Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

Middleware for efficient and confidentiality-aware federation of access control policies

Maarten Decat*, Bert Lagaisse and Wouter Joosen

Abstract

Software-as-a-Service (SaaS) is a type of cloud computing in which a tenant rents access to a shared, typically web-based application hosted by a provider. Access control for SaaS should enable the tenant to control access to data that are located at the provider side, based on tenant-specific access control policies. Moreover, with the growing adoption of SaaS by large enterprises, access control for SaaS has to integrate with on-premise applications, inherently leading to a federated set-up. However, in the state of the art, the provider completely evaluates all policies, including the tenant policies. This (i) forces the tenant to disclose sensitive access control data and (ii) limits policy evaluation performance by having to fetch this policy-specific data. To address these challenges, we propose to decompose the tenant policies and evaluate the resulting parts near the data they require as much as possible while keeping sensitive tenant data local to the tenant environment. We call this concept *policy federation*. In this paper, we motivate the need for policy federation using an in-depth case study analysis in the domain of e-health and present a policy federation algorithm based on a widely-applicable attribute-based policy model. Furthermore, we show the impact of policy federation on policy evaluation time using the policies from the case study and a prototype implementation of supporting middleware. As shown, policy federation effectively succeeds in keeping the sensitive tenant data confidential and at the same time improves policy evaluation time in most cases.

Keywords: Software-as-a-Service; Security; Access control; Policy-based access control; Federation; Performance

1 Introduction

Software-as-a-Service or SaaS is a type of cloud computing in which a *tenant* rents access to a shared application hosted by a *provider* [1]. The tenant is an organization representing multiple end-users, who use the application through a thin client, typically a web browser. The provider protects the data in the application, for example by ensuring tenant isolation or preventing data leakage. However, for the tenant, SaaS is a form of outsourcing: while the SaaS application belongs to the provider, the application data, although hosted by the provider, still belongs to the tenant. Therefore, SaaS applications should also enable the tenants to control access to their data in the application, based on tenant-specific access control policies.

Traditional SaaS applications such as Google Apps (an office suite) and Salesforce (CRM) allow the tenant to control access to the application by offering the tenants a dashboard for configuring access control. These SaaS applications are mainly targeted at small and medium enterprises looking for a fully outsourced IT infrastructure and this approach fits them well.

Recently however, large enterprises have started to adopt SaaS as well, for example Cisco in the domain of CRM [2] or large hospitals in the domain of e-health [3,4]. While these enterprises employ SaaS to outsource specific, non core-business functionality, the organization-wide policies of the tenant still apply. These policies reason about data that remain stored in on-premise applications such as patient management or medical record systems (illustrated in Figure 1). A federated setup between tenant and provider is inherent to such a deployment context.

The federated set-up between tenant and provider poses important challenges. While techniques for federated authentication [5,6] allow user data to be securely shared between tenant and provider, the provider still completely evaluates the tenant policies. This approach causes two

*Correspondence: maarten.decat@cs.kuleuven.be
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium

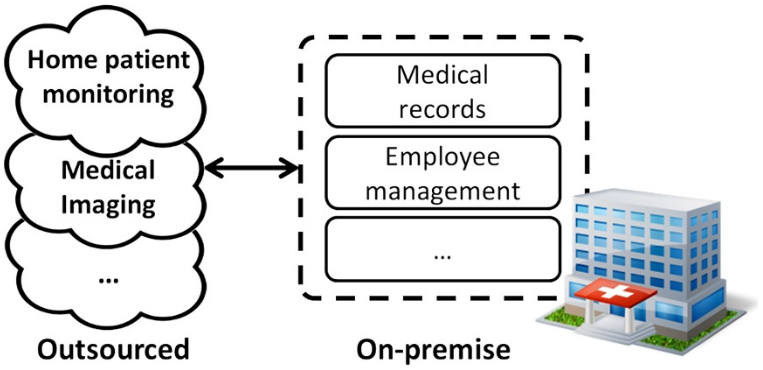

Figure 1 Large organizations such as hospitals employ both SaaS applications and on-premise applications, leading to a federated setup.

main problems: (i) it forces the tenant to disclose sensitive access control data, such as lists of patients being treated by a physician. Although the tenant may trust the provider with the data in the SaaS application, it does not necessarily trust the provider with this sensitive on-premise application data and wants to keep it confidential. Moreover, stringent regulatory requirements such as HIPAA [7] or the European DPD [8] even forbid the hospital to share this data. (ii) This approach limits policy evaluation performance by having to fetch the required data. Many of the access control policies require large amounts of access control data and fetching this data from the tenant takes a considerable amount of time.

