workflow $W$ ($W$ is a required input as it contains the decorating information returned by the functions in lines 15—18 in Algorithm 2), we attach to each transition $t \in x$ — where $t$ was derived from a flow or connecting object with symbol $s$ (recall the definitions of these notions in Section 3.1) — the pool or lane $l$ that contains $s$ and $s$ itself, obtaining a tuple $\langle t, l, s \rangle$ (lines 3—4). If $s$ is a flow object, we also collect the identifiers of data (associations and messages) incoming to and outgoing from $s$ into two sets: one for incoming (i.e., required) data $req_t$ and one for outgoing (i.e., produced) data $prod_t$, respectively (lines 6—7). Summarizing, we attach $l$, $s$, $req_t$ and $prod_t$ to $t$; by abusing notation, we write $\langle t, l, s, req_t, prod_t \rangle$. A decorated WF net execution $x^d$ obtained from a WF net execution $x$ is thus of the form $\langle\langle t_1, l_1, s_1, req_{t_1}, prod_{t_1}\rangle, ..., \langle t_n, l_n, s_n, req_{t_n}, prod_{t_n}\rangle\rangle$. We denote with $\mathcal{D}$ (see Table 5) the function that takes as input a BPMN workflow $W$ and the set of WF net executions $X$ of the corresponding WF net, and returns the set of decorated WF net executions $X^d$ by repeatedly invoking Algorithm 2 on each $x \in X$.

**Algorithm 2:** Decoration of a Workflow Net Execution

**Input:** $W, x$
**Output:** $x^d$
1 let $x^d$ = []; `// array for the decorated transitions`
2 **foreach** $t \in x$ **do**
3 let $s$ = `findObject`$(t, W)$;
4 let $l$ = `findPoolOrLane`$(s, W)$;
5 **if** *s is a flow object* **then**
6 let $req_t$ = `findRequiredData`$(s, W)$;
7 let $prod_t$ = `findProducedData`$(s, W)$;
8 **else**
9 let $req_t = \emptyset$;
10 let $prod_t = \emptyset$;
11 **end**
12 add $\langle t, l, s, req_t, prod_t \rangle$ to $x^d$;
13 **end**
14 **return** $x^d$;

15 **Function** `findObject`$(t, W)$:
`// return` $t$`'s BPMN object in` $W$
16 **Function** `findPoolOrLane`$(s, W)$:
`// return` $s$`'s BPMN pool or lane in` $W$
17 **Function** `findRequiredData`$(s, W)$:
`// return` $s$`'s set of incoming data in` $W$
18 **Function** `findProducedData`$(s, W)$:
`// return` $s$`'s set of outgoing data in` $W$

### *5.4. Derive Access Control Requests - $\mathcal{A}$*

We now discuss how to derive sequences of state-change rules in $\mathbf{\Psi}$ (i.e., those reported in Table 1) which are representative of the business processes described by workflows. The ideal scenario would be for workflows to already embed AC information (e.g., describing the permissions of roles on resources). However, even though several researchers proposed to extend BPMN to express security requirements (e.g., [51, 36, 46, 8, 41, 40, 33]), at the time of writing BPMN does not integrate symbols to express AC information. Therefore, as similarly done in the literature (see Section 2.1) we rely on the syntax of the modeling language — BPMN, in our case — to extract state-change rules from workflows automatically. We report the pseudocode for the derivation of a sequence of state-change rules $c$ from a decorated WF net execution $x^d$ in Algorithm 3. Similarly to [16, 36], we create a role for each pool or lane, along with a user which we assign to the role (lines 5—9

**Algorithm 3:** Derivation of State-Change Rules

**Input:** $x^d$
**Output:** $c$
1 let $c$ = []; `// array for state-change rules`
2 let $r$ = []; `// array for roles already created`
3 **foreach** $\langle t, l, s, req_t, prod_t \rangle \in x^d$ **do**
4 **if** *s is a flow object* **then**
5 **if** $l \notin r$ **then**
6 add addUser($user_l$) to $c$;
7 add addRole($l$) to $c$;
8 add assignUserToRole($user_l, l$) to $c$;
9 add $l$ to $r$;
10 **else if** $s$ = "*end event*" **then**
11 add revokeUserFromRole($user_l, l$) to $c$;
12 add deleteRole($l$) to $c$;
13 add deleteUser($user_l$) to $c$;
14 remove $l$ from $r$;
15 **end**
16 **end**
17 **foreach** $f \in req_t$ **do**
18 let $p = \langle f, \texttt{read} \rangle$;
19 add assignPermissionToRole($l, p$) to $c$;
20 add entailment $\vdash (\langle user_l, p \rangle)$ to $c$;
21 add revokePermissionFromRole($l, p$) to $c$;
22 **if** *f is a transient resource* **then**
23 add deleteResource($f$) to $c$;
24 **end**
25 **end**
26 **foreach** $f \in prod_t$ **do**
27 **if** *f is a persistent resource* **then**
28 let $p = \langle f, \texttt{write} \rangle$;
29 add assignPermissionToRole($l, p$) to $c$;
30 add entailment $\vdash (\langle user_l, p \rangle)$ to $c$;
31 add revokePermissionFromRole($l, p$) to $c$;
32 **else**
33 add addResource($f$) to $c$;
34 **end**
35 **end**
36 **end**
37 **return** $c$;

in Algorithm 3). Once the corresponding pool is concluded — i.e., we reach the end event of the pool — we revoke the user from the role and delete both (lines 10—15). If a workflow does not group activities under pools and lanes (i.e., if *A4* in Table 3 is not respected), we can conservatively consider a dedicated user and role for each activity.

We add, read and write over resources according to the identifiers in $req_t$ and $prod_t$, distributing and revoking the corresponding permissions to the involved roles accordingly, as done in [51, 16, 36] (lines 17—35). Transient resources (e.g., messages and data objects) are created within the execution of the workflow and deleted once read, while persistent resources (e.g., data stores) are supposed to exist a-priori. Consequently, transient resources can only be added, read, and deleted, while persistent resources can only be read and written over. If a workflow does not contain messages or data associations (i.e., if *A3* in Table 3 is not respected), we can conservatively assign a (transient) input and output to each activity, as also done in [16, 48]; each activity would take as input the output of the activities preceding it.

Finally, we denote with $\mathscr{A}$ (see Table 5) the function that takes as input the set of decorated WF net executions $X^d$ and returns the list of sequences of state-change rules $C$ by repeatedly invoking Algorithm 3 on each $x^d \in X^d$.

### *5.5. Implementation of ACE*

We implement ACE (see Table 5) in Python — our implementation expects workflows encoded as XML files according to the BPMN standard — as open-source software. Then, we applied ACE on the workflow in Figure 2 to derive sequences of state-change rules.[12] As the input workflow needs to be acyclic, we first pre-process it by removing the loops it contains so as to respect *A2* in Table 3. In detail, we unroll the "Calm Customer"—"Where is My Pizza" loop by repeating it once. Then, we note a circular dependency between the "Pay the Pizza" and "Receive Payment" activities on the "money" and "receipt" messages. We can easily solve this dependency by splitting the "Pay the Pizza" activity into two (sub)activities, the first being the target of the sequence connecting object from the "Pizza received" event and producing the "money" message, and the second being the source of the sequence connecting object toward the "Eat the Pizza" activity and requiring the "receipt" message.

By applying ACE to the (pre-processed) workflow in Figure 2 we obtain 50 different WF net executions from which we derive 20 unique sequences of (the same) 54 state-change rules. First, we note that different WF net executions may sometimes lead to the same sequence of state-change rules. This happens when, e.g., WF net executions differ just for the order of transitions which do not yield any state-change rule (e.g., activities that do neither require nor produce any data). Then, as the workflow in Figure 2 does not contain any exclusive or inclusive gateway, all 20 sequences have the same 54 state-change rules but differ in their order. Nonetheless, we are still interested in these sequences, as we cannot know a-priori whether the order in which state-change rules are executed affects the performance of an AC enforcement mechanism. For instance, this is often the case in CAC, where the computational effort in distributing secret cryptographic keys depends on the current number of users, roles, and resources; consequently, adding a new role before or after distributing a new cryptographic key has an impact on performance [6].

We highlight that ACE is logically limited to the WF net mapping in [15]. However, it is relatively easy to add WF net modules for further BPMN symbols. Finally, we measure the running time of the implementation of ACE for the workflow in Figure 2, finding it to be 0.809s (average of 100 runs) on a device running Ubuntu 20.04 on an Intel(R) Core(TM) i7-11370H and 16GB of RAM. We believe that the running time of ACE is not really relevant, as ACE runs offline (i.e., with no real-time latency requirements) and just once per workflow.

