Using Provenance for Secure Data Fusion in Cooperative Systems

Clara Bertolissi
Aix-Marseille University
clara.bertolissi@lis-lab.fr

Jerry den Hartog
Eindhoven University of Technology
j.d.hartog@tue.nl

Nicola Zannone
Eindhoven University of Technology
n.zannone@tue.nl

ABSTRACT

In the context of cooperative systems, data coming from multiple, autonomous, heterogeneous information sources, is processed and fused into new pieces of information that can be further processed by other entities participating in the cooperation. Controlling the access to such evolving and variegated data, often under the authority of different entities, is challenging. In this work, we identify a set of access control requirements for multi-source cooperative systems and propose an attribute-based access control model where provenance information is used to specify access constraints that account for both the evolution of data objects and the process of data fusion. We demonstrate the feasibility of the proposed model by showing how it can be implemented within existing access control mechanisms with minimal changes.

CCS CONCEPTS

• **Security and privacy** → **Access control**; *Formal security models*;

KEYWORDS

Provenance; Data fusion; ABAC

ACM Reference Format:
Clara Bertolissi, Jerry den Hartog, and Nicola Zannone. 2019. Using Provenance for Secure Data Fusion in Cooperative Systems. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3322431.3325100

1 INTRODUCTION

Cooperation has become a key element of modern IT systems. Cooperative systems often require large amounts of information gathered from a variety of sources, possibly controlled by different authorities, to properly function [3, 22]. This information is typically processed and fused to create "new" data objects that can be further processed and shared to provide new services.

Although providing clear benefits to users, organizations and society, cooperation introduces new challenges in the governance of data and especially in the control of its usage. In fact, information gathered by cooperative systems is distributed, i.e. physically located on different hosts, and heterogeneous, i.e. coming from sensors and internal databases as well as from public sources like blogs and social media. Moreover, information sources are typically managed independently and maintained by different organizations that might wish to retain control over their information. The trust of parties, their willingness to collaborate and thus the added value of the collaborative system all depend on offering contributors appropriate control over the evolution of their information.

To ensure a high level of both security and business continuity in cooperative systems, fine-grained access control at the data level is needed. To implement such a level of granularity, one needs to keep track of *(i)* the evolution of data in the cooperative system, *(ii)* the security requirements and policies on these data, and *(iii)* past accesses to the data.

Unfortunately, traditional access control models are not expressive enough to cope with the security needs of multi-source cooperative systems. To address this gap, a range of novel access control models have been proposed in the last years. On the one hand, we can find history-based access control models [1, 6, 9, 19, 21] that account for the execution history (i.e., past accesses) for access decision making. These models have been further extended by exploiting provenance information about data objects and, thus, also accounting for their evolution [2, 10, 14, 20]. On the other hand, we can find policy frameworks for data fusion [5, 23] that aim to determine which access restrictions should be imposed on data objects based on the restrictions on the data used for their creation. However, to date there is no access control model that allows for the specification and enforcement of access constraints accounting for both the evolution of data objects and access restrictions on the underlying data fusion processes. Moreover, the aforementioned works usually provide ad-hoc policy languages to specify provenance-based access constraints. Therefore, these models are not supported by existing implementations, thus hindering their adoption on a large scale.

In this work, we address the complex problem of access control for secure data fusion in cooperative systems. In particular, we present a unified access control framework that relies on data provenance to account for both the evolution and fusion of data objects.

Our contributions can be summarized as follows:

- Based on a motivating scenario in a military domain, we gather the main requirements for the specification of access control policies tailored to multi-source cooperative systems.
- We propose a formalization for the specification of access constraints tailored to multi-source cooperative systems. The proposed language leverages the expressiveness of both attribute-based access control and provenance. In particular, provenance is used not only to take into account the evolution of data objects in access decision making (as done in previous work [2, 10, 14, 20]), but also to determine which policies should be considered when enforcing access restrictions on derived objects.
- We provide the formal semantics of the proposed language as well as of provenance information updates.

SACMAT '19, June 3–6, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6753-0/19/06…$15.00
https://doi.org/10.1145/3322431.3325100

- We demonstrate the feasibility of the proposed framework by showing how it can be implemented within existing access control mechanisms. Specifically, we show how our access control policies can be expressed in XACML [15] without any adaptation of the standard and discuss the machinery needed to resolve provenance-based access constraints.
- We evaluate the proposed model by assessing its expressiveness against the identified requirements for multi-source cooperative systems and comparing it with related work.

The remainder of the paper is structured as follows. The next section presents a motivation example and discusses the challenges that have to be faced in the design of access control systems for multi-source cooperative systems. Section 3 presents our access control model and Section 4 discusses how it can be implemented in XACML. Finally, Section 5 discusses related work and Section 6 concludes the paper and provides directions for future work.

2 MOTIVATING EXAMPLE

This section illustrates the challenges of regulating access to sensitive resources in cooperative systems through a running example within the military domain inspired from [5]. All names and facts introduced in the scenario are purely fictional.

Our scenario is set in Petros, an unstable country controlled by a military dictatorship. In recent years, protests against the dictatorial regime governing Petros have considerably grown. In response to protests, the government has mobilized the army, leading to an escalation of tension. This situation has attracted the attention of the international community and the NATO has created a Joint Task Force (JTF), named ALPHA, with the mission to restore peace in the territory. The JTF is a multinational operation composed of combat and intelligence units belonging to NATO countries, including the Netherlands (NL), Estonia (EE) and Norway (NO). As an illustrative scenario, we focus on a "hearts and minds" campaign within the JTF. A Dutch unit of the JTF is appointed to gather intelligence in a village; before approaching the village, the team leader of the unit needs information about possible riots in the village.

Figure 1 illustrates the process to gather intelligence in order to produce a riot report. The leader of the Dutch unit contacts an intelligence officer of the Norwegian army to get intelligence on the village. The intelligence officer uses an analysis tool to assess the threat of riots in the mission area using information from different sources. Riots are often characterized by crowds of people in a certain area. To this end, the intelligence officer requests a movement report to the Command and Control Centre of the Dutch army. To assess the movement in the area, the latter requests UAV images and motion data collected by UAV and motion sensors deployed in the area by the Estonian army. The intelligence officer also requests a sentiment report from an OSINT analyst of the Norwegian army. The OSINT analyst runs crawlers on open source blogs and tweets concerning the mission area and its inhabitants. The OSINT analyst provides the intelligence officer the sentiment report. The intelligence officer processes the collected information and fuses them in a riot report, which is sent to the team leader.

This scenario shows that international cooperation and orchestration is needed to ensure the success of the mission. In particular, information from heterogeneous sources should be collected and

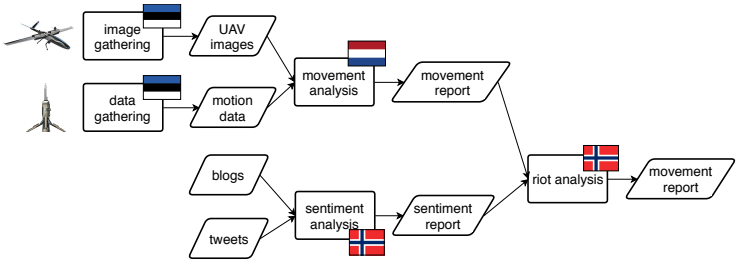

Figure 1: Data flow representing the generation of riot reports

fused to enable situation awareness. Information sources, however, can be under the control of different authorities, and each authority might impose constraints on how and by whom its data can be accessed and processed. For example:

(1) The Norwegian army might require a riot report to be accessed only by users with rank officer that participate to a certain mission and belong to the army of a country that has contributed to the creation of the requested report.
(2) The Norwegian army might also require a riot report to be accessed only by users that are authorized to access all artifacts used for its creation.
(3) The Norwegian army might impose that a riot report is created by a different user from the one who created the sentiment report used as input for the riot report.