To address these challenges, we introduce *policy federation*[a]. In this process, the tenant policies are decomposed and the resulting parts are evaluated near the data they require as much as possible while keeping sensitive tenant data local to the tenant premises. As shown, policy federation effectively succeeds in keeping the sensitive tenant data confidential and at the same time improves policy evaluation time in most cases.

This paper first presents an in-depth case study analysis in the domain of e-health motivating the need for policy federation. The paper then describes a confidentiality-aware policy federation algorithm for optimal policy evaluation time using a widely-applicable attribute-based policy model. Finally, the paper shows the impact of policy federation on policy evaluation time, using the policies from the case study and a prototype of supporting middleware.

In summary, the contributions of this paper are:

1. An in-depth case study analysis in the domain of e-health, showing the need for policy federation.
2. A full description of policy federation consisting of (i) an attribute-based policy model, (ii) a policy federation algorithm and (iii) a description of supporting middleware.
3. A practical evaluation of the impact of policy federation on policy evaluation time, using the policies from the case study and a prototype of the supporting middleware for policy federation.

The rest of this paper is structured as follows. Section 2 discusses the context of this work: access control for SaaS applications. Section 3 describes the e-health case study that motivates this work. Section 4 defines the attribute-based policy model and Section 5 the policy federation algorithm. Section 6 evaluates policy federation in terms of performance and thereby elaborates on the design of supporting middleware. Section 7 provides a discussion of policy federation. Section 8 covers related work and Section 9 concludes this paper.

2 Context: access control and SaaS applications

This section first discusses access control in the domain of SaaS applications as background to this paper.

Access control is an important part of application-level security that limits the *actions* (e.g., read, write) which a *subject* (e.g., a physician) can take on an *object* in the system (e.g., a patient file). Access control rules are often externalized from the application they constrain and expressed in modular, declarative *access control policies* for reasons of separation of concerns and modifiability. Policy-based access control fits SaaS applications well, because it allows tenant-specific security logic to be externalized from the shared application and be bound at run-time.

Multiple models have been proposed for expressing access control policies, such as Mandatory Access Control (MAC, [9]), Discretionary Access Control (DAC, [9]) and Role-Based Access Control (RBAC, [10]). The more recent Attribute-Based Access Control (ABAC, [11]) generalizes previous models and expresses access control policies in terms of key-value properties called *attributes* of the subject (e.g., the subject id, username or roles), the object (e.g., the object id, location or content) and the environment (e.g., the time, physical location or usage context). Attributes provide increased expressivity with regard to previous models and offer a unit of data transport between the different components or parties involved in access control. For both reasons, this work builds upon ABAC.

The reference architecture for policy-based access control infrastructures was defined by IETF and DMTF and refined by the XACML standard [12]. In the reference architecture (see Figure 2), the policy decision point (PDP) makes the actual access control decision. The policy enforcement point (PEP, e.g., an API or a reference monitor) requests an access control decision from the PDP through the context handler. An access control request generally consists of information about the subject, the object, the action and the environment. The context handler gathers initially known attributes from one or more policy information points (PIPs, e.g., a database), which the PDP uses to evaluate the applicable policies loaded from the policy administration point (PAP). Since the required attributes for evaluating a policy depend on the values of former attributes, it is generally impossible to determine the set of required attributes up-front and the PDP can request additional attributes from the context handler if needed. Eventually, the PDP returns its decision (permit or deny), which the PEP enforces.

3 Case study analysis: home patient monitoring

To show the need for policy federation, this section describes the SaaS application that inspired this work: a home monitoring system for patients of cardiovascular diseases, provided to hospitals as a service. As stated in the introduction, large enterprises and non-profit organizations have started to adopt SaaS, amongst others in the domain of e-health. Health care organizations employ on-premise applications for core-business functionality such as patient data management, but outsource functionality which is not core-business to SaaS applications, such as the patient monitoring system. This section firsts gives an overview of the system, then illustrates the hospital's access control policies for the SaaS application and finally describes the problem statement of this paper in detail.

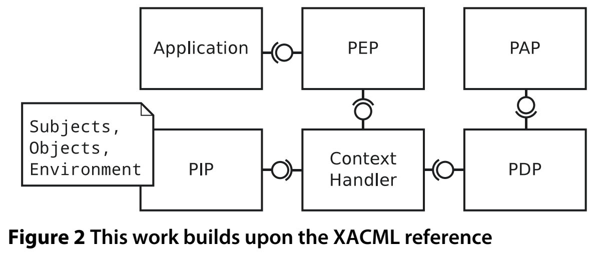

Figure 2 This work builds upon the XACML reference architecture for policy-based access control infrastructures [12].