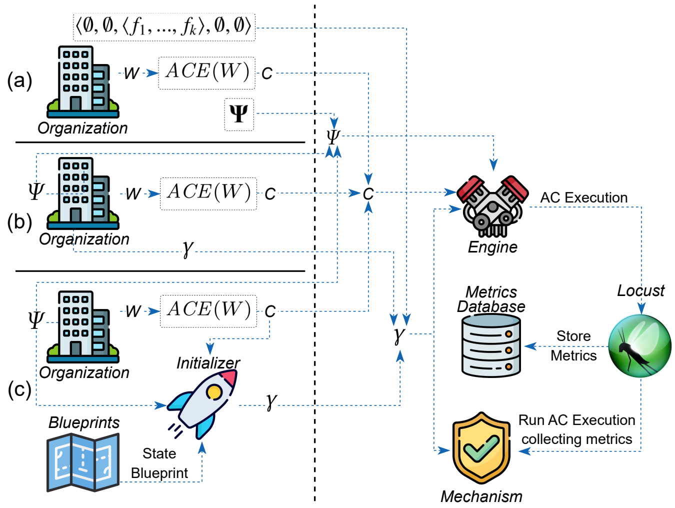


Figure 5: Representation of ACME in the Context of our Methodology for (a) Basic, (b) Highly Customized, and (c) Partially Customized RBAC System

## 6. ACME

We now present our simulator tool, ACME, for evaluating the performance of AC enforcement mechanisms based on a set of lists of sequences of state-change rules $\{C_1, ..., C_n\}$ derived by ACE (see Section 5) from a set of BPMN workflows $\{W_1, ..., W_n\}$ — i.e., $C_i$ =ACE($W_i$) for $1 \leq i \leq n$. ACME is composed of two modules: the initializer (Section 6.1) and the engine (Section 6.2). We represent ACME in the context of our methodology — previously shown in Figure 1 — in Figure 5. Finally, we present the implementation of ACME (Section 6.3).

### *6.1. Initializing the Mechanism*

The inducing of AC executions from $\{C_1, ..., C_n\}$ requires the definition of an RBAC system $\langle \gamma, \Psi \rangle$, as explained in Section 3.3. As said in Section 4.2, the set of state-change rules $\Psi \subseteq \mathbf{\Psi}$ is given by the organization (i.e., the *Types of Requests*

[12]The implementation of ACE is available at `https://github.com/stfbk/ACE`, while the sequences of state-change rules are available at `https://github.com/stfbk/ACME/tree/A_Simulation_Framework_Replication_Package`

in Figure 1), while the RBAC policy state $\gamma \in \Gamma$ is determined by either the initializer or the organization. Below, we elaborate on three alternatives for the definition of the RBAC system.

### *6.1.1. Basic RBAC System*

A sequence of state-change rules $c \in C$ produced by Algorithm 3 from a workflow $W$ contains — at least potentially — all state-change rules in $\mathbf{\Psi}$ (see Table 1); in other words, $c$ contains rules for creating (and deleting) users, roles and (transient) resources, and for assigning (and revoking) users and permissions to roles. Indeed, BPMN requires persistent resources only to exist before the execution of the workflow (see Section 5.4) — a requirement which is in line with their semantics (see Section 3.1). Therefore, the basic (i.e., simplest) RBAC system that can be used to induce an AC execution from $c$ — as shown in Figure 5a — is $\langle\langle\emptyset, \emptyset, \langle f_1, ..., f_k\rangle, \emptyset, \emptyset\rangle, \mathbf{\Psi}\rangle$, where $k$ is the number of persistent resources in $W$, $f_i$ is a persistent resource of $W$ for $0 < i \leq k$ and $\mathbf{\Psi}$ contains all state-change rules in Table 1.

### *6.1.2. Highly Customized RBAC System*

Evaluating the performance of AC enforcement mechanisms on all state-change rules in $\mathbf{\Psi}$ — as it happens when using the basic RBAC system — may be a realistic approach for some scenarios. For instance, a user may be assigned to a role only when needed to carry out a specific task (*run-time role provisioning*) and based on contextual conditions such as environmental attributes (e.g., time of day) and dynamic separation of duty constraints [30]. However, the same approach may be unrealistic for other scenarios. For instance, creating and deleting the "Clerk" role every time a customer orders a pizza in the workflow in Figure 2 may not be particularly meaningful or representative of the modeled business process; instead, we could assume that the "Clerk" role already exists in the initial state. Therefore — as shown in Figure 5b — an organization can autonomously define the RBAC system $\langle\gamma, \Psi\rangle$ used by ACME to induce AC executions. In other words, an organization can fine-tune the initial state of the RBAC policy — i.e., $\gamma$ — and the subset of state-change rules used to induce AC executions — i.e., $\Psi \subseteq \mathbf{\Psi}$. Although providing great flexibility, this definition requires some knowledge and effort on the organization's part, as the organization has to define a state $\gamma$ representative of its internal structure (e.g., number of employees and qualifications) manually or using ad-hoc policy state generation tools — e.g., those presented in [9, 2] specialized for XACML. Moreover, the names (i.e., identifiers) of users, roles, and resources in $\gamma$ need to match those contained in $\{C_1, ..., C_n\}$. Finally, $\gamma$ should be compatible with $\Psi$, i.e., the application of state-change rules in $\Psi$ should result in neither adding elements — i.e., users, roles, resources, assignments and permissions — already in $\gamma$ nor deleting elements not in $\gamma$; for instance, if $\gamma$ already contains users, then the addUser($\cdot$) state-change rule — where the "$\cdot$" symbol represents a wildcard value — should not be in $\Psi$.

Table 6: Partially Customized RBAC System Constraints

| |
|---|
| $Pr_1(\Psi)$= addUser($\cdot$)$\in\Psi \iff$ deleteUser($\cdot$)$\in\Psi$ |
| $Pr_2(\Psi)$= addRole($\cdot$)$\in\Psi \iff$ deleteRole($\cdot$)$\in\Psi$ |
| $Pr_3(\Psi)$= addResource($\cdot$)$\in\Psi \iff$ deleteResource($\cdot$)$\in\Psi$ |
| $Pr_4(\Psi)$= assignUserToRole($\cdot$)$\in\Psi \iff$ revokeUserFromRole($\cdot$)$\in\Psi$ |
| $Pr_5(\Psi)$= assignPermissionToRole($\cdot$)$\in\Psi \iff$ revokePermissionFromRole($\cdot$)$\in\Psi$ |
| $Pr_6(\Psi)$= (addUser($\cdot$)$\in\Psi\vee$addRole($\cdot$)$\in\Psi$) $\implies$ assignUserToRole($\cdot$)$\in\Psi$ |
| $Pr_7(\Psi)$= (addRole($\cdot$)$\in\Psi\vee$addResource($\cdot$)$\in\Psi$) $\implies$ assignPermissionToRole($\cdot$)$\in\Psi$ |

### *6.1.3. Partially Customized RBAC System*

As a middle way between a basic RBAC system — i.e., easy definition where an organization specifies neither $\gamma$ nor $\Psi$ — and a highly customized RBAC system — i.e., complex definition where the organization specifies both $\gamma$ and $\Psi$ — ACME provides the organization with an initializer that generates a suitable $\gamma$ automatically. More in detail — as shown in Figure 5c — the initializer derives a state $\gamma$ by combining the input set $\Psi$ of state-change rules chosen by the organization and the output of ACE $\{C_1, ..., C_n\}$ with an RBAC state blueprint like those presented in Table 2 (see Figure 5). In this way, besides the workflows, the organization just sets what state-change rules to use in the performance evaluation (i.e., $\Psi$). For instance, if the organization wants to evaluate the performance of a mechanism on the entailment relation only, then the organization can provide as input $\Psi = \{\vdash (\langle\cdot, \cdot\rangle)\}$, and the initializer can generate a full-fledged $\gamma$ already containing users, roles, resources, assignments and permissions. Similarly, the set $\Psi$ = {assignPermissionToRole($\cdot,\cdot$),revokePermissionFromRole($\cdot,\cdot$), assignUserToRole($\cdot,\cdot$),revokeUserFromRole($\cdot,\cdot$)} leads to an evaluation based on the assignment (and revocation) of users and permissions to roles only. Using the initializer, the organization can thus decide which state-change rules to consider in AC executions — to a certain extent. Indeed, to prevent inconsistencies, AC executions must have the same initial and final RBAC state; otherwise, it would be impossible to run an AC execution more than once. For instance, if an AC execution creates a role, that role must be deleted before re-running the same AC execution — as otherwise, in the second run, the AC execution would try to create a role that already exists, resulting in an error. Formally, given an RBAC system $\langle\gamma, \Psi\rangle$ and $c = \langle\psi_1, ..., \psi_{n-1}\rangle$ (where $\psi_i \in \Psi$ for $0 < i < n$), $\Psi$ must be defined so to induce an AC execution $\langle\gamma_1, \psi_1, \gamma_2, ..., \gamma_{n-1}, \psi_{n-1}, \gamma_n\rangle$ where $\gamma_1 = \gamma_n = \gamma$. We translate this requirement for $\Psi$ into 7 constraints which we report in Table 6 as predicates $Pr_i(\Psi)$ for $1 \leq i \leq 7$. The first 5 predicates state that every element added to the state must also be removed from the state before the end of the AC execution. Then, if users or roles are created dynamically during the execution of a workflow, it naturally follows that assignments of users to roles are created dynamically as well; the same concept also applies to roles and persistent resources (last 2 predicates in Table 6). Instead, transient resources can exist within the execution of a workflow only (see Section 3.1) and — by definition — can never be part of the initial state $\gamma$. More formally, the organization can pick any one set of state-change rules among those in the set $\{\Psi \mid \Psi \in \mathcal{P}(\mathbf{\Psi}) \wedge Pr_1(\Psi) = Pr_2(\Psi) = Pr_3(\Psi) = Pr_4(\Psi) = Pr_5(\Psi) = Pr_6(\Psi) = Pr_7(\Psi) = true\}$, where $\mathcal{P}(\mathbf{\Psi})$ represents the power set of $\mathbf{\Psi}$.