Traditional access control models are not expressive enough to capture the access constraints above. We need policies that have "knowledge" about the process used for the creation of the intermediate and final artifacts. To specify those access constraints, we have identified the following criteria that a multi-source cooperation model should encompass:

C1: Multi-Perspective. Access constraints should not only account for the characteristics of the user requesting access (e.g., identity, role, rank, mission) but also for the characteristics of the object to be accessed (e.g., type, security level), for the action to be performed and for the relations between these elements (policy 1).

C2: Artifact Evolution. Access constraints should account for the creation and evolution of objects within the system to determine the access permissions to be granted (policy 1). This includes the ability to constrain access permissions based on users' past actions as well as dynamic Separation and Binding of Duty (SoD/BoD) constraints (policy 3).

C3: Input Objects' Policies. Access constraints should account for the policies associated to the object(s) used for the generation of the requested object (policy 2).

In the next section, we present an access control model able to address the criteria above.

3 ACCESS CONTROL MODEL FOR MULTI-SOURCE COOPERATIVE SYSTEMS

A main property for capturing the access constraints presented in the previous section is the ability to capture the evolution of artifacts. To this end, we rely on the notion of *provenance*, which

Basic casual dependencies
$\text{used}(p:\mathcal{P}, d:\mathcal{D}, r:\mathcal{R})$
$\text{wasGeneratedBy}(d:\mathcal{D}, p:\mathcal{P}, r:\mathcal{R})$
$\text{wasControlledBy}(p:\mathcal{P}, s:\mathcal{S}, r:\mathcal{R})$
$\text{wasTriggeredBy}(p_1:\mathcal{P}, p_2:\mathcal{P}) \triangleq \exists d \in \mathcal{D}, r_1, r_2 \in \mathcal{R} : \text{used}(p_1, d, r_1) \wedge \text{wasGeneratedBy}(d, p_2, r_2)$
$\text{wasDerivedFrom}^{\circ}(d_1:\mathcal{D}, d_2:\mathcal{D}) \triangleq \exists p \in \mathcal{P}, r_1, r_2 \in \mathcal{R} : \text{wasGeneratedBy}(d_1, p, r_1) \wedge \text{used}(p, d_2, r_2)$
Multi-step casual dependencies
$\text{used}^{+}(p:\mathcal{P}, d:\mathcal{D}) \triangleq \exists p_i \in \mathcal{P}, r \in \mathcal{R} : \text{wasTriggeredBy}^{+}(p, p_i) \wedge \text{used}(p_i, d, r)$
$\text{wasGeneratedBy}^{+}(d:\mathcal{D}, p:\mathcal{P}) \triangleq \exists p_i \in \mathcal{P}, r \in \mathcal{R} : \text{wasGeneratedBy}(d, p_i, r) \wedge \text{wasTriggeredBy}^{+}(p_i, p)$
$\text{wasTriggeredBy}^{+}(p_1:\mathcal{P}, p_2:\mathcal{P}) \triangleq \text{wasTriggeredBy}(p_1, p_2) \vee \exists p_i \in \mathcal{P} : \text{wasTriggeredBy}(p_1, p_i) \wedge \text{wasTriggeredBy}^{+}(p_i, p_2)$
$\text{wasDerivedFrom}^{+}(d_1:\mathcal{D}, d_2:\mathcal{D}) \triangleq \text{wasDerivedFrom}(d_1, d_2) \vee \exists d_i \in \mathcal{D} : \text{wasDerivedFrom}(d_1, d_i) \wedge \text{wasDerivedFrom}^{+}(d_i, d_2)$
Custom casual dependencies (not in the Open Provenance model)
$\text{wasDerivedFrom}(d_1:\mathcal{D}, d_2:\mathcal{D}, r:\mathcal{R}) \triangleq \exists p \in \mathcal{P}, r_i \in \mathcal{R} : \text{wasGeneratedBy}(d_1, p, r_i) \wedge \text{used}(p, d_2, r)$
$\text{owns}(s:\mathcal{S}, d:\mathcal{D}) \triangleq \exists p \in \mathcal{P}, r \in \mathcal{R} : \text{wasGeneratedBy}(d, p, r) \wedge \text{wasControlledBy}(p, s, \text{`}owner\text{'})$
$\text{contributedTo}(s:\mathcal{S}, d:\mathcal{D}) \triangleq \exists p \in \mathcal{P} : \text{wasGeneratedBy}^{+}(d, p) \wedge \text{wasControlledBy}(p, s, \text{`}owner\text{'})$

Table 1: Casual dependencies in the provenance graph

provides a traceability system to record the history of artifacts. In this section, we present an access control model for representing provenance-based access constraints. Rather than proposing yet another access control model, we rely on the attribute-based access control (ABAC) paradigm and show how provenance-based access constraints can be accommodated in ABAC policies. Next, we first recall the Open Provenance model [12] that we use to represent provenance information and then we present a model for the specification of access constraints relying on provenance information.

3.1 Provenance Model

To model and reason over provenance information, we adopt the Open Provenance model [12], which provides a core representation of provenance. Here, we present an overview of this model and refer to [12] for an in-depth description.

Provenance information is usually represented as a labeled direct graph (called *provenance graph*) that expresses how a certain object was derived. A node of a provenance graph can be: an *artifact*, which is used to denote an immutable piece of state that can have a physical or digital representation; a *process*, which is used to denote the actions performed on an artifact and resulting in a new artifact; or an *agent*, which is used to denote the entity controlling or affecting the execution of a process. Hereafter, we use $\mathcal{N}$, $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{S}$ to denote the sets of nodes, artifacts, processes and agents, respectively (with $\mathcal{N} = \mathcal{D} \cup \mathcal{P} \cup \mathcal{S}$ and $\mathcal{D}, \mathcal{P}, \mathcal{S}$ pairwise disjoint). Moreover, we use $\mathcal{R}$ to denote the set of role labels, which define the function of an agent or an artifact in a process.[1]

The edges of the provenance graphs capture three types of role labeled casual dependencies (top of Table 1): $\text{used} \subseteq \mathcal{P} \times \mathcal{D} \times \mathcal{R}$ captures, for processes, the artifacts they use; $\text{wasGeneratedBy} \subseteq \mathcal{D} \times \mathcal{P} \times \mathcal{R}$ captures which artifacts are generated by which processes; $\text{wasControlledBy} \subseteq \mathcal{P} \times \mathcal{S} \times \mathcal{R}$ captures, for processes, which agents control them. To consider a specific role or ignore the role label of a casual dependency T, we respectively use $T^{r}(n, n') \triangleq T(n, n', r)$, where r is a role label, and $T^{\circ}(n, n') \triangleq \exists r \in \mathcal{R} : T(n, n', r)$ (for all $n, n' \in \mathcal{N}$).