3.1 Overview of the system

The home patient monitoring system (HPMS, see Figure 3) allows patients of cardiovascular diseases to be monitored continuously after leaving the hospital by wearing sensors such as a chest band or a wrist band. These sensors collect measurements such as the electric activity of the heart, the blood pressure or the temperature. The measurements are sent from the patients to the application back-end using a smart-phone as an intermediary device and are then stored and processed by the provider. In the first place, the provider employs telemedicine operators which continuously check upon their patients. For this, the system offers an overview of the patient's status, showing recent measurements, health charts and an estimated risk level. If medical assistance is required, the patient's physician at the hospital is notified. These physicians can also check upon the status of the patient proactively using a status overview similar to that of the telemedicine operators. A patient's status can also be viewed by the patients themselves or by other physicians and nurses at the hospital, for example when the patient is admitted there. Finally, the system provides functionality such as patient questionnaires and shared notes on a patient overview.

The HPMS is a good example of a state-of-the-art SaaS application. In this system, the hospital is the tenant of the application and in itself manages multiple end-users, i.e., the patients, physicians and nurses. Next to the HPMS, the hospital also employs other SaaS applications, e.g., for medical imaging, and on-premise applications, e.g., for patient records or employee management.

As for all e-health applications, security is paramount for the HPMS. For example, it handles personal data and is subject to stringent regulatory requirements (e.g., HIPAA [7] or the European DPD [8]). Of these security requirements, this paper focuses on the sub-domain of access control.

3.2 Access control policies from the case study

The hospital's access control policies that apply to the HPMS provide a good example of policies that apply to current SaaS applications. This section first discusses the general structure of the hospital policies and then provides a part of these policies in detail.

3.2.1 Structure of the hospital's policies

As mentioned in Section 2, this work builds upon attribute-based access control, which structures policies

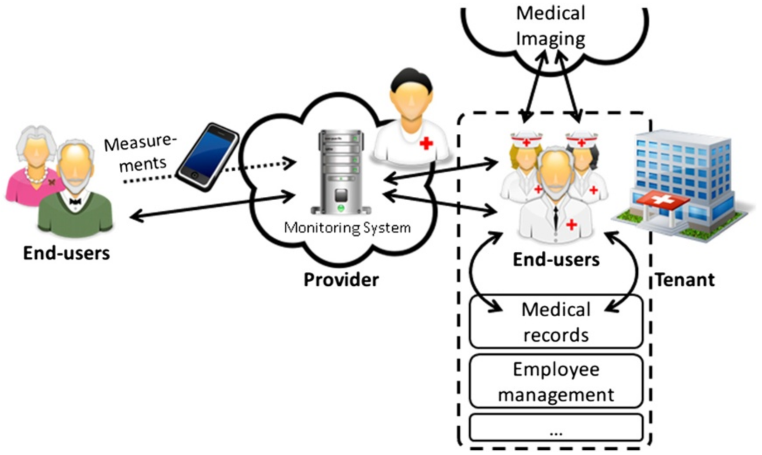

Figure 3 The case study that inspired this work: a home patient monitoring system (HPMS) offered to hospitals as a SaaS application. Next to the HPMS, the hospital also employs several on-premise applications (e.g., for employee management) and several other off-premise SaaS applications (e.g., for medical imaging).

by making the distinction between the subject, the object, the action and the environment. We apply the same structure in this discussion.

Objects and actions. The objects of the hospital policies and the possible actions on them are determined by the structure of the data in the HPMS. The previous section mentioned five types of application data: (1) the raw measurements, (2) the overview of the patient's status, (3) the notifications sent to physicians, (4) the notes added to a patient's status overview and (5) the patient questionnaires. The actions on these objects are as follows: The raw measurements, the patient's status overview and the notifications are all created by the system and cannot be altered; end-users can only view them. Notes on the other hand can be created, viewed, updated and deleted. Patient questionnaires can be created and assigned to patients by physicians. Patients can view and fill in open patient questionnaires and both patients and physicians can view completed patient questionnaires. Next to the five types of application data, the hospital can also constrain access to the HPMS as a whole.