We report the pseudocode of the initializer in Algorithm 4. First, the initializer checks the consistency of the blueprint, i.e., it asserts that the numbers of users, roles, and resources are compatible with the minimum and maximum numbers of assignments and permissions (line 1 in Algorithm 4), and prepares an empty state (line 2). Then, the initializer populates **U** (lines 3—8) and **R** (lines 9—14) if addUser($\cdot$) $\notin \Psi$ and addRole($\cdot$) $\notin \Psi$, respectively, with names extracted from $\{C_1, ..., C_n\}$ (lines 43—44). Persistent resources are always present in **P** (lines 15—18 and 45), while transient resources are created during AC executions only. Afterwards, if assignUserToRole($\cdot,\cdot$) $\notin \Psi$, the initializer assigns users to roles (i.e., **UR**) according to the input blueprint (lines 19—40). The distribution of permissions of roles over (persistent) resources (i.e., **PA**) follows the same concept — for brevity, we omit it from Algorithm 4.

### 6.2. Running the Engine

As shown in Figure 5, the engine takes as input an RBAC system $\langle \gamma, \Psi \rangle$ and $\{C_1, ..., C_n\}$ derived from $\{W_1, ..., W_n\}$ as described in Section 5. Then, the engine induces an AC execution — containing only state-change rules in $\Psi$ — for each sequence of state-change rules in $\{C_1, ..., C_n\}$, as discussed in Section 3.3. Finally, the engine runs each AC execution by invoking — for each state-change rule — the corresponding functionalities (or APIs) of the mechanism being evaluated.

The engine can run one or more AC executions for each $\{C_1, ..., C_n\}$ — i.e., for each workflow — concurrently to simulate the presence of multiple ongoing business processes. The engine also allows defining weights — expressed as natural numbers — to guide the selection of which workflow(s) to instantiate, as to model the fact that some workflows occur more frequently than others. For instance, given $\{C_1, ..., C_n\}$, assigning to $C_i$ a weight $w_{C_i} = 2$ — while $w_{C_j} = 1$ for $1 \leq j \leq n \mid i \neq j$ — means that the workflow $W_i$ corresponding to $C_i$ (i.e., $C_i$ =ACE($W_i$)) occurs twice as often than the other workflows; organizations can derive values for these weights from, e.g., logs and analytic tools integrating workflow engines. Similarly, the engine can run multiple AC executions of the same workflow concurrently; again, organizations can assign weights to AC executions. In any case, the engine takes care of avoiding inconsistencies in the AC policy state (e.g., two AC executions trying to add the same role at the same time) by running each AC execution in a separate and isolated context, as suggested in [51]. In detail, all AC executions of a workflow — and of other workflows as well — share the same initial state, but then any state-change rule — albeit being applied over the same state — is customized to each AC execution. Concretely, this means that identifiers of users, roles, and transient resources are made unique to each AC execution. For instance, two AC executions $c_1$ and $c_2$ of the workflow in Figure 2 induced by a basic RBAC system (see Section 6.1.1) would not both create the "Clerk" role, but instead two "Clerk$_1$" and "Clerk$_2$" roles. If an organization deems that creating a different version of the same role for each AC execution of a workflow is not realistic or representative of the modeled business process, then the organization can just avoid running multiple AC executions of the same workflow concurrently. Otherwise, the organization can assume the role to already exist in the initial state, thus choosing a highly or partially customized RBAC system (see Sections 6.1.2 and 6.1.3). The same reasoning applies to users and transient resources, while persistent resources — according to their semantics — are shared among all AC executions.

Finally, whenever a user is selected to carry out an action (e.g., read a resource) through a role in an AC execution, that user is marked as "busy", and no other AC execution — even of other workflows — can select that user until the user's first ac-

**Algorithm 4:** The Pseudocode of the Initializer

**Input:** $\Psi, \{C_1, ..., C_n\}, \langle |\mathbf{U}|, |\mathbf{R}|, |\mathbf{P}|, |\mathbf{UR}|, |\mathbf{PA}|, m_{(u\to r)}, M_{(u\to r)}, m_{(r\to u)}, M_{(r\to u)}, m_{(p\to r)}, M_{(p\to r)}, m_{(r\to p)}, M_{(r\to p)} \rangle$
**Output:** $\langle \mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{UR}, \mathbf{PA} \rangle$

1 assert $(|\mathbf{U}| \cdot m_{(r\to u)}) \leq |\mathbf{R}| \leq (|\mathbf{U}| \cdot M_{(r\to u)}) \wedge (|\mathbf{R}| \cdot m_{(u\to r)}) \leq |\mathbf{U}| \leq (|\mathbf{R}| \cdot M_{(u\to r)}) \wedge (|\mathbf{R}| \cdot m_{(p\to r)}) \leq |\mathbf{P}| \leq (|\mathbf{R}| \cdot M_{(p\to r)}) \wedge (|\mathbf{P}| \cdot m_{(r\to p)}) \leq |\mathbf{R}| \leq (|\mathbf{P}| \cdot M_{(r\to p)})$;
2 let $\mathbf{U} = \mathbf{R} = \mathbf{P} = \mathbf{UR} = \mathbf{PA} = \emptyset$;
3 **if** *(addUser($\cdot$) $\notin \Psi$)* **then**
4   let $u$ = `getUsernames`($|\mathbf{U}|, \{C_1, ..., C_n\}$);
5   **for** $i \leftarrow 1$ *to* $|\mathbf{U}|$ **do**
6     add $u[i]$ to **U**;
7   **end**
8 **end**
9 **if** *(addRole($\cdot$) $\notin \Psi$)* **then**
10   let $r$ = `getRoleNames`($|\mathbf{R}|, \{C_1, ..., C_n\}$);
11   **for** $i \leftarrow 1$ *to* $|\mathbf{R}|$ **do**
12     add $r[i]$ to **R**;
13   **end**
14 **end**
15 let $f$ = `getPersistentResourceNames`($|\mathbf{P}|, \{C_1, ..., C_n\}$);
16 **for** $i \leftarrow 1$ *to* $|\mathbf{P}|$ **do**
17   add $f[i]$ to **P**;
18 **end**
19 **if** *(assignUserToRole($\cdot,\cdot$) $\notin \Psi$)* **then**
20   let $u2r$ = {}; `// Num of assigned roles per user`
21   let $r2u$ = {}; `// Num of assigned users per role`
22   let $counter$ = 0; `// Total user-role assignments`
23   **foreach** $u$ *in* **U** **do**
24     **while** $u2r[u] < m_{(r\to u)}$ **do**
25       pick $r \in \mathbf{R} \mid r2u[r] < M_{(u\to r)}$;
26       invoke `assignUserToRole`($u, r$);
27     **end**
28   **end**
29   **foreach** $r$ *in* **R** **do**
30     **while** $r2u[r] < m_{(u\to r)}$ **do**
31       pick $u \in \mathbf{U} \mid u2r[u] < M_{(r\to u)}$;
32       invoke `assignUserToRole`($u, r$);
33     **end**
34   **end**
35   **while** $counter < |\mathbf{UR}|$ **do**
36     pick $u \in \mathbf{U} \mid u2r[u] < M_{(r\to u)}$;
37     pick $r \in \mathbf{R} \mid r2u[r] < M_{(u\to r)}$;
38     invoke `assignUserToRole`($u, r$);
39   **end**
40 **end**
41 `... // Assign permissions to roles`
42
43 **Function** `getUsernames`($m, \{C_1, ..., C_n\}$)**:**
  `// return` $m$ `user names from` $\{C_1, ..., C_n\}$
44 **Function** `getRoleNames`($m, \{C_1, ..., C_n\}$)**:**
  `// return` $m$ `role names from` $\{C_1, ..., C_n\}$
45 **Function** `getPersistentResourceNames`($m, \{C_1, ..., C_n\}$)**:**
  `// return` $m$ `resource names from` $\{C_1, ..., C_n\}$
46 **Function** `assignUserToRole`($u, r$)**:**
47   add $(u, r)$ to **UR**;
48   set $u2r[u] = u2r[u] + 1$;
49   set $r2u[r] = r2u[r] + 1$;
50   set $counter = counter + 1$;

tion is completed. As also suggested in [51], this approach concurs in simulating a realistic environment in which each user carries out a single activity at a given time, but can take part in more workflows simultaneously. If no user is available to carry out an action in an AC execution (because, e.g., all users are busy with other activities), the AC execution becomes "idling" until a user with the required role assignment is available. The idling time is measured separately from the workflow execution time and may potentially be used to identify bottlenecks in the AC policy, i.e., important roles with too few users assigned because of, e.g., separation of duty constraints; we defer the study of this possibility to future work (see Section 8.1).