The Open Provenance model includes several additional dependencies that can be derived from a provenance graph by composing dependencies (we use ";" to denote composition of casual dependencies, i.e. $T_i; T_j(n, n') \triangleq \exists x \in \mathcal{N} : T_i(n, x) \wedge T_j(x, n')$) while ignoring the roles: $\text{wasTriggeredBy} \triangleq \text{used}^{\circ}; \text{wasGeneratedBy}^{\circ}$ captures that the execution of a process was triggered by another process and $\text{wasDerivedFrom}^{\circ} \triangleq \text{wasGeneratedBy}^{\circ}; \text{used}^{\circ}$ captures that an artifact was derived from another artifact. Transitive closure $\text{wasDerivedFrom}^{+}$ captures, for an artifact, all artifacts used to derive it, possibly indirectly. Similarly, $\text{wasTriggeredBy}^{+}$ captures, for a process, all its direct and indirect triggering processes. The later is also helpful to capture indirect use and generation of artifacts: $\text{used}^{+} \triangleq \text{wasTriggeredBy}^{+}; \text{used}^{\circ}$ captures, for a process, all artifacts it indirectly depends on, whereas $\text{wasGeneratedBy}^{+} \triangleq \text{wasGeneratedBy}^{\circ}; \text{wasTriggeredBy}^{+}$ captures, for an artifact, all processes leading up to its generation.

Aiming to use provenance information to specify access requirements, we extend the Open Provenance model. For an artifact, we want to be able to refer to the role another artifact played in its creation. We thus label the Open Provenance model relation $\text{wasDerivedFrom}^{\circ}$ by making the role explicit: $\text{wasDerivedFrom}^{r} \triangleq \text{wasGeneratedBy}^{\circ}; \text{used}^{r}$ (for all $r \in \mathcal{R}$). We also introduce two custom dependencies that use a specific role owner $\in \mathcal{R}$ to capture artifact owners and contributors. Following the classic assumption in discretionary access control where subjects own the objects they create [7], we say an owner of a process that generates an artifact owns that artifact: $\text{owns} \triangleq \text{wasGeneratedBy}^{\circ}; \text{wasControlledBy}^{\text{owner}}$. Actors owning any process indirectly involved in the creation of an artifact are considered contributors: $\text{contributedTo} \triangleq \text{wasGeneratedBy}^{+}; \text{wasControlledBy}^{\text{owner}}$. These relations are formally defined in Table 1. We will see in the next section how standard and custom causal dependencies can be exploited to define access constraints in the form of *path conditions*.

A provenance graph can be visually represented. Figure 2 shows the provenance graph for the scenario presented in Section 2. Following the graphical notation defined in [12, 13], we represent artifacts using circles, processes using rectangles and agents using

[1] Note that the notion of *role* in provenance should not confused with the notion of role in role-based access control (RBAC) where *role* indicates the job function of users within an organization. Although the distinction between these two concepts should always be clear from the context, to avoid confusion, hereafter we will use attribute subject_role to refer to RBAC roles.

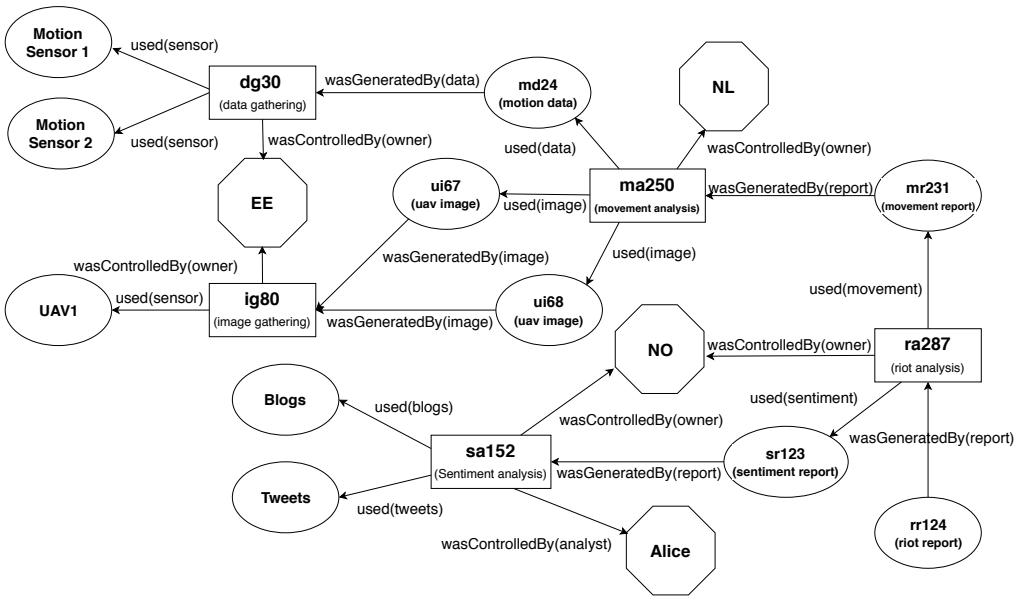

Figure 2: Provenance graph corresponding to the motivating example

octagons. Edges are annotated with the type of dependency and the role of artifacts and agents in processes. It is worth noting that the nodes in a provenance graph represent instances of agents, artifacts and processes. For instance, Figure 2 represents that: motion data *md*24 have been gathered from *motion sensor 1* and *motion sensor 2* through the instance of the data gathering process *dg*30; riot report *rr*124 has been generated from movement report *mr*231 and sentiment report *sr*123 through the instance of process riot analysis *ra*287; and so on. For the sake of clarity, in the figure the type of artifacts and processes is reported in parenthesis.

3.2 Policy Representation

In ABAC, policies and access requests are defined in terms of attribute name-value pairs. Formally, let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a finite set of attributes. Given an attribute $a \in \mathcal{A}$, $\mathcal{V}_a$ denotes the domain of a. The set of queries $\mathcal{Q}_\mathcal{A}$ is defined as $\wp(\bigcup_{i=1}^{n} a_i \times \mathcal{V}_{a_i})$ and a query $q = \{(a_1, v_1), \ldots, (a_k, v_k)\}$ is a set of attribute name-value pairs (a_i, v_i) such that $a_i \in \mathcal{A}$ and $v_i \in \mathcal{V}_{a_i}$. We also assume an infinite set of variables. Each variable x has a domain $\mathcal{V}_x \subseteq \mathcal{N}$.

Example 1. *The access request made by the team leader of the Dutch unit can be modeled as query*

$$q = \{(\text{subject\_id}, \textit{Paul}), (\text{subject\_rank}, \textit{officer}), (\text{subject\_army}, \textit{NL}), (\text{subject\_mission}, \textit{alpha}), (\text{action}, \textit{read}), (\text{resource\_id}, \textit{rr124}), (\text{resource\_type}, \textit{riot\_report})\}$$

*This query states that Paul, an officer of the Dutch army participating to mission ALPHA, wants to read the riot report identified by code rr*124*.*

For policy specification, we adopt a language $\mathcal{P}ol_\mathcal{A}$ that provides a compact representation of XACML [15] with the addition of path conditions. Formally, a policy $pol \in \mathcal{P}ol_\mathcal{A}$ (a target expression $t \in \mathcal{T}arExp_\mathcal{A}$, a path condition $path \in \mathcal{P}ath_\mathcal{A}$ and a node expression $node \in \mathcal{N}odeExp_\mathcal{A}$) is defined as:

$$\begin{aligned} pol &= 1 \mid 0 \mid (t, pol) \mid ca(pol_1, \ldots, pol_n) \mid \\ &\quad ca(\{pol\_ref(x_1, \ldots, x_k) \mid path\}) \\ t &= op(a, v) \mid path \mid \top \mid \bot \mid \neg t \mid t \wedge t \mid t \vee t \\ path &= T^{\circ}(node, node) \mid T^{r}(node, node) \mid \neg path \mid path \wedge path \mid \\ &\quad path \vee path \mid \forall x : path \mid \exists x : path \\ node &= n \mid x \mid a \mid self \\ ca &= \text{pov} \mid \text{dov} \mid \text{fa} \end{aligned}$$