Subjects. The subjects of the hospital policies are determined by the structure of the hospital. The hospital consists of multiple medical departments, such as cardiology, oncology, elder care, general medicine and the emergency department. Each department employs nurses and specialist physicians, such as cardiologists, oncologists, surgeons and anesthetists. The general medicine department also employs a number of general practitioners. Inside a department, the personnel is structured in teams, for example, consisting of multiple cardiologists, a head cardiologist and assisting nurses. Finally, the hospital also provides a number of supporting services, such as general administration and finances.

Environment. The environment of the hospital policies provides the current time and date.

3.2.2 Detailed policies

Following the general policy structure, this section illustrates a hospital policy from the case study in detail by zooming in to the policies for viewing the status overview of a patient. Of all the actions, this action can be executed by the most types of subjects, leading to the most extensive policies in the case study. Other actions are constrained by similar rules. We start from broad organization-wide policies and end with specific policies for specific kinds of subjects. Notice that while we try to be as specific as possible, the textual format is still informal and a translation step towards a more formal policy language is necessary to remove all ambiguities. We provide the XACML encoding and an extensive overview of the required attributes on-line [13].

The following organization-wide policies of the hospital also apply to the HPMS:

P_1. A member of the medical personnel can not access any data about a patient who has explicitly withdrawn consent for him or her, except in case of emergency.

The following hospital policies apply to the HPMS as a whole:

P_2. Only physicians, nurses and patients can access the HPMS.

P_3. Of the physicians, only general practitioners, physicians of the cardiology department, physicians of the elder care department and physicians of the emergency department can access the HPMS.

P_4. Of the nurses, only nurses of the cardiology and the elder care department can access the HPMS.

P_5. Nurses can only access the HPMS during their shifts.

P_6. Nurses can only access the HPMS from the hospital.

P_7. Of the nurses of the cardiology department, all nurses can access the HPMS.

P_8. Of the nurses of the elder care department, only nurses who have been allowed to use the HPMS can access the HPMS.

The following hospital policies apply to viewing the status of a patient:

P_9. Physicians of the cardiology department, physicians of the elder care department and physicians of the emergency department can always view a patient's status in case of emergency (triggered by the physician, triggered by a telemedicine operator or as indicated by the monitoring data).

P_{10}. General practitioners can only view the status of a patient who is currently on consultation or whom they treated in the last two months or for whom they are assigned the primary general practitioner at the hospital or for whom they are assigned responsible in the HPMS.

P_{11}. Head physicians of the cardiology department can view the patient status of any patient in the HPMS.

P_{12}. Physicians of the cardiology department can view the patient status of any patient treated by themselves or by a physician in their team.

P_{13}. Physicians of the elder care department can only view the patient status of a patient who is currently admitted to their care unit or whom they have treated in the last six months.

P_{14}. Physicians of the emergency department can only view the status of a patient in case the status of that patient is bad.

P_{15}. Nurses can only view a patient's status of the last 5 days.

P_{16}. Nurses of the cardiology department can only view the patient status of a patient admitted to their nurse unit for whom they are assigned responsible, up to three days after they were discharged.

P_{17}. Nurses of the elder care department can only view the patient status of a patient currently admitted to their nurse unit for whom they are assigned responsible.

P_{18}. A patient can only access the HPMS if (still) allowed by the hospital.

P_{19}. A patient can only view his own status.

3.2.3 Analysis

In terms of attribute-based access control, the 19 policies given above require 30 different attributes in total, such as the subject id, the department of the subject, the list of patients treated by a physician, the owner of an object, the current date etc (see [13]). Of these attributes, 19 are hosted by the hospital (e.g., the list of patients treated by a physician), 7 are hosted by the provider (e.g., the owner of an object) and 4 are shared in the policy evaluation process (e.g., the id of the subject making the request). Of the 19 tenant attributes, 8 are sensitive, such as the lists of patients. The number of policies required to reach a decision for a single request ranges from 3 to 7 (with a mean of 4.79) and the number of attributes ranges from 4 to 13 (with a mean of 7.65). The case study illustrates that the policies of a tenant of a SaaS application require attributes from both the tenant and the provider. This leads to a federated set-up, which is the focus of this work.