### 6.3. Implementation of ACME

We implement ACME in Python and make the implementation available as open-source software.[13] ACME is independent of the underlying AC enforcement mechanism, that is, ACME has to be complemented with an "adapter" (i.e., a Python subclass) acting as an interface toward a specific AC enforcement mechanism. In other words, an adapter for ACME specifies how to translate a given state-change rule in Table 1 — that is, an AC request produced by ACE — into the sequence of API invocations to implement that rule in the corresponding mechanism. We implement 3 adapters in ACME for OPA, the AuthzForce Server (Community Edition) open-source implementation of XACML, and `CryptoAC`. We implement the first two adapters by encoding RBAC policy states as described in the OPA documentation[14] and the XACML standard,[15] respectively, while `CryptoAC`'s APIs already have a 1-to-1 mapping with the state-change rules in Table 1.[16] More in detail, the XACML standard prescribes that the state of the RBAC policy should be incorporated directly into policy sets, policies, and rules. Consequently, the adapter for XACML translates a given state-change rule in a XACML request by creating XML documents relying on the "urn:oasis:names:tc:xacml:3.0:core:schema:wd-17" namespace. In particular, XML documents corresponding to requests for adding user-role or role-permission assignments are derived by using the `<PolicySet>`, `<Policy>`, and `<Rule>` namespace elements, while entailment relations are derived by using the `<Request>` namespace element.[17] We remark that the way a mechanism actually carries out a state-change rule (e.g., assigning permissions resource-wise, based on patterns, or by distributing cryptographic keys to authorized users) depends on the implementation of the mechanism itself, whose performance is exactly the object of our evaluation. In this regard, we notice that the performance of a mechanism is not limited to the evaluation of state-change rules only, but it also includes the handling of resources. For instance, CAC-based enforcement mechanisms perform cryptographic computations on resources [6]; omitting these computations would lead to non-realistic results. For this reason, we represent a resource as a 1MB file, although organizations can easily modify this representation (e.g., using a message queue instead of files for an IoT-based workflow) or the file size resource-wise in ACME. Then, we implement the adapters so that each of the 4 state-change rules concerning (both transient and persistent) resources — i.e., addResource($f$), deleteResource($f$), and the entailment relation for read and write queries over resources — considers the handling of resources themselves. For instance, reading a file in `CryptoAC` implies both evaluating the entailment relation and also downloading — and decrypting — the (1MB file representing the) resource. Further details can be found directly in the documentation of the adapters.[18]

Finally, ACME uses Locust,[19] a modern open-source load testing tool, as the underlying framework to invoke APIs while collecting two metrics, i.e., the number of Requests per Second (RPS) and the response time of each API invocation. Locust can perform load testing through any kind of API, not only HTTP-based, and can easily distribute the load generation across multiple devices[20] to, e.g., test the scalability of a mechanism, as we show in Section 7.

## 7. Methodology Application

In this section, we first define the experimental settings for the application of our methodology (Section 7.1). Then, we use our methodology to evaluate `CryptoAC`, OPA, and the AuthzForce implementation of XACML (Section 7.2). Finally, we discuss the obtained results, arguing the internal and external validity of our evaluation (Section 7.3) and comparing them against micro-benchmarking (Section 7.4).[21]

### 7.1. Experimental Settings

We now define the set of workflows $\{W_1, ..., W_n\}$, RBAC system $\langle \gamma, \Psi \rangle$, and infrastructure used in our experimentation.

#### 7.1.1. Workflows and RBAC System

Our experimental evaluation does not involve a specific scenario or organization, thus we lack both the business processes (i.e., the workflows) and the RBAC system. Therefore, we employ 5 workflows from the official OMG report [23], i.e., "The Pizza Collaboration" ($W_1$ hereafter) — already presented in Section 3.1 — the "The Nobel Prize" ($W_2$), the "Incident Management as Detailed Collaboration" ($W_3$), the "BPI Web Registration with Moderator" ($W_4$) and the "Patient Treatment - Collaboration" ($W_5$).[22] We choose these workflows among

[13] `https://github.com/stfbk/ACME/`

[14] `https://www.openpolicyagent.org/docs/latest/comparison-to-other-systems/`

[15] `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.pdf`

[16] `https://cryptoac.readthedocs.io/en/latest/`

[17] The interested reader can find the description of the generation of role-permission assignment requests in Sections 2.1 to 2.4 of the XACML standard, the description of the generation of user-role assignment requests in Section 3 of the XACML standard, and the description of the generation of the entailment relation request in Section 2.5 of the XACML standard. The corresponding implementation can be found in ACME's source code at `https://github.com/stfbk/ACME/`

[18] `https://github.com/stfbk/ACME/`

[19] `https://locust.io/`

[20] `https://docs.locust.io/en/stable/running-distributed.html`

[21] The results, plots, and replication package are available at `https://github.com/stfbk/ACME/tree/A_Simulation_Framework_Replication_Package`

[22] The workflows are available at `https://github.com/stfbk/ACME/tree/A_Simulation_Framework_Replication_Package`

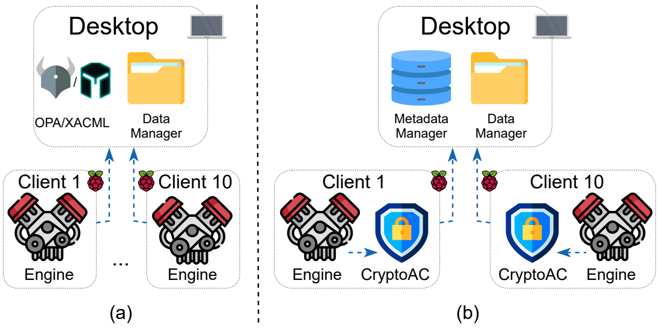


Figure 6: Infrastructure for (a) Centralized and (b) Decentralized Evaluations

those available in the OMG report as they group activities under pools and lanes — allowing for clear identification of roles, as discussed in Section 5.4 — contain exclusive gateways — originating different sequences of state-change rules — and encompass several types of resources (i.e., messages, data objects and data stores). Moreover, the use of CAC (i.e., `CryptoAC`) is relevant to workflows dealing with sensitive data (e.g., $W_2$, $W_5$) that need to be protected by both AC and cryptography. We parse these workflows with ACE (see Section 5), obtaining $\{C_1, ..., C_5\}$ and set all weights (recall the concept of weight in Section 6.2) to 1 for the sake of simplicity. ACE identifies 576 unique WF net executions among 5.586 for $W_2$, 313 unique WF net executions among 761 for $W_3$, 11 unique WF net executions among 18 for $W_4$ and 9 unique WF net executions among 9 for $W_5$. Combined, these 5 workflows expect 17 users, 17 roles, and 39 (4 persistent and 35 transient) resources. Finally, we partially customize the RBAC system (see Section 6.1.3) by considering that the AC policy state already contains users, roles, persistent resources, and assignments of users and permissions to roles (as it is usually the case [17]). Concretely, we obtain $\gamma$ from the initializer by providing as input $\Psi = \{\vdash (\langle \cdot \cdot \rangle)\}$, $\{C_1, ..., C_5\}$ and the *healthcare* state blueprint (shown in Table 2), whose numbers for roles and resources are closer to these workflows than those of the other blueprints.

### *7.1.2. Infrastructure*

We present in Figure 6 the infrastructure used in our evaluation (we do not represent Locust or the database components for the sake of simplicity). For both configurations (i.e., centralized with OPA or XACML and decentralized with `CryptoAC`), we use 10 Raspberry Pi 3 Model B+[23] running the Raspberry Pi Operating System (OS) as clients hosting the engine.