A target expression t is a Boolean expression built in standard way over primitive target expressions using Boolean operators (e.g., $\neg$, $\wedge$, $\vee$). A primitive target expression can be *true* ($\top$), *false* ($\bot$), a Boolean expression of the form $op(a, v)$ with $a \in \mathcal{A}$, $v \in \mathcal{V}_a$ and $op \in \{=, \neq, >, <\}$, or a path condition *path* over the provenance graph. A path condition can be a casual dependency in the provenance graph (as the ones defined in Table 1), or can be constructed from casual dependencies using logical operators (e.g., $\neg$, $\wedge$, $\vee$). In quantified path conditions, i.e. $\forall x$ *path* and $\exists x$ *path*, x is a variable occurring in *path*. When an attribute is used as an argument of a path condition, we assume it is instantiated by its value as given in the access request and this value corresponds to a node in the provenance graph.[2] In addition, we use a special attribute *self* to indicate the current instance of the requested action. This is needed because the attributes concerning actions used in policies and access requests typically refer to action 'types' rather than to instances. (See also Example 4 below.)

A policy can be a decision, either *permit* (1) or deny (0), a target policy (t, pol) where the target t defines the applicability of the policy, or a composite policy $ca(pol_1, \ldots, pol_n)$ with ca a combining algorithm. Here, we consider standard combining algorithms [15]: *permit-overrides* (pov), *deny-overrides* (dov) and *first-applicable*

[2]Recall that the nodes in a provenance graph correspond to artifact, process and agent instances.

(fa). A policy reference *pol_ref* is an expression that resolves into a *uri* pointing at a policy. We do not detail here how to specify and resolve policy references but we simply assume that a policy reference *pol_ref* yields a function $uri : \mathcal{N}^k \rightarrow \mathcal{P}ol_{\mathcal{A}}$. Note that $uri(x_1, \ldots, x_n)$ is considered to bind variables $x_1, \ldots, x_n$ in *path*. Since the number of the referred policies is not know a priori, we always combine those policies using a combining algorithm *ca*.[3]

We now illustrate policy references through an example.

Example 2. *Consider a policy involving the entities in the provenance graph of Figure 2 that includes policy reference*

$$\mathsf{dov}(\{uri(x, y) \mid \mathsf{wasDerivedFrom}^{+}(\mathsf{resource\_id}, x) \wedge \mathsf{owns}(y, x)\})$$

This policy reference can be resolved with a function

$$uri(x, y) = \mathsf{uri{:}policies.alpha.jtf.nato.org?resource\_id} = x \;\&\mathsf{agent} = y$$

which encodes a Uniform Resource Identifier (URI) that identifies the policies applicable to the nodes in the provenance graph that satisfy the given query. The string after the question mark (?) represents the query component of the URI.

Given an applicable request {(resource_id, *mr231*), . . .} *for movement report mr231, the policy reference resolves to*

$$\mathsf{dov}(\{uri(md24, EE), uri(ui67, EE), uri(ui68, EE)\})$$

which becomes $\mathsf{dov}(\overline{pol_{md}}, \overline{pol_{ui}})$ *where* pol_{md} *is the EE policy applicable to motion data and* pol_{ui} *is the EE policy applicable to its UAV gathered images (notation* $\overline{pol}$ *is explained below).*

For a request {(resource_id, *rr124*), . . .}*, it would also consider the NL policy that applies to mr231 and the NO policy that applies to sr231.*

Note that, when referring to an external policy, we have to ensure that such a policy applies to the given access request. As shown in [5], the use of attributes characterizing the object and action in the target can be problematic. To this end, we assume that the referred policy is stripped of constraints involving those attributes and, given a policy pol_d associated to an artifact d, $\overline{pol}_d$ denotes pol_d without constraints involving object and action attributes. We refer to [5] for the procedure for this policy transformation.

The policy language $\mathcal{P}ol_{\mathcal{A}}$ allows the definition of three main types of access constraints:

(1) *attribute-based constraints* that define permissions and prohibitions based on subject, action and object attributes, thus accounting for multi-perspectives in access decision making (**C1**);
(2) *provenance-based constraints* that define permissions and prohibitions based on path conditions over the provenance graph, thus accounting for the evolution of artifacts (**C2**);
(3) *input-dependent constraints* that define permissions and prohibitions based on the policies of the artifacts and processes used (possibly indirectly) in the generation of new artifacts (**C3**). These constraints are modeled using policy references pointing to (external) access control policies identified though a path condition in the provenance graph.

[3]Note that the order of the referred policies cannot be predefined. Therefore, we assume that only combining algorithms *permit-overrides* and *deny-overrides* can be used together with policy references.

In the following example, we demonstrate how policy language $\mathcal{P}ol_{\mathcal{A}}$ can be used to model the access control policies underlying the scenario in Section 2.

Example 3. *The access control policy* p_{rr} *used in our motivating example to regulate the access to objects of type* riot report *(policies 1 & 2) can be formalized as follows:*

$$\begin{aligned}
pol_{rr} &= (\mathsf{resource_type} = \mathit{riot_report}, \mathsf{dov}(pol_A, pol_B)) \\
pol_A &= \mathsf{pov}(pol_1, pol_2, 0) \\
pol_B &= \mathsf{pov}(pol_3, 0) \\
pol_1 &= (\mathsf{subject_rank} = \mathit{officer} \wedge \\
&\quad\; \mathsf{subject_mission} = \mathit{alpha} \wedge \mathsf{action} = \mathit{read}, 1) \\
pol_2 &= (\mathsf{contributedTo}(\mathsf{subject_army}, \mathsf{resource_id}) \wedge \\
&\quad\; (\mathsf{action} = \mathit{read} \vee \mathsf{action} = \mathit{download}), 1) \\
pol_3 &= \mathsf{dov}(\{uri(x) \mid \mathsf{wasDerivedFrom}^{\circ}(\mathsf{resource_id}, x)\}
\end{aligned}$$

Policy p_{rr} *combines two composite policies using the* deny-overrides *combining algorithm, one* (pol_A) *specifying access constraints specific to riot reports and one* (pol_B) *used to refer to the policies of the object(s) used for its creation. In particular,* pol_A *comprises two policies,* pol_1 *and* pol_2*, and a default* Deny *policy (represented by* 0*).*[4] *The first policy* pol_1 *encodes an attribute-based constraint stating that a subject having rank* officer *in her/his national army and participating to mission* alpha *is allowed to read the riot report. Policy* pol_2 *encodes a provenance-based constraint stating that access to the report is allowed if the subject belongs to (the army of) a country that has contributed to the creation of the riot report. The notion of* contributedTo *is formalized as a path condition over the provenance graph (its formal definition is at the bottom of Table 1), meaning that the country has participated to at least one activity in the whole process leading to the creation of the riot report.*

Policy pol_B *comprises policy* pol_3 *(and a* Deny *default policy). In particular,* pol_3 *specifies an input-dependent constraint in the form of a reference to the (external) policies* pol_x *associated to the artifacts used in the creation of the riot report, in our case movement report* $mr231$ *and sentiment report* $sr123$ *(Figure 2). This reference is defined by a path condition (*wasDerivedFrom*) on the provenance graph taking as arguments the id of the resource from the access query, i.e.* $rr124$*, and variable* x*. The policies associated to the movement and sentiment reports are combined with the* deny-overrides *combining algorithm. Accordingly, the subject is allowed to access the riot report if he can access all artifacts used in the creation of the report.*

It is worth noting that the path condition in pol_3 *refers to the policies of* all *artifacts used in the creation of the riot report. This may be too restrictive in some cases as one may only want to consider the policies associated to input artifacts of a certain type. For instance, in case the access to the riot report should only depend on the policy of the movement report used for its creation, the following policy could be used (instead of* pol_3*):*

$$pol'_3 = \mathsf{dov}(\{uri(x) \mid \mathsf{wasDerivedFrom}^{\mathsf{movement}}(\mathsf{resource\_id}, x)\})$$

In the next example, we show that the proposed language $\mathcal{P}ol_{\mathcal{A}}$ can also be used for the specification and enforcement of dynamic SoD constraints.