3.3 Problem statement and solution

As discussed in the introduction, the hospital's access control policies would be evaluated by the provider in traditional SaaS applications. This causes two main problems:

1. The hospital would be forced to share all required attributes with the provider, including sensitive attributes which the hospital does not want to share for reasons of limited trust or even cannot share by law. More precisely, we assume the provider to be honest, but curious: the provider correctly communicates with the tenant, but can analyze the communication for the tenant's sensitive data and has an interest in this from a business point of view, because of a malicious employee or because of an external attacker. We do not directly take into account third party attacks such as eavesdropping on the channel between tenant and provider since other solutions exist for those.
2. All required attributes would have to be fetched by the provider during policy evaluation. While the presented policies are only a subset of all hospital policies and will also be much more detailed in practice, the policies already require 30 different attributes of which 19 are hosted by the hospital. Given that a single attribute request can have a large latency because of the complex data flows in federated applications and the geographical distance between tenant and provider, this approach would limit the performance of policy evaluation.

Both issues can be addressed if the hospital evaluates parts of its policies itself. For example, if the hospital evaluates whether a user has treated the owner of the status overview in the last two months (P_{10}), this data remains confidential. Similarly, if the hospital evaluates whether a user is a general practitioner (P_3), this data does not have to be fetched by the provider. In this approach, tenant and provider will cooperate to achieve an access control decision, a concept we call *federated authorization* [14]. In this paper, we describe how to decompose and distribute the hospital policies over the provider and the hospital based on the location and sensitivity of the attributes, a process we call *policy federation.*

The complete solution presented in this paper consists of three parts: (i) an attribute-based policy model which allows us to reason about policy federation, (ii) the actual policy federation algorithm and (iii) a description, prototype and evaluation of supporting middleware. In the next sections, we discuss each of these.

4 Policy model

In order to reason about policy federation, this section first defines an attribute-based policy model based on the core features of current policy languages such as XACML [12]. This minimal subset supports all the policies of the case study, but remains generic in order to guarantee its wide applicability. Several other authors have taken similar approaches, e.g., Crampton and Huth [15]. With respect to these, our model focuses on the aspects related to policy federation, i.e., the general structure of a policy and how a policy is evaluated.

4.1 Structure of a policy

The policy model used in this work represents policies using the concept of a policy tree, similar to [15,16]. Each policy in the tree states for which requests it is applicable by means of a target. The leafs of the policy tree are called *atomic policies*, the others are called *composed policies*.

4.1.1 Atomic policies

Atomic policies state in which conditions a certain request is permitted and in which it is not. They therefore consist of a target, an effect and a condition. The target determines whether the policy applies to the request or not. The effect of a policy is either Permit or Deny, respectively permitting or denying the request. The condition determines whether the effect holds or not. Thus, the result of evaluating a policy is either Permit, Deny or NotApplicable.

As mentioned before, this work builds upon ABAC and as a consequence, targets and conditions are expressions on the attributes of the subject (s), the object (o), the action (a) and the environment (e). Such expressions can contain three kinds of elements: (i) functions, e.g., "and", "in" or "==", (ii) attribute references, e.g., "s.roles" referring to the roles of the subject and (iii) literal values, e.g., "physician". Possible attribute types are primitive types such as integers, strings, booleans and dates, or lists of these.

Using the notation P_{Atom}= <*Target, Effect, Condition*>, policy P_1 as defined in Section 3.2 can be represented as follows:

P_1 = <*a.id == "access" & "medical_personnel" in s.roles, Deny, s.id in o.owner_withdrawn_consents & ... >*

4.1.2 Composed policies

Composed policies combine the results of several other policies, either atomic policies or other composed policies. They therefore consist of a target, a policy combination algorithm and an ordered list of sub-policies. The target is defined the same as for atomic policies. The policy combination algorithm combines the effects of the sub-policies into the effect of the composed policy. In order to remain compatible to XACML, we limit ourselves to three policy combination algorithms, which suffice to express the policies from the case study: PermitOverrides, DenyOverrides and FirstApplicable [12]. Notice that policy evaluation requires a single result, i.e., the access control decision. Since every set of policies can be combined to a single combined policy using the policy combination algorithms, we assume the policy tree to have a single root, which applies to all requests.

Using the notation P_{Comp} = <*Target, PolicyCombinationAlgorithm, Sub-policies*>, the example policies of Section 3.2 can be combined into a single composed policy as follows (illustrated in Figure 4):

P_0 = <*true, FirstApplicable, [*P_1, P_2, <*"physician" in s.roles, DenyOverrides, [*P_3, P_9, ..., P_{14}]>, <*"nurse" in s.roles, DenyOverrides, [*P_4, ...]>, ...>

4.1.3 Sensitive elements

In the model, two elements of a policy can be declared sensitive: (i) the attributes used in a policy and (ii) the policies themselves. For composed policies, confidentiality applies to the whole policy tree below it. In practice, these confidentiality constraints can be expressed by providing a separate meta-policy or by annotating the access control policies themselves. Since attributes can be referenced multiple times throughout a policy, using a separate meta-policy provides the advantage of central management. Policy elements on the other hand are best annotated in the access control policies themselves. The result for the policies of the case study is available on-line [13].