For the centralized configuration (Figure 6a), we deploy OPA or the AuthzForce Server for XACML on a desktop Dell Precision T1650[24] running Linux Mint 19 cinnamon. The deployment of OPA consists of two (micro) services, i.e., OPA itself (which evaluates state-change rules) and a data manager for handling resources created, updated, read, and deleted during the execution of workflows (recall the discussion about resources in Section 6.3); in other words, while OPA acts as both PDP and PAP, the data manager acts as a PEP. Similarly, the deployment of XACML consists of the AuthzForce Server — acting as PDP and PAP — and the data manager — acting as PEP. As said in Section 6.3, we encode the RBAC policy in OPA and in the AuthzForce Server as described in the OPA documentation[25] and the XACML standard,[26] respectively, and develop the data manager as a simple Kotlin program exposing four APIs following the Create, Read, Update and Delete (CRUD) paradigm to handle resources. For the decentralized configuration (Figure 6b), we deploy both the engine and `CryptoAC` on each of the Raspberry Pi. Indeed, `CryptoAC` provides decentralized enforcement for RBAC policies through cryptography, i.e., `CryptoAC` is expected to run on the clients in a distributed fashion. The deployment of `CryptoAC` consists of three (micro) services: `CryptoAC` itself — which performs the cryptographic computations described in [6] — a data manager (we reuse the same service of the centralized configuration) and a metadata manager for metadata (e.g., public keys). The last two services run on the desktop centrally, as expected by [6].

We connect all devices (i.e., the 10 Raspberry Pi and the desktop) to the same wired LAN to guarantee a reliable network. We measure the latency and the throughput from each Raspberry Pi to the desktop with the `iperf3` tool,[27] finding them to be less than 1 ms and equal to 294 Mbps, respectively.

### *7.2. Experimental Evaluation*

Below, we define how we elaborate the measurements collected during the evaluation, present the experiments we conducted, and report the obtained results.

#### *7.2.1. Measurements Elaboration*

We are interested in evaluating the performance — i.e., scalability, response time and throughput — of OPA, XACML and `CryptoAC`. Therefore, we aggregate individual response times of HTTPS requests to determine the overall average response time for the AC executions of a workflow — that is, the average overhead due to the AC enforcement mechanism in the execution of the workflow (*workflow response time*). Moreover, we measure the number of RPS (*throughput*). The amount of RAM consumed may be another relevant measurement. However, `CryptoAC` is deployed on different devices with respect to OPA and the AuthzForce Server, and consists of a different number of services (as described in Section 7.1). Thus, their RAM consumptions are not comparable, at least not directly. As such, we decide not to measure the amount of RAM consumed. Moreover, we highlight that there already exist several tools (e.g., `top`,[28] Valgrind[29]) allowing to measure the RAM consumed by a process.

[23] Each Raspberry Pi is equipped with a Cortex-A53 at 1.4GHz with 1GB of LPDDR2 SDRAM and Gigabit ethernet with a maximum throughput of 300 Mbps

[24] The desktop is equipped with an Intel Xeon E3-1240 V2 at 3.40GHz with 16GB of DDR3 RAM and Gigabit Ethernet

[25] https://www.openpolicyagent.org/docs/latest/comparison-to-other-systems/

[26] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.pdf

[27] https://iperf.fr/iperf-download.php

[28] https://man7.org/linux/man-pages/man1/top.1.html

[29] https://valgrind.org/

### 7.2.2. Experiments

To investigate the (presumed) benefits of the decentralization of CryptoAC with respect to the (centralized) OPA and XACML (i.e., to measure scalability), we exploit Locust's distributed load generation capabilities (see Section 6.3) to scale the number of clients (i.e., engine instances) running AC executions concurrently from 5 to 10. In other words, we repeat the experiment 6 times for CryptoAC, OPA and XACML (i.e., 18 experiments in total), each time increasing the number of clients (thus, increasing the load on centralized services). At the beginning of each experiment, we launch the engine in each of the clients involved in the current evaluation, providing as input $\{C_1, ..., C_5\}$ and the RBAC system $\langle \gamma, \Psi \rangle$ (see Section 7.1). We activate clients 5 seconds apart from each other to avoid them starting at the same time; this allows intertwining AC executions of workflows, i.e., ensuring that AC executions run at different stages, as in a realistic scenario. We let the engine run AC executions multiple times by setting a time limit of 20 minutes. In short, we run each experiment separately, resetting all services each time.

### 7.2.3. Results

As clients activate 5 seconds apart from each other (i.e., with 10 clients, it takes 45 seconds to fully start the experiment), we discard the first minute's worth of results in which not all clients have started already; we do this for all 18 experiments to match their duration. We report in Table 7 the evaluation results, from 5 to 10 clients (*Cl*), considering only AC executions that run entirely in the experimentation time span. In detail, we report the average workflow response time of AC executions of the 5 workflows and the average throughput (*TR*) for OPA (columns 2—7), XACML (columns 8—13) and CryptoAC (columns 14—19). We also chart the results of Table 7 for $W_1$ in the plot in Figure 7:[30] on the X-axis we report the number of users, while on the Y-axis we report the average workflow response time (in milliseconds). We mark average values for OPA with blue circles, while we mark average values for XACML and CryptoAC with red crosses and green squares, respectively. Finally, the black line traversing each set of marks of a mechanism is the linear regression model computed on the average values. As Locust keeps track of the number of RPS globally, we highlight that the throughput amounts at the number of HTTPS requests satisfied (i.e., answered) per second by the mechanisms across AC executions of the 5 workflows. We remark once again that the average workflow response times shown in Figure 7 and reported in Table 7 do not refer to the time an AC enforcement mechanism takes to evaluate a single AC state-change rule, but rather to the evaluation of the whole workflow $W_1$ (which comprises 54 state-change rules, as explained in Section 5.5) when concurrently executed by 10 users — that is, when stressing the AC enforcement mechanism with many requests. For instance, if we consider the average throughput results with 10 clients reported in Table 7 (last row of columns 7, 13, and 19), we infer that OPA, XACML, and CryptoAC take 3.30ms, 59.38ms, and 35.63ms to elaborate a single AC state-change rule on average, respectively (that is, 1000ms divided by 302.9, 16.84, and 28.07).

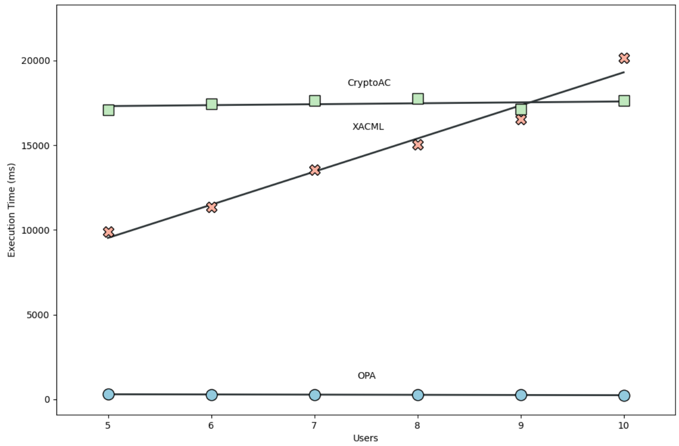


Figure 7: Average Workflow Response Times for $W_1$

### 7.3. Discussion

We can draw a number of considerations from Table 7 and Figure 7. First, we observe that OPA is faster and has a higher throughput than both XACML and CryptoAC. For instance, OPA takes 301.92ms on average for running an AC execution of $W_1$ with 5 clients, while XACML takes 9,880.10ms and CryptoAC takes 17,091.03ms. CryptoAC is the slowest among the evaluated mechanisms, probably because it enforces AC through cryptography, i.e., it performs several cryptographic computations for each state-change rule [6]. Instead, it is noticeable how the workflow response time of XACML — which is still a centralized mechanism — is an order of magnitude worse than the workflow response time of OPA. A possible explanation is that OPA stores all AC policies in cache memory, resulting in a faster evaluation; moreover, AC policies in OPA are stored as code instead of eXtensible Markup Language (XML), a cumbersome language to read and parse.[31] Then, XACML was designed to enforce ABAC rather than RBAC policies, and the standard encoding of RBAC for XACML seems to not be optimal. Finally, we remark that we are evaluating just one of the several implementations for XACML (i.e., the AuthzForce Server, which is not representative of all implementations of XACML) which is also open-source, hence not necessarily optimized or ready for production.