[4]Alternatively, one may use *deny-unless-permit* or *permit-unless-deny* combining algorithms defined in XACML v. 3 [15], which account for default policies implicitly.

Example 4. *The Norwegian Army might impose that riot reports should be created by a different analyst from the one who created the sentiment report used as input (policy 3 in the motivating example). This policy can be formalized as follows:*

$$\begin{aligned} pol_{ra} &= (\text{action} = \textit{riot\_analysis}, \text{dov}(pol_4, pol_5)) \\ pol_4 &= (\text{subject\_role} = \textit{analyst} \wedge \text{subject\_army} = \textit{NO}, 1) \\ pol_5 &= (\text{used}^{\text{sentiment}}; \text{wasGeneratedBy}^{\text{report}}; \\ &\quad \text{wasControlledBy}^{\text{analyst}}(\textit{self}, \text{subject\_id}), 0) \end{aligned}$$

*where '*self*' in pol_5 indicates the current instance of process* riot analysis. *Policy pol_5 can be read as "if the current instance of the process (self) uses a sentiment report generated by a process controlled by the analyst that is currently requesting access (*subject_id*), then access is denied".*

3.3 Policy Evaluation

In this section, we define the semantics of the policy language $\mathcal{P}ol_{\mathcal{A}}$ introduced in Section 3.2. Given the set of policies $\mathcal{P}ol_{\mathcal{A}}$, the set of queries $\mathcal{Q}_{\mathcal{A}}$ and a set of decisions $\mathcal{D}$, a policy evaluation function is a function $[\![\cdot]\!]_P : \mathcal{P}ol_{\mathcal{A}} \times \mathcal{Q}_{\mathcal{A}} \to \mathcal{D}$ such that, given a query q and a policy *pol*, its semantics $[\![pol]\!]_P(q)$ represents the decision of evaluating *pol* against q. For the sake of simplicity, here we assume $\mathcal{D} = \{Permit, Deny, NA\}$ (where *NA* stands for Not Applicable) but the provided semantics can be easily extended to deal with the *Indeterminate* decision supported by XACML [15]. Before formally defining the semantics for policy evaluation, we discuss a special case concerning the execution of new processes in the system. In order to make the query evaluation possible, we need to (temporary) extend the provenance graph with an instance of the requested process. In the next paragraph, we detail this particular case.

A case requiring special attention is the execution of processes, as this leads to extension of the provenance graph. On the one hand, whenever a process (e.g., $ra287$) produces some output artifact (e.g., $rr124$), this is recorded in the provenance graph: a new node is added to the provenance graph for this artifact and linked to the process using the wasGeneratedBy relation, using a role corresponding to the type of output (e.g., *report*). On the other hand, when the execution of a new process is requested, the policy may need to refer to the process instance (by using *self*) before its actual execution. As such, we first extend the graph with a process node linked to its parameters and then evaluate the policy over this extended graph. For example, an access request to execute riot analysis will trigger a call $exec(process = riot\_analysis, owner = NO, sentiment = sr123, movement = mr231)$. Upon receiving such a call, the provenance graph is extended with a new process node (e.g., $ra287$) and links matching the given arguments (e.g., used(*sentiment*) link to $sr123$ and used(*movement*) link to $mr231$, wasControlledBy(*owner*) link to *NO*) are added to the graph. If the decision is to permit the execution of the process (i.e., the user is authorized to execute the process and to use the objects taken as input by the process), this extension is kept; otherwise the extension is discarded after policy evaluation. Technically, this means we should extend the evaluation function $[\![\cdot]\!]_P$ with arguments for the provenance graph and the additional process node p_{self}. However, we suppress them here for sake of readability.

Given a request $q \in \mathcal{Q}_{\mathcal{A}}$ and added process node p_{self} (if needed) to the provenance graph, we evaluate node expressions to sets of nodes using the evaluation function $[\![\cdot]\!]_N : \mathcal{N}odeExp_{\mathcal{A}} \to \wp(\mathcal{N})$ defined as follows:

$$\begin{aligned} [\![n]\!]_N(q) &= \{n\} \\ [\![x]\!]_N(q) &= \emptyset \\ [\![a]\!]_N(q) &= \{n \in V_a \mid (a, n) \in q\} \\ [\![self]\!]_N(q) &= \{p_{self}\} \end{aligned}$$

Path conditions evaluate to Booleans using the evaluation function $[\![\cdot]\!]_C : \mathcal{P}ath_{\mathcal{A}} \to \{True, False\}$ defined as follows:

$$[\![T^{\circ}(node, node')]\!]_C(q) = \begin{cases} True & \text{if } \exists n \in [\![node]\!]_N(q), \\ & n' \in [\![node']\!]_N(q) : \mathbf{T}^{\circ}(n, n') \\ False & \text{otherwise} \end{cases}$$

$$[\![T^{r}(node, node')]\!]_C(q) = \begin{cases} True & \text{if } \exists n \in [\![node]\!]_N(q), \\ & n' \in [\![node']\!]_N(q) : \mathbf{T}^{r}(n, n') \\ False & \text{otherwise} \end{cases}$$

$$\begin{aligned} [\![\neg path]\!]_C(q) &= \neg [\![path]\!]_C(q) \\ [\![path \wedge path']\!]_C(q) &= [\![path]\!]_C(q) \wedge [\![path']\!]_C(q) \\ [\![path \vee path']\!]_C(q) &= [\![path]\!]_C(q) \vee [\![path']\!]_C(q) \\ [\![\exists x : path]\!]_C(q) &= \exists n \in V_x : [\![path[n/x]]\!]_C(q) \\ [\![\forall x : path]\!]_C(q) &= \forall n \in V_x : [\![path[n/x]]\!]_C(q) \end{aligned}$$

Targets evaluate to Booleans using the evaluation function $[\![\cdot]\!]_T : \mathcal{T}arExp_{\mathcal{A}} \to \{True, False\}$ defined as follows:

$$\begin{aligned} [\![\top]\!]_T(q) &= True \\ [\![\bot]\!]_T(q) &= False \\ [\![op(a, v)]\!]_T(q) &= \begin{cases} True & \text{if } \exists v' \in V_a : (a, v') \in q \wedge \boldsymbol{op}(v', v) \\ False & \text{otherwise} \end{cases} \\ [\![\neg t]\!]_T(q) &= \neg [\![t]\!]_T(q) \\ [\![t \wedge t']\!]_T(q) &= [\![t]\!]_T(q) \wedge [\![t']\!]_T(q) \\ [\![t \vee t']\!]_T(q) &= [\![t]\!]_T(q) \vee [\![t']\!]_T(q) \\ [\![path]\!]_T(q) &= [\![path]\!]_C(q) \end{aligned}$$