4.2 Policy evaluation

The evaluation of a policy structured as described above also impacts policy federation. We here define two

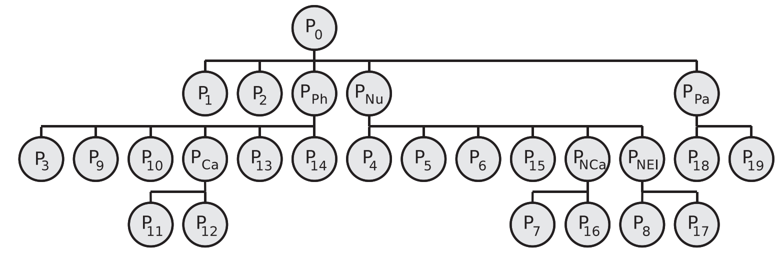

Figure 4 Representation of the example policies of Section 3.2 as a policy tree using our policy model.

aspects: (i) the order in which the elements of the policy tree are evaluated and (ii) how attributes are fetched.

4.2.1 Evaluation order

A policy is evaluated by first evaluating its target. If the policy does not apply to the request, NotApplicable is returned. If the policy does apply, its condition is evaluated (in case of an atomic policy) or its sub-policies are evaluated (in case of a composed policy) and the result is returned. For composed policies, the sub-policies are evaluated in the given order; as a consequence, the policy tree is evaluated depth-first. For now, we assume all sub-policies and expressions to be evaluated sequentially.

4.2.2 Fetching attributes

As mentioned in Section 2, the required attributes are fetched from their respective policy information points during policy evaluation. Because the required attributes for evaluating a policy depend on the values of former attributes, it is generally impossible to determine the set of required attributes up-front and we generally assume that an attribute is only fetched when it is required. To enable this, the identifiers of the subject, the object and the action are given by the policy enforcement point for initiating the policy evaluation. We also make the realistic assumption that attribute values are cached during the evaluation of a policy for a single request in order to avoid unnecessary attribute fetches and to guarantee correct evaluation of policies that require the same attribute multiple times in the presence of out-of-band attribute updates. We do not take into account attribute caching across multiple requests in order to avoid freshness issues.

5 Policy federation algorithm

Based on the policy model described in the previous section, this section defines the policy federation algorithm, i.e., the algorithm that will decompose and deploy the tenant policies across tenant and provider. We first give an overview of the algorithm and then go into each of the major steps. Finally, we discuss the correctness of the algorithm in terms of policy equivalence.

5.1 Overview

The goal of the policy federation algorithm is to decompose and distribute the tenant policies so that sensitive attributes and policies remain confidential and the evaluation performance is optimized, i.e., the evaluation duration is minimized. For attribute-based policies, this evaluation duration is mainly determined by the latency of fetching the required attributes [17]. The latency of a remote attribute fetch between tenant and provider will be an order of magnitude larger than a local database call, taking into account the complex data flows in federated applications and the geographical distance between tenant and provider. Therefore, the goal of the algorithm is to minimize the number of requests between tenant and provider.

An important design decision is the granularity of the policy distribution. In theory, even internal parts of an atomic policy could be distributed. However, we deliberatly limit the granularity to sub-policies in the policy tree. As such, the decomposed policy remains compatible with existing policy infrastructures and the existing policy combination algorithms can be used for handling the results. However, this approach also limits the granularity of policy decomposition. Therefore, the first step in the algorithm is to normalize larger policies into an equivalent set of smaller policies, which can then be separately deployed. Afterward, the algorithm tries to combine multiple remote policy references into a single reference again, in order to minimize the number of remote policy evaluation requests.

An overview of the resulting policy federation algorithm is given in Algorithm 1. The algorithm requires two inputs: (i) the policy P to be federated, annotated with sensitivity labels in the policy tree and (ii) the list of attributes, each having a location and sensitivity label. The location of an attribute is either tenant-side or provider-side, the sensitivity label of an attribute or policy is a boolean that determines whether the attribute or policy can be shared with the provider or not. The algorithm provides three outputs: (i) *root*: the policy at the root of the new policy tree which can reference remote policies, (ii) S_P: the set of referenced policies to be deployed provider-side and (iii) S_T: the set of referenced policies to be deployed tenant-side. Throughout the algorithm, several policy transformations are applied to the policy tree (see Equations (T1–T9)). Of these transformations, T1, T2, T3 and T4 allow policies to be split in an equivalent set of smaller policies; T5, T6 and T7 allow sub-policies of combined policies to be combined; T8 and T9 show the commutativity of PermitOverrides and DenyOverrides. The correctness of these rules can be proven using their respective decision tables. The algorithm itself consists of three major steps: normalization, decomposition and combination. In the next sections, we go into detail about each of these steps.