From Table 7 — and especially from Figure 7 — we also notice how the average workflow response times of AC executions change with the number of clients. The workflow response time of OPA across the 5 workflows remains almost constant from 5 to 10 clients, showing great vertical scalability and, perhaps, the presence of optimizations (e.g., caching); in any case, we highlight that these optimizations are the same that would

[30] The plots for $W_2$, $W_3$, $W_4$, and $W_5$ are available at https://github.com/stfbk/ACME/tree/A_Simulation_Framework_Replication_Package

[31] https://www.styra.com/blog/opa-vs-xacml-which-is-better-for-authorization/

be activated in a realistic scenario, as discussed in Section 4.3, thus they concur in producing realistic results. Conversely, the workflow response time of XACML quickly deteriorates when increasing the number of clients, suggesting low vertical scalability (e.g., from 9,880.1ms with 5 clients to 20,138.11ms with 10 clients for $W_1$, as shown in Figure 7); instead, the workflow response time of `CryptoAC` seems unaffected by the number of clients. In fact, the results indicate that `CryptoAC` has excellent vertical scalability — probably due to being a decentralized AC enforcement mechanism — even to the point that `CryptoAC` performs better than XACML when considering 10 clients, as illustrated in Figure 7. This impression is also confirmed by the results on the throughput, which increases linearly with the number of clients for `CryptoAC` (i.e., from 14.02 RPS with 5 clients to 28.07 RPS with 10 clients) while remains almost constant for XACML.

Finally, we remark that, besides performance, there exist other (subjective) conditions — not immediately related to performance but relevant nonetheless — which an organization may want to consider when choosing an AC enforcement mechanism, as discussed in Section 1. For instance, even though `CryptoAC` is slower than OPA, it provides greater security and privacy guarantees for the confidentiality and integrity of data through the use of cryptography. Thus, an organization should carefully evaluate the trade-off between performance and security, depending on the sensitivity of the involved data and the requirements of the underlying scenario.

### *7.3.1. Internal validity*

As also argued in [9], internal validity is relatively easy to achieve, since we control all experimental settings (i.e., those discussed in Section 7.1) directly. In other words, it is possible to define the computational and network resources available, and even fine-tune them to better reproduce specific scenarios (e.g., by artificially introducing network latency with tools such as `tc`[32]). Then, the implementation of the core logic of the engine is common for all configurations (i.e., OPA, XACML and `CryptoAC`); only the adapter — which essentially just sends HTTPS requests — differs. We even use the same data manager service in all configurations to avoid introducing biases. As a side note, we highlight that ACME is flexible enough to re-run the same evaluation with different components (e.g., data manager) to measure their efficiency in realistic scenarios. Furthermore, one may ask whether having both `CryptoAC` and the engine on the same device (i.e., the Raspberry Pi) could alter the results of the evaluation, as the two services might compete over the (limited) computational resources of the device. Therefore, during the experiments of `CryptoAC`, we monitor the CPU and RAM usage of the Raspberry Pi clients, finding them to be constantly around 40% and 30%, respectively. Consequently, we conclude that the two services can run together on the Raspberry Pi without any mutual interference, i.e., without altering the results. Finally, one may deem 10 concurrent clients to be too few for a realistic evaluation. However, we remark that — within any system — only a small fraction of the total number of clients is active simultaneously. For instance, industrial benchmarks have quantified the average number of clients active concurrently to be within the range 10%-20% of the total number of clients,[33] while empirical measurements in research studies estimate an even lower percentage (e.g., 5% in [25]). If we consider that the healthcare state blueprint employed in our experimental evaluation expects 46 total users (see Table 2), we can conclude that 10 clients — which are 22% of 46 — is realistic for emulating a medium-load scenario. Moreover, a similar range of clients has been considered in prior work (e.g., [39] uses 5 to 10 concurrent clients). As a final remark, the primary objective of our experimental evaluation is to validate the methodology — our core contribution — by providing initial and realistic evidence of its applicability, and not to deliver large-scale performance benchmarks (which may be interesting but would deserve their own space in a separate follow-up article).

### *7.3.2. External validity*

We now discuss whether our findings can generalize to actual deployments. First, we consider that often — although not always — workflows have a mix of activities carried out by software (i.e., automatically) and humans (i.e., manually) [51]. However, the results obtained with our methodology concern the overhead due to AC enforcement mechanisms in the execution of workflows only. To obtain more representative results, these can be integrated (i.e., summed) with the time needed by software agents and humans to carry out the activities in the workflows (which can be defined by organizations). Intuitively, this would dilute (in relative terms) eventual performance gaps among mechanisms. Nonetheless, this time is also independent of the mechanism being used, thus it does not alter its performance or a comparison with other mechanisms. Similarly, network latency — which is negligible in our evaluation — is another relevant aspect that may have a significant impact on

[32] `https://man7.org/linux/man-pages/man8/tc.8.html`

[33] `https://loadfocus.com/blog/2025/04/calculate-concurrent-users`

Table 7: Average Workflow Response Time (ms) and Throughput *TR* (i.e., Number of Requests per Second)

| *Cl* | OPA | | | | | | XACML | | | | | | `CryptoAC` | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | *TR* | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | *TR* | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | *TR* |
| 5 | 301.92 | 457.65 | 407.25 | 111.34 | 327.14 | 141.7 | 9,880.1 | 15,059.98 | 9,792.55 | 3,191.33 | 11,210.61 | 16.39 | 17,091.03 | 26,577.29 | 16,185.51 | 6,089.74 | 18,227.53 | 14.02 |
| 6 | 285.14 | 429.02 | 382.0 | 107.35 | 312.71 | 172.45 | 11,339.37 | 17,282.57 | 11,836.76 | 3,626.97 | 13,958.79 | 16.49 | 17,460.31 | 27,027.49 | 16,141.53 | 6,240.69 | 18,309.22 | 16.7 |
| 7 | 274.62 | 405.0 | 361.99 | 101.9 | 298.62 | 206.93 | 13,556.74 | 20,474.54 | 13,086.97 | 3,940.23 | 15,584.98 | 17.01 | 17,624.6 | 27,275.6 | 15,933.43 | 6,307.76 | 18,500.2 | 19.31 |
| 8 | 258.75 | 384.27 | 343.26 | 102.67 | 278.0 | 238.16 | 15,032.91 | 24,043.45 | 14,486.4 | 4,398.52 | 16,523.94 | 17.13 | 17,749.26 | 27,265.83 | 16,136.32 | 6,343.48 | 18,745.12 | 21.91 |
| 9 | 256.01 | 370.08 | 336.02 | 95.75 | 272.84 | 271.1 | 16,552.14 | 25,708.71 | 15,996.81 | 5,258.16 | 19,245.75 | 17.37 | 17,113.88 | 25,398.23 | 15,806.56 | 6,179.73 | 18,119.65 | 25.69 |
| 10 | 250.58 | 362.4 | 329.83 | 92.5 | 263.71 | 302.9 | 20,138.11 | 31,046.99 | 17,419.01 | 6,116.17 | 20,815.66 | 16.84 | 17,656.92 | 26,038.53 | 16,035.27 | 6,263.67 | 18,505.22 | 28.07 |

the results of the experimentation. This is particularly true for the overall performance of centralized AC enforcement mechanisms (e.g., OPA and XACML) which suffer network latency more than decentralized mechanisms (e.g., `CryptoAC`) as, intuitively, the channel toward centralized services may become a bottleneck for the whole application. For instance, the latency for both AWS[34] and Microsoft Azure[35] Cloud — which naturally vary depending on the selected region and the location of the user — was found to be around 200ms on average in 2021 [37]. Suppose now that an organization deploys OPA — which mainly targets Cloud environments[36] — on AWS or Azure; considering that any AC execution of $W_1$ has 54 state-change rules — and considering one HTTPS request for each rule — only the overhead due to the latency of the Cloud would be more than ten seconds (i.e., 10,800ms = 200ms $\cdot$ 54). Therefore, organizations should consider network latency, as well as all other network characteristics, to obtain results that are representative of a specific scenario.

Then, in our methodology we do not consider the impact that other processes may have in case they are carried out on the same device hosting the AC enforcement mechanism (e.g., `CryptoAC`), possibly leading to scarcity of (computational or network) resources. However, this behavior can be easily simulated by using dedicated tools (e.g., `stress`[37]).

Finally, as we do not consider a specific scenario for the sake of generality, we choose 5 BPMN workflows from the official report of OMG [23] to consider well-designed workflows only. Naturally, these workflows belong to different application fields. As a consequence, the results we obtain in our evaluation are still realistic (as they are built starting from a realistic RBAC state and sequences of state-change rules derived from BPMN workflows) but not representative of any concrete scenario. This issue is indeed solved when an organization uses its own workflows as input to our methodology.