Finally, policies evaluate to a decision in $\mathcal{D}$ as follows:

$$\begin{aligned} [\![1]\!]_P(q) &= Permit \\ [\![0]\!]_P(q) &= Deny \\ [\![(t, pol)]\!]_P(q) &= \begin{cases} [\![pol]\!]_P(q) & \text{if } [\![t]\!]_T(q) = True \\ NA & \text{otherwise} \end{cases} \\ [\![ca(pol_1, \ldots, pol_m)]\!]_P(q) &= \mathbf{ca}([\![pol_1]\!]_P(q), \ldots, [\![pol_m]\!]_P(q)) \\ [\![ca(\{pol\_ref(x_1, \ldots, x_k) \mid path\})]\!]_P(q) &= \\ & \mathbf{ca}([\![pol_1]\!]_P(q), \ldots, [\![pol_m]\!]_P(q)) \end{aligned}$$

where

$$\{pol_1 \ldots, pol_m\} = \{uri(n_1, \ldots, n_k) \mid n_1 \in \mathcal{V}_{x_1}, \ldots, n_k \in \mathcal{V}_{x_k}, [\![path[n_1/x_1, \ldots, n_k/x_k]]\!]_C(q) = True\}$$

Here $[n/x]$ denotes substituting n for every free occurrence of x. Note that we use $\mathbf{X}$ to distinguish a semantical operator from its (syntactical) representation X. Outside this section we do not make this distinction explicit.

p_1	p_2	pov(p_1, p_2)	dov(p_1, p_2)	fa(p_1, p_2)
Permit	*Permit*	*Permit*	*Permit*	*Permit*
Permit	*Deny*	*Permit*	*Deny*	*Permit*
Permit	*NA*	*Permit*	*Permit*	*Permit*
Deny	*Permit*	*Permit*	*Deny*	*Deny*
Deny	*Deny*	*Deny*	*Deny*	*Deny*
Deny	*NA*	*Deny*	*Deny*	*Deny*
NA	*Permit*	*Permit*	*Permit*	*Permit*
NA	*Deny*	*Deny*	*Deny*	*Deny*
NA	*NA*	*NA*	*NA*	*NA*

Table 2: Semantics of policy combining algorithms

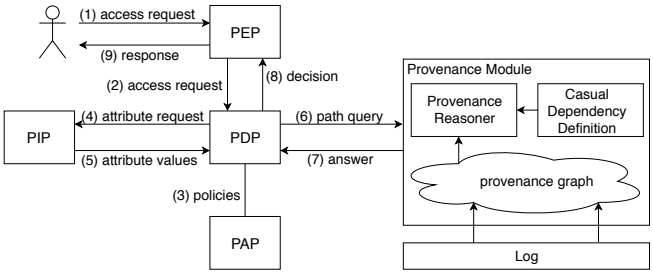

Figure 3: Architecture of the provenance-based access control mechanism

When a variable is used as a node it should always be bound. If a free variable x occurs, we assume that it does not match any node. Node expression *self* refers to the added process node p_{self}. Node expressions could match multiple nodes, as an attribute a may take multiple values in q. Specifically, attributes are instantiated to the nodes corresponding to the attribute values in the query. We evaluate casual dependencies $\mathbf{T}^{\circ}$ and $\mathbf{T}^{r}$ on the extended graph, considering any possible value for the node expressions. Similarly for $op(a, v)$, we look for any matching value of a in the request q. The logical operators (in paths and targets) have their usual interpretation. Policy (t, pol) is not applicable if t does not evaluates to *True*; otherwise its decision is that of pol. Policy $ca(pol_1, ..., pol_m)$ is evaluated by executing the combining algorithm **ca** on the decisions $[\![pol_1]\!]_P(q)$ through $[\![pol_m]\!]_P(q)$. The semantics of combining algorithms is given in Table 2 (here we only show how to combine two policies, but the semantics can be trivially extended to consider an arbitrary number of policies). To evaluate $ca(\{pol\_ref(x_1, ..., x_k) \mid path\})$ we find any nodes for $x_1, \ldots, x_k$ that make the path evaluate to *True*, get the corresponding policies using the policy reference function *uri* and combine the decisions for these policies with **ca**. Recall that the order of the policies does not matter for the combining algorithms we allow here.

4 POLICY ENFORCEMENT IN XACML

In this section we demonstrate how the proposed access control model can be accommodated within existing access control mechanisms. In particular, we discuss how it can be implemented in eXtensible Access Control Markup Language (XACML) [15], the de facto standard for the specification and enforcement of ABAC policies. Next, we first present an overview of the architecture and, then, we discuss implementation related aspects of our access control model and of the management and reasoning on provenance information.

4.1 Architecture Overview

For the evaluation and enforcement of provenance-based and input-dependent constraints, we conservatively extend the XACML reference architecture. An overview of the architecture is presented in Figure 3.

The XACML reference architecture (on the left of Figure 3) consists of four main components:[5] the *Policy Enforcement Point* (PEP), which intercepts access requests from users and enforces authorization decisions; the *Policy Decision Point* (PDP), which is responsible to evaluate access requests against the given policies; the *Policy Administration Point* (PAP), which maintains a policy repository and provides the policies to the PDP for the evaluation of access requests; the *Policy Information Point* (PIP), which provides the PDP with the information necessary for policy evaluation.

In addition to the XACML architecture components, we introduce a *Provenance Module* that encompasses: *(i)* the provenance graph generated from the system logs, *(ii)* the definition of casual dependencies, i.e. capturing Table 1, and *(iii)* a provenance reasoner for evaluating path queries, which are path conditions in which attributes have been instantiated to the value they take in the request. The Provenance Module is invoked whenever the PDP encounters a path condition to evaluate for the current request in the policies.

Now we describe the interaction flow as illustrated in Figure 3. The user's access request is intercepted by the PEP (1). The request (in the form of a set of attribute name-value pairs) is forwarded to the PDP (2). The PDP loads the policies from the PAP (3). The PDP may ask the PIP for additional information about the subject or object, if some subject or object attributes are missing in the request (4,5).

If the evaluation of a policy requires evaluating a path condition, the PDP asks the Provenance Module to resolve the corresponding path query (6,7). Recall from Section 3 that our framework supports two kinds of path conditions: *(a)* Boolean path conditions (e.g., contributedTo(subject_army, resource_id) like in policy pol_2) and *(b)* policy reference path conditions (e.g., pol_3 of our motivating example). In the first case, the Provenance Module takes the path query as input and returns *true* if there exists a path in the provenance graph that matches the path query; otherwise, it returns *false*. Path queries representing policy references are resolved through a two-step process: first, the module runs a path matching algorithm to return all entities that satisfy the path query in the provenance graph, and second the policies associated to these entities are retrieved and returned for evaluation. Steps 6 and 7 may be repeated in case of nested policies.

[5] The XACML reference architecture also includes a *Context Handler*, which is responsible to coordinate the other components. We omit this component here for the sake of simplicity.

```
<Apply FunctionId="urn:nl:tue:sec:provenance:def:function:contributedTo">
  <SubjectAttributeDesignator
    AttributeId="urn:nl:tue:sec:example:subject:army"
    DataType="http://www.w3.org/2001/XMLSchema#string"/>
  <ResourceAttributeDesignator
    AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
    DataType="http://www.w3.org/2001/XMLSchema#string"/>
</Apply>
```

Listing 1: User-defined function to resolve path condition contributedTo(subject_army, resource_id) **in** pol_2

Once all relevant information has been gathered and path conditions have been evaluated, the PDP makes a decision and returns it to the PEP (8). The last step (9) concerns the enforcement of the access decision. If the answer is positive, the PEP authorizes the request, otherwise the PEP throws an access denied exception.