Algorithm 1 Overview of the policy federation algorithm. The methods normalize(), decompose() and combine() are defined in Algorithms 2, 3 and 4.

Inputs: *P*: a policy, annotated with sensitivity labels (true or false), *A*: a list of attributes, each having a location (tenant-side or provider-side) and sensitivity label (true or false).

Outputs: *root*: the policy at the root of the new policy tree which can reference remote policies, S_P: the set of referenced policies to be deployed provider-side, S_T: the set of referenced policies to be deployed tenant-side.

```
S_P, S_T = [ ]
// Step 1: Normalization
P = normalize(P)
// Step 2: Decomposition
root = decompose(P, “providerSide”)
// Step 3: Combination
root = combine(root)
for Policy p in S_T: S_T.replace(p, combine(p))
for Policy p in S_P: S_P.replace(p, combine(p))
```

5.2 Step 1: normalization

As said, the goal of the normalization step is to convert larger policies into an equivalent set of smaller policies, which can then be separately deployed. Therefore, the first step of the federation algorithm iteratively applies transformations T1, T2, T3 and T4 as defined in Equations (T1–T4) to the given policy *P* until no more sub-policies can be transformed, as shown in Algorithm 2.

Algorithm 2 Definition of the `normalize()` method.

```
def normalize(Policy p):
  Policy p’ = p.applyTransformations( [T1, T2, T3, T4] )
  if p’ != p:
    // a transformation was applied
    return normalize(p’)
  else:
    if p is AtomicPolicy: return p
    else: // composed policy
      for Policy sub in p.subpolicies:
        p.subpolicies.replace(sub, normalize(sub))
      return p
```

Notice that transformations T1 to T4 only utilize `or` statements. The reason for this is that we want to remain compatible to XACML and only employ FirstApplicable, PermitOverrides and DenyOverrides, but converting an `and` statement would require other policy combination algorithms. For example, the equivalents of T1 and T2 would require the policy combination algorithm BothApplicable.

5.2.1 *Results from the case study*

When applying the federation algorithm to the policies from the case study, P_9 will be split into three times three parts because both its target and condition consist of a ternary term that can be split using T1 or T3. Similarly,

$$< T_1|T_2, E, C > \;\Leftrightarrow\; < true, FirstApplicable, [\, < T_1, E, C >, < T_2, E, C >] > \tag{T1}$$

$$< T_1|T_2, PCA, [P_1...P_n]> \;\Leftrightarrow\; < true, FirstApplicable, [< T_1, PCA, [\,P_1...P_n]>, < T_2, PCA, [P_1...P_n]>]> \tag{T2}$$

$$< T, Permit, C_1|C_2 > \;\Leftrightarrow\; < T, PermitOverrides, [\, < true, Permit, C_1 >, < true, Permit, C_2 >] > \tag{T3}$$

$$< T, Deny, C_1|C_2 > \;\Leftrightarrow\; < T, DenyOverrides, [\, < true, Deny, C_1 >, < true, Deny, C_2 >] > \tag{T4}$$

$$< T, PermitOverrides, [\,P_1, P_2, P_3] > \;\Leftrightarrow\; < T, PermitOverrides, [\, < true, PermitOverrides, [\,P_1, P_2] >, P_3] > \tag{T5}$$

$$< T, DenyOverrides, [\,P_1, P_2, P_3] > \;\Leftrightarrow\; < T, DenyOverrides, [\, < true, DenyOverrides, [\,P_1, P_2] >, P_3] > \tag{T6}$$

$$< T, FirstApplicable, [\,P_1, P_2, P_3] > \;\Leftrightarrow\; < T, FirstApplicable, [\, < true, FirstApplicable, [\,P_1, P_2] >, P_3] > \tag{T7}$$

$$< T, PermitOverrides, [\,P_1, P_2] > \;\Leftrightarrow\; < T, PermitOverrides, [\,P_2, P_1] > \tag{T8}$$

$$< T, DenyOverrides, [\,P_1, P_2] > \;\Leftrightarrow\; < T, DenyOverrides, [\,P_2, P_1] > \tag{T9}$$

P_{10} will be split in four parts using T3, P_{12} in two parts using T3 and P_{13} in two parts using T3.