### 7.4. Micro-benchmark Evaluation

We now compare the results of our methodology (presented in Section 7.2) against those of an evaluation based on micro-benchmarks. In this context, we consider the AuthzForce implementation of XACML and `CryptoAC` as AC enforcement mechanisms, since their performance is comparable while OPA outperforms both by far. In detail, we reuse the same experimental settings described in Section 7.1 (e.g., RBAC system, infrastructure, number of clients). However, this time we consider a single state-change rule at a time. In other words, an experiment based on micro-benchmarking consists of a number of clients from 5 to 10 all sending the same state-change rule toward the same mechanism — either XACML or `CryptoAC` — for 10 minutes and then measuring the average response time, similarly to what we do in Section 7.2; this amounts to a total of 72 experiments. As a side note, we highlight that this experimentation demonstrates that it is possible to use our methodology also to run micro-benchmarks that consider a realistic RBAC system and have multiple clients sending state-change rules concurrently, two requirements to obtain significant results, as discussed in Section 4.3.

We report in Table 8 the results of the micro-benchmark evaluation (best times are underlined). As we can see from the table, XACML always performs better than `CryptoAC` in the majority of the considered state-change rules (i.e., 4 out of 6) — regardless of the number of clients. In particular, XACML is faster at applying administrative state-change rules, probably because `CryptoAC` involves the use of (slow) asymmetric cryptography for, e.g., distributing permissions. However, given that in CAC the enforcement of the policy is pre-compiled and embedded into the (distribution of the) secret keys, `CryptoAC` outperforms XACML when evaluating the entailment relation (i.e., read and write on resources). In other words, XACML has to evaluate policy sets and policies for each (user) request sent by all clients (recall the discussion in Section 7.3), while in `CryptoAC` the evaluation is implicit. Looking at Table 8, an organization may then choose XACML over `CryptoAC` because "generally better performing", but it is clear that the real answer actually depends on how frequent read and write actions are; this is an aspect that micro-benchmarks cannot capture, while our methodology can.

## 8. Conclusion and Future Work

In this paper, we presented a methodology for more realistic performance evaluation of AC enforcement mechanisms. First, we devised a procedure, ACE, to derive sequences of state-change rules from BPMN workflows automatically. Then, we designed a simulator tool, ACME, allowing to measure and compare the performance of different mechanisms. Finally, we provided a concrete application of our methodology on three AC enforcement mechanisms — OPA, XACML and `CryptoAC` — to demonstrate its practicality and accuracy. Our experiments show that our methodology provides a robust basis for measuring and comparing the performance of different AC en-

[34] `https://aws.amazon.com/`
[35] `https://azure.microsoft.com/`
[36] `https://www.openpolicyagent.org/`
[37] `http://linux.die.net/man/1/stress`

Table 8: Micro-Benchmarks Average Response Time (ms) per single State-Change Rule from 5 to 10 Concurrent Clients

| State-Change Rule | XACML | | | | | | CryptoAC | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 6 | 7 | 8 | 9 | 10 | 5 | 6 | 7 | 8 | 9 | 10 |
| addResource($\cdot$) (Transient Resource) | 14.7 | 15.97 | 17.69 | 20.17 | 76.28 | 30.44 | 618.19 | 638.68 | 674.03 | 643.22 | 672.51 | 921.24 |
| $\vdash (\langle \cdot, \langle \texttt{read} \rangle \rangle)$ (Transient/Persistent Resource) | 4595.2 | 5466.61 | 5551.29 | 7110.38 | 5409.9 | 8071.93 | 850.17 | 890.28 | 1033.85 | 792.11 | 544.76 | 350.57 |
| $\vdash (\langle \cdot, \langle \texttt{write} \rangle \rangle)$ (Persistent Resource) | 2866.09 | 3359.65 | 2979.35 | 4323.29 | 3545.29 | 4096.19 | 932.57 | 761.68 | 731.52 | 667.37 | 546.98 | 364.28 |
| deleteResource($\cdot$) (Transient Resource) | 11.45 | 16.06 | 17.65 | 20.24 | 15.97 | 26.88 | 148.32 | 134.12 | 130.07 | 128.91 | 134.28 | 160.14 |
| assignPermissionToRole($\cdot, \cdot$) | 228.23 | 227.79 | 245.0 | 250.19 | 227.32 | 289.2 | 430.66 | 395.67 | 351.31 | 334.57 | 294.4 | 291.44 |
| revokePermissionFromRole($\cdot, \cdot$) | 230.55 | 279.42 | 297.0 | 298.0 | 292.44 | 297.5 | 652.1 | 565.63 | 540.31 | 479.98 | 406.04 | 338.22 |

forcement mechanisms realistically, being even able to capture the benefits — in terms of scalability — of a decentralized mechanism (i.e., `CryptoAC`) over a centralized one (i.e., XACML). Also, we compared our methodology with microbenchmarking, showing how the latter — although still useful to assess single functionalities of a mechanism from the developers' point of view — is not capable of returning realistic performance evaluation results from the users' point of view, and may thus not always suggest the best mechanism for a given scenario. We make the implementation of both ACE[38] and ACME[39] open-source and freely available.

### 8.1. Future Work

Natural improvements to our methodology consist in allowing more fine-grained customization of the evaluation (e.g., tuning default activities input/output) and implementing other adapters for ACME, so to support further AC enforcement mechanisms out-of-the-box. In any case, we remark that an organization can easily implement adapters for further mechanisms by following the instructions in the engine module of ACME; essentially, an organization just needs to specify what APIs to invoke (e.g., to what URLs send the HTTPS requests and what parameters to provide) to implement a certain state-change rule (e.g., add resource) and the credentials to use for authentication (e.g., username and password). Also, our methodology currently allows for conducting *performance* evaluations of AC enforcement mechanisms. Hence, it would be interesting to extend our methodology for conducting *security* evaluations. For instance, we could incorporate a reference RBAC implementation as an oracle to validate the functional correctness of both ACME and the AC enforcement mechanism under test. Additionally, we could extend our selection of sequences of AC requests to specifically target corner cases, considering both the entailment relation and administrative operations as testing criteria. Moreover, we are planning to expand the set of BPMN symbols supported by ACE and parse more workflows; this would allow the creation of a library of workflows that organizations (as well as researchers) can readily use to evaluate the performance of their AC enforcement mechanisms — acting therefore as a reference benchmarking solution. Another interesting research direction is extending our methodology to support the evaluation of ABAC enforcement mechanisms as well. At the conceptual level, we note that ACE could support ABAC already, except for the derivation of state-change rules for ABAC from WF net executions (step 4 in Figure 1 — see the conceptual map in Table 4). Finally, it would be interesting to correlate the idling time of workflows (discussed in Section 6.2) with the concept of AC policy resiliency [32] and, more in general, the workflow satisfiability problem [52]. In this regard, we could also include the specification of (dynamic and static) separation of duty constraints [30] into ACE by, e.g., introducing ad-hoc state-change rules. For instance, given two permissions $p_1$ and $p_2$ assigned to two roles $r_1$ and $r_2$, respectively, and a separation of duty constraint saying that the same user $u$ cannot assume $r_2$ to execute $p_2$ if $u$ already assumed $r_1$ to execute $p_1$, we could introduce a special state-change rule that revokes $u$ from $r_2$ whenever $u$ executes $p_1$ using $r_1$.

[38] `https://github.com/stfbk/ACE/`

[39] `https://github.com/stfbk/ACME/`


## Acknowledgments

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU and by Ministero delle Imprese e del Made in Italy (IPCEI Cloud DM 27 giugno 2022 – IPCEI-CL-0000007) and European Union (Next Generation EU).


## References


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

[2] Shayan Ahmadi, Mohammad Nassiri, and Mohsen Rezvani. XACBench: a XACML policy benchmark. *Soft Computing*, 24(21):16081–16096, 2020.

[3] Yosef Ashibani and Qusay H. Mahmoud. Cyber physical systems security: Analysis, challenges and solutions. *Computers & Security*, 68:81–97, 2017.

[4] Stefano Berlato. *A Security Service for Performance-Aware End-to-End Protection of Sensitive Data in Cloud Native Applications*. Phd thesis, University of Genoa, May 2024. Available at `https://hdl.handle.net/11567/1174596`.

[5] Stefano Berlato, Roberto Carbone, Adam J. Lee, and Silvio Ranise. Formal modelling and automated trade-off analysis of enforcement architectures for cryptographic access control in the cloud. *ACM Transactions on Privacy and Security*, 25(1):1–37, 2022.

[6] Stefano Berlato, Roberto Carbone, and Silvio Ranise. Cryptographic enforcement of access control policies in the cloud: Implementation and experimental assessment. In *Proceedings of the 18th International Conference on Security and Cryptography*, pages 370–381. SECRYPT 2021, 2021.