4.2 Implementation & Deployment

This section describes how existing access control mechanisms can be adapted to implement the access control framework presented in this work. We assume an existing XACML installation and discus how the language $\mathcal{P}ol_{\mathcal{A}}$ can be encoded and the new architectural component – the Provenance Module – implemented and incorporated to support the evaluation of provenance-based and input-dependent constraints. We then evaluate the effort needed and the impact the adaption has on this existing installation.

Encoding $\mathcal{P}ol_{\mathcal{A}}$ in XACML. The policy language $\mathcal{P}ol_{\mathcal{A}}$ provides an abstraction of the XACML policy specification language and, thus, most of the constructs provided by $\mathcal{P}ol_{\mathcal{A}}$ can be trivially encoded using the policy elements provided by the XACML standard. Path conditions can also be encoded in XACML policies using standard XACML elements. When path conditions are used as policy references, they can be specified in XACML policies using element `<PolicyIdReference>`. When path conditions are used as Boolean expressions, they can be encoded as user-defined functions. Listing 1 provides an example of how path conditions can be encoded in XACML policies using user-defined functions.

Provenance Module. We have implemented the Provenance Module in Prolog [11]. This allows us to benefit from the efficient reasoning capabilities provided by Prolog solvers for path condition resolution rather than relying on an ad-hoc algorithm. In particular, we implemented rules both for constructing the provenance graph from a given access log and for resolving path queries based on the obtained provenance graph. Intuitively, the first set of rules are used to infer basic casual dependencies (i.e., used, wasGeneratedBy and wasControlledBy) from the access log, whereas the latter is used to derive the other standard and custom casual dependencies as defined in Table 1.

Path queries are verified by the Provenance Module through Prolog queries. Upon receiving a request to resolve a path query from the PDP, the Provenance Module constructs the corresponding Prolog query (i.e., the goal to be proved) and interacts with the underlying Prolog solver, which uses its knowledge of deduction rules (i.e., the program consisting of the rules for constructing casual dependencies) to prove or to refute the goal. When a path condition is used for policy referencing, the correspond query might contain unbounded variables. In this case, Prolog solvers enumerate all the values of these variables (which corresponds to nodes in the provenance graph) that satisfy the query. The answers to the query are used to fetch the relevant policies from the PAP, which are evaluated by the PDP to determine whether access should be granted or denied.

Several deployments of our framework are possible, in particular concerning the storage and management of provenance information. The scenario presented in Section 2 is characterized by the presence of multiple authorities and entities involved in the processing and protection of data. In order to guarantee the availability and integrity of provenance information in such a distributed environment, one can imagine that access logs are stored in a private blockchain (visible only to the participants of the coalition). In this setting, the Provenance Module looks up into the blockchain to retrieve the information it needs to create the provenance graph and resolve path conditions.

Impact on existing installations. The XACML standard allows for extensions through user-defined functions and supports policy references. As such, the enforcement approach presented in this section uses standard XACML principles and mechanisms. How these features are implemented, however, is outside the scope of the XACML specification [15, Section 5.11]. Therefore, we also need to consider the actual XACML implementation. Adding support for the evaluation provenance-based and input-dependent constraints to existing implementations through the approach described above requires that the implementation supports evaluating user defined functions and policy references using an external service.

Using external services to evaluate user-defined functions has already shown feasible and implementable in different commonly used XACML implementations, e.g. [8]. That work introduces a framework offering a number of extension points, including the use of user-defined functions that are realized as external services. In our context, each casual dependency can be implemented as a user-defined function that invokes the Provenance Module for its resolution. The advantage of decoupling the implementation of the mechanism for the resolution of path conditions from the PDP is that this approach does not require modification of the XACML implementation and, thus, can be easily extended to support custom casual dependencies in a transparent way w.r.t. the other components.

In addition, gathering of provenance information can be limited to resources for which it is relevant. Finally, and perhaps most

importantly, by encoding path conditions through user-defined functions (or through element `<PolicyIdReference>` depending on how path conditions are used), existing policies do not need to be changed at all. Thus, our approach can be added to existing systems incrementally and with minimal disruption.

5 RELATED WORK

Our work is related to history-based access control, provenance-based access control and access control for data fusion. A summary of related work against the criteria identified in Section 2 is presented in Table 3.

The past history of the system is at the core of authorization decision making in our model. This is also the main feature of history-based access control systems, which can be grouped into two main families: dynamic history-based access control systems in which programs are monitored at run-time [9, 21] and static history-based access control systems in which one (statically) verifies that (an approximation of) the program's behavior will always conform to the policy (using, e.g., type systems or model checking) [1, 19]. Our approach is more closely related to the first type as it uses provenance information for decision making. Krukow et al. [9] present a logical framework for behavior-based decision-making in which policies define access requirements on the past behavior of principals that must be fulfilled in order for an action to be authorized. The framework comprises a formal model of behavior, based on event structures, a declarative logical language for specifying properties of past behavior and an automata-based algorithm for checking whether a particular behavior satisfies a property expressed in the given language. The notion of security automata was firstly introduced in [18], where the policy is given in terms of an automata and safety properties are enforced using a (automata-based) reference monitor that tracks execution history. Automata are also adopted in [21] where access control policies prescribe constraints on the order in which permissions should be exercised. These works usually focus on the principals' past behavior and do not consider the evolution of the object to be accessed, thus only partially satisfying **C2**.[6] On the contrary, our work also exploits information on how and by whom the object of interest has been generated for access decision making. In this respect, our approach is closer to provenance-based approaches, discussed next. In particular, our framework allows specifying access constraints that restrict user permissions based on the features of the users and extends standard provenance relations with relations involving users (or agents, using provenance terminology) such as owns and contributedTo. Therefore, our model is able to express access constraints taking into account both the object and subject dimensions. In addition, relying on data provenance, it also allows accounting for the process (the action dimension) used to derive a given artifact.

A number of works [2, 10, 14, 17, 20] propose to exploit provenance information in access control. Sun et al. [20] define an access control model that uses provenance information for access decision making and this is modeled by including provenance paths in the rules forming a policy. This work was later extended in [14] where

[6]We consider static history-based access control approaches only provide very *limited* support for **C2** as they do not account for the actual user behavior.

		C1			C2	C3
Static history-based access control	Bartoletti et a. [1]			A	○	✘
	Skalka et a. [19]			A	○	✘
Dynamic history-based access control	Krukow et al. [9]	S		A	●	✘
	Tandon et al. [21]			A	●	✘
Provenance-based access control	Cadenhead et al. [2]	S	O	A	✔	✘
	Sun et al. [20]	S	O	A	✔	✘
	Nguyen et al. [14]	S	O	A	✔	✘
	Locroix et al. [10]	S	O	A	✔	✘
	Pei and Ye [17]	S	O	A	✔	✘
Data fusion	Zannone et al. [23]		O	A	○	✔
	den Hartog et al. [5]		O	A	○	✔
Relationship-based access control	Community-based	S			✘	✘
	Crampton et al. [4]	S	O		✘	✘
Our work		S	O	A	✔	✔

Legend
Perspective: subject (S), object (O), action (A)
Supported: full (✔), partial (●), limited (○), not (✘)

Table 3: Comparison with related work

dynamic SoD constraints are supported by adding contextual attribute information in the provenance graph. A similar model is presented in [10] where provenance is combined with RBAC and applied to a cloud environment. Cadenhead et al. [2] extend an XML-based language for the specification of ABAC policies with regular expression and use SPARQL to query the provenance graph (represented as RDF triples). Similarly to these works, we use provenance information to express access constraints. However, our work differs from these approaches in several ways. First, our formalization of provenance-based access constraints can be encoded in XACML and their evaluation requires minimal changes to existing XACML implementations. Moreover, we also exploit provenance for referencing external policies in order to support data fusion (**C3**), which is a new feature comparing to previous provenance-based access control models. Pei and Ye [17] propose a policy retrieval framework based on provenance. This framework exhaustively generates candidate policies for each artifact role related to the target action type by enumerating all possible combinations of dependency type and policy combining algorithm. Then, each candidate policy is verified against provenance and recorded along with the corresponding access control decision. These training examples are fed into a decision tree classifier to derive the provenance-based access control policies to be enforced. Our framework differs from the one of Pei and Ye in the fact that it uses provenance to identify the policies applicable to the artifacts used in the creation of the object of interest.