5.3 Step 2: decomposition

After the policy tree has been normalized, step 2 of the algorithm decomposes it so that every sub-tree is deployed on its optimal location (see Algorithm 3). The algorithm estimates the cost of evaluating a certain sub-tree either provider-side or tenant-side in terms of evaluation time and minimizes the total evaluation cost as follows: If the cost of evaluating a sub-policy of a composed policy on the same side as the composed policy is larger than the cost of evaluating it on the other side plus the cost of making a policy evaluation request, the sub-policy is deployed on the other side and it is replaced by a remote policy reference to it. The algorithm applies this reasoning recursively starting from the top policy, which should always be deployed provider-side. For a policy that handles sensitive attributes or is labeled sensitive itself, the cost of evaluating it provider-side is infinite (i.e., it has to be evaluated tenant-side). For the other cases, we here define several cost functions, which focus on the number of required attributes.

Algorithm 3 Definition of the `decompose()` method. $C_{i,P}$, $C_{i,T}$ and C_{PR} are as defined in Section 5, S_T and S_P are as defined in Algorithm 1.

```
def decompose(Policy p, Side parentSide):
    if p is ComposedPolicy:
        for Policy sub in p.subpolicies:
            p.subpolicies.replace(sub, decompose(sub))
    ($C_{i,P}$, $C_{i,T}$) = evaluationCost(p)
    if parentSide == “tenantSide”:
        if $C_{i,P} + C_{PR} < C_{i,T}$:
            $S_P$.add(p)
            return new RemotePolicyReference(p)
        else: return p
    else:
        if $C_{i,T} + C_{PR} < C_{i,P}$:
            $S_T$.add(p)
            return new RemotePolicyReference(p)
        else: return p
```

5.3.1 Cost functions for atomic policies

For atomic policies, the cost functions are as follows:

$$C_{Atom,P} = N_{A,P} * C_L + N_{A,T} * C_R \qquad \text{(CF1)}$$

$$C_{Atom,T} = N_{A,T} * C_L + N_{A,P} * C_R \qquad \text{(CF2)}$$

The cost functions determine the cost of the provider ($C_{Atom,P}$) and the tenant ($C_{Atom,T}$) evaluating a certain atomic policy based on the total number of required provider attributes ($N_{A,P}$) and tenant attributes ($N_{A,T}$) and the cost for fetching an attribute locally (C_L) or remotely (C_R). The location of every attribute determines the cost of fetching the attribute: C_L will be much smaller than C_R.

An important detail is the handling of cached attributes (see Section 4.2). The cost of fetching an attribute from the cache is assumed to be zero and the cost functions should only take into account newly required attributes. However, it is impossible to fully statically determine the set of cached attributes, for example because previous policies in the policy tree can be fully evaluated, but still return NotApplicable. In order to come to a static estimation, we assume the worst case and calculate the minimal set of cached attributes by only taking into account the attributes required by the targets of previously evaluated policies, i.e., super-policies, previous policies on the same level and previous policies on the same level as super-policies. In case an atomic policy has a target that matches all requests, the attributes in the condition are taken into account as well. In case a composed policy has a target that matches all requests, the required attributes of the first policy are taken into account. For simplicity, we assume that non-sensitive cached attributes are shared between tenant and provider by adding them to the policy evaluation requests. Notice that the cost functions above also assume the worst case by taking into account all attributes of the policy, while some attributes may not be needed every time, e.g., the attributes required by the condition if the policy is not applicable (see Section 3.2.3).

5.3.2 Cost functions for composite policies

For composite policies, the cost functions are as follows:

$$C_{Comp,P} = N_{A,P} * C_L + N_{A,T} * C_R + \sum K_{i,P} \qquad \text{(CF3)}$$

$$C_{Comp,T} = N_{A,T} * C_L + N_{A,P} * C_R + \sum K_{i,T} \qquad \text{(CF4)}$$

$N_{A,P}$, $N_{A,T}$, C_L and C_R are defined similarly as for atomic policies. Notice that composite policies only directly require attributes because of their targets and that again, cached attributes are not taken into account. $K_{i,P}$ and $K_{i,T}$ represent the cost of evaluating the i'th sub-policy P_i of composite policy P_{Comp} in case P_{Comp} is evaluated provider-side or tenant-side respectively. In case P_i is evaluated