[7] Stefano Berlato, Umberto Morelli, Roberto Carbone, and Silvio Ranise. End-to-end protection of IoT communications through cryptographic enforcement of access control policies. In Shamik Sural and Haibing Lu, editors, *Data and Applications Security and Privacy XXXVI*, volume 13383, pages 236–255. Springer International Publishing, 2022. Series Title: Lecture Notes in Computer Science.

[8] Achim D. Brucker. Integrating security aspects into business process models. *it - Information Technology*, 55(6):239–246, 2013.

[9] Bernard Butler and Brendan Jennings. Measurement and prediction of access control policy evaluation performance. *IEEE Transactions on Network and Service Management*, 12(4):526–539, 2015.

[10] Bernard Butler, Brendan Jennings, and Dmitri Botvich. Xacml policy performance evaluation using a flexible load testing framework. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 648–650, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Bernard Butler, Brendan Jennings, and Dmitri Botvich. An experimental testbed to predict the performance of XACML policy decision points. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 353–360. IEEE, 2011.

[12] Fangbo Cai, Nafei Zhu, Jingsha He, Pengyu Mu, Wenxin Li, and Yi Yu. Survey of access control models and technologies for cloud computing. *Cluster Computing*, 22:6111–6122, 2019.

[13] Marco Centenaro, Stefano Berlato, Roberto Carbone, Gianfranco Burzio, Giuseppe Faranda Cordella, Roberto Riggio, and Silvio Ranise. Safety-related cooperative, connected, and automated mobility services: Interplay between functional and security requirements. *IEEE Vehicular Technology Magazine*, 16(4):78–88, 2021.

[14] H Cormen, CE Leiserson, RL Rivest, and C Stein. Topological Sort, Introduction to Algorithms, 2009.

[15] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[16] Dulce Domingos, António Rito Silva, and Pedro Veiga. Workflow access control from a business perspective. In *Proceedings of the 4th International Workshop on Pattern Recognition in Information Systems*, pages 18–25. SciTePress - Science and and Technology Publications, 2004.

[17] Alina Ene, William Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert E. Tarjan. Fast exact and heuristic methods for role minimization problems. In *Proceedings of the 13th ACM symposium on Access control models and technologies - SACMAT '08*, page 1. ACM Press, 2008.

[18] Anna Lisa Ferrara, P. Madhusudan, and Gennaro Parlato. Security analysis of role-based access control through program verification. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 113–125. IEEE, 2012.

[19] American National Standard for Information Technology. Role based access control. Standard, American National Standards Institute, Inc., February 2004.

[20] Gang Ma, Kehe Wu, Tong Zhang, and Wei Li. A flexible policy-based access control model for workflow management systems. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, pages 533–537. IEEE, 2011.

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

[22] Leigh Griffin, Bernard Butler, Eamonn de Leastar, Brendan Jennings, and Dmitri Botvich. On the performance of access control policy evaluation. In *2012 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 25–32. IEEE, 2012.

[23] Object Management Group. Bpmn 2.0 by example - version 1.0. Non-normative, Object Management Group, June 2010.

[24] Object Management Group. Business process model and notation (bpmn) - version 2.0.2. Standard, Object Management Group, December 2013.

[25] Lucien Heitz, Julian Andrea Croci, Madhav Sachdeva, and Abraham Bernstein. Informfully - research platform for reproducible user studies. In *18th ACM Conference on Recommender Systems*, pages 660–669. ACM, 2024.

[26] Vincent C. Hu, David F. Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2014.

[27] Omer Malik Ilhan, Dirk Thatmann, and Axel Kupper. A performance analysis of the XACML decision process and the impact of caching. In *2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, pages 216–223. IEEE, 2015.

[28] Charbel Kady, Anna Maria Chedid, Ingred Kortbawi, Charles Yaacoub, Adib Akl, Nicolas Daclin, François Trousset, François Pfister, and Gregory Zacharewicz. IoT-driven workflows for risk management and control of beehives. *Diversity*, 13(7):296, 2021.

[29] Anna A. Kalenkova, Wil M. P. Van Der Aalst, Irina A. Lomazova, and Vladimir A. Rubin. Process mining using BPMN: relating event logs and process models. *Software & Systems Modeling*, 16(4):1019–1048, 2017.

[30] D. Richard Kuhn, Edward J. Coyne, and Timothy R. Weil. Adding attributes to role-based access control. *Computer*, 43(6):79–81, 2010.

[31] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur.*, 9(4):391–420, nov 2006.

[32] Ninghui Li, Qihua Wang, and Mahesh Tripunitara. Resiliency policies in access control. *ACM Transactions on Information and System Security*, 12(4):1–34, 2009.

[33] Fernando Lins, Julio Damasceno, Robson Medeiros, Erica Sousa, and Nelson Rosa. Automation of service-based security-aware business processes in the cloud. *Computing*, 98:847–870, 09 2016.

[34] Fabio Martinelli, Oleksii Osliak, Paolo Mori, and Andrea Saracino. Improving security in industry 4.0 by extending OPC-UA with usage control. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10. ACM, 2020.

[35] Francisco Martins and Dulce Domingos. Modelling IoT behaviour within BPMN business processes. *Procedia Computer Science*, 121:1014–1022, 2017.

[36] Jutta Mülle, Silvia von Stackelberg, and Klemens Böhm. A security language for bpmn process models. Technical Report 9, Karlsruher Institut für Technologie (KIT), 2011.

[37] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. Characterization and analysis of cloud-to-user latency: The case of azure and aws. *Computer Networks*, 184:107693, 2021.

[38] Christoph Ruland and Jochen Sassmannshausen. Access control in safety critical environments. In *2018 12th International Conference on Reliability, Maintainability, and Safety (ICRMS)*, pages 223–229. IEEE, 2018.

[39] Fariza Sabrina and Julian Jang-Jaccard. Entitlement-based access control for smart cities using blockchain. *Sensors*, 21(16):5264, 2021. Publisher: MDPI AG.

[40] Mattia Salnitri, Achim Brucker, and Paolo Giorgini. From secure business process models to secure artifact-centric specifications. *Lecture Notes in Business Information Processing*, 214:246–262, 06 2015.

[41] Mattia Salnitri, Fabiano Dalpiaz, and Paolo Giorgini. Modeling and verifying security policies in business processes. In Ilia Bider, Khaled Gaaloul, John Krogstie, Selmin Nurcan, Henderik A. Proper, Rainer Schmidt, and Pnina Soffer, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 175, pages 200–214. Springer Berlin Heidelberg, 2014. Series Title: Lecture Notes in Business Information Processing.

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

[43] Ravi Sandhu. Access control: principle and practice. *Advances in Computers*, 46:237 – 286, 10 1998.

[44] David Spence, George Gross, Cees de Laat, Stephen Farrell, Leon HM Gommans, Pat R. Calhoun, Matt Holdrege, Betty W. de Bruijn, and John Vollbrecht. AAA Authorization Framework. RFC 2904, August 2000.

[45] Ferucio Laurentiu Tiplea, Corina Bocaneala, and Raluca Chirosca. On the complexity of deciding soundness of acyclic workflow nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(9):1292–1298, 2015.

[46] Sameh Hbaieb Turki, Farah Bellaaj, Anis Charfi, and Rafik Bouaziz. Modeling security requirements in service based business processes. In Ilia Bider, Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Stanisław Wrycza, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 113, pages 76–90. Springer Berlin Heidelberg, 2012. Series Title: Lecture Notes in Business Information Processing.

[47] Fatih Turkmen and Bruno Crispo. Performance evaluation of XACML PDP implementations. In *Proceedings of the 2008 ACM workshop on Secure web services - SWS '08*, page 37. ACM Press, 2008.

[48] Mumina Uddin, Shareeful Islam, and Ameer Al-Nemrat. A dynamic access control model using authorising workflow and task-role-based access control. *IEEE Access*, 7:166676–166689, 2019.

[49] Wil MP Van der Aalst. Structural characterizations of sound workflow nets. *Computing science reports*, 96(23):18–22, 1996.

[50] Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. Verification of workflow task structures: A petri-net-baset approach. *Information Systems*, 25(1):43–69, 2000.

[51] Jacques Wainer, Paulo Barthelmess, and Akhil Kumar. W-RBAC — a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12(4):455–485, 2003.

[52] Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow systems. In Joachim Biskup and Javier López, editors, *Computer Security – ESORICS 2007*, volume 4734, pages 90–105. Springer Berlin Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.

[53] Wei Xu, Jun Wei, Yu Liu, and Jing Li. SOWAC: a service-oriented workflow access control model. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 128–134. IEEE, 2004.

[54] Yixin Jiang, Chuang Lin, Han Yin, and Zhangxi Tan. Security analysis of mandatory access control model. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, volume 6, pages 5013–5018. IEEE, 2004.