To the best of our knowledge, only few works have proposed to account for the policies associated to the artifacts used in the generation of the requested object [5, 23]. In particular, den Hartog et al. [5] propose a policy framework to regulate data fusion processes and the artifacts obtained from these processes. This framework makes use of policy templates to define how the policies of the artifacts used in data fusion processes should be combined. However, it only considers the direct inputs for an artifact, thus only providing limited support for **C2**. Moreover, these policies are static in the sense that they are defined when an object is created while the link with the input artifacts is not maintained. On the other hand, in our work we exploit the provenance graph to maintain the link between an artifact and the artifacts used for its creation

and use policy references to retrieve the policies of those artifacts at evaluation time.

Our framework relies on provenance graphs to determine user permissions. The use of graphs for access decision making can be found also in community-based systems and, in particular, on-line social networks. In these systems, access decisions are based on interpersonal relationships between users, which are modeled in a social graph. Several relationships-based access control (ReBAC) models have been proposed (see [16, Section 3] for a survey). These models define permissions in terms of paths in the social graph (e.g., friends, friends of friends) as well as enable the use of topology policies to specify access restrictions based on topological properties of the social graph (e.g., degree of separation, k-clique). However, these models only account for relationships between users and do not consider the relations between users and objects that is requested. This limitation is addressed by Crampton and Sellwood [4], who propose a generic ReBAC model based on path conditions. A path condition in [4] is modeled as a sequence of relationships, which is used to map the resource requester to a (set of) authorization principal(s). The subject and resource specified in the access request are first used to find the set of applicable principals. These principals are then used to determine whether the action specified in the request should be authorized based on a given authorization policy. This model, however, only represent a static view of the system and does not account for processes and, in general, for the creation and evolution of artifacts.

6 DISCUSSION & CONCLUSIONS

This paper presents an access control policy framework that relies on provenance information to keep track of the evolution of an artifact in the system, as well as to retrieve the policies associated to the artifacts used in its generation. Specifically, our framework extend the ABAC paradigm with the notion of path condition, which is used to define access constraints concerning the past history of an artifact (provenance-based constraints) and to identify policies regulating the access to the artifacts used for its creation (input-dependent constraints). We have evaluated the expressiveness of our framework and compared with related work. As shown in Table 3, it is the only framework able to meet all criteria identified in Section 2, thus offering the expressiveness necessary to deal with the needs of multi-source cooperative systems. We have also illustrated how provenance-based and input-dependent constraints can be implemented in existing XACML-based access control frameworks. In particular, we propose to implement mechanisms for path condition resolution as an external service (represented by the Provenance Module in the architecture of Figure 3), which is invoked through APIs at policy evaluation time. This allows integrating the logic for path condition resolution into existing authorization frameworks with only minimal changes to the existing platform.

In this work, we have considered core provenance information for policy specification. We believe that accounting for contextual information in the provenance graph would enable the definition of more fined-grained policies. Nguyen et al. [14] have addressed this issue by adding a *ContextualInfoSet* to transactions, i.e. the set of all contextual information related to a transaction, which is materialized in the provenance graph as a node linked to the process node through an additional type of casual dependency hasAttributeOf. For future work, we plan to extend our policy language and provenance graphs to accommodate contextual information along the lines of the PROV-DM data model [13] where provenance graphs are augmented with attributes in form of relation annotations. The use of a standardized provenance model will allow us to reuse provenance graphs generated by existing systems.

Acknowledgement. This work is funded by the ECSEL project SECREDAS (783119) and the ITEA3 project APPSTACLE (15017).

REFERENCES

[1] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Proceedings of International Conference on Foundations of Software Science and Computation Structures*, pages 316–332. Springer, 2005.
[2] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham. A language for provenance access control. In *Proceedings of Conference on Data and Application Security and Privacy*, pages 133–144. ACM, 2011.
[3] P. Colombo and E. Ferrari. Access Control in the Era of Big Data: State of the Art and Research Directions. In *Proceedings of Symposium on Access Control Models and Technologies*, pages 185–192. ACM, 2018.
[4] J. Crampton and J. Sellwood. Path Conditions and Principal Matching: A New Approach to Access Control. In *Proceedings of Symposium on Access Control Models and Technologies*, pages 187–198. ACM, 2014.
[5] J. den Hartog and N. Zannone. A policy framework for data fusion and derived data control. In *Proceedings of International Workshop on Attribute Based Access Control*, pages 47–57. ACM, 2016.
[6] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings of Conference on Computer and Communications Security*, pages 38–48. ACM, 1998.
[7] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.
[8] S. P. Kaluvuri, A. I. Egner, J. den Hartog, and N. Zannone. SAFAX - an extensible authorization service for cloud environments. *Front. ICT*, 2, 2015.
[9] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for history-based access control and reputation systems. *J. Comput. Secur.*, 16(1):63–101, 2008.
[10] J. Lacroix and O. Boucelma. Provenance-based access control in the cloud. In *Proceedings of International Conference on Services Computing*, pages 755–756. IEEE, 2013.
[11] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
[12] L. Moreau, B. Cliffor, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, 2011.
[13] L. Moreau and P. Missier. PROV-DM: The PROV Data Model. W3C Recommendation, W3C, 2013.
[14] D. Nguyen, J. Park, and R. S. Sandhu. A provenance-based access control model for dynamic separation of duties. In *Proceedings of Annual International Conference on Privacy, Security and Trust*. IEEE, 2013.
[15] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard, 2013.
[16] F. Paci, A. Squicciarini, and N. Zannone. Survey on access control for community-centered collaborative systems. *ACM Comput. Surv.*, 51(1):6:1–6:38, 2018.
[17] J. Pei and X. Ye. Towards Policy Retrieval for Provenance Based Access Control Model. In *Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications*, pages 769–776. IEEE, 2014.
[18] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
[19] C. Skalka and S. Smith. History effects and verification. In *Programming Languages and Systems*, pages 107–128. Springer, 2004.
[20] L. Sun, J. Park, D. Nguyen, and R. Sandhu. A provenance-aware access control framework with typed provenance. *IEEE Transactions on Dependable and Secure Computing*, 13(4):411–423, 2016.
[21] L. Tandon, P. W. L. Fong, and R. Safavi-Naini. HCAP: A History-Based Capability System for IoT Devices. In *Proceedings of Symposium on Access Control Models and Technologies*, pages 247–258. ACM, 2018.
[22] D. Trivellato, N. Zannone, M. Glaundrup, J. Skowronek, and S. Etalle. A semantic security framework for systems of systems. *Int. J. Cooperative Inf. Syst.*, 22(1), 2013.
[23] N. Zannone, S. Jajodia, and D. Wijesekera. Creating Objects in the Flexible Authorization Framework. In *Data and Applications Security XX*, pages 1–14. Springer, 2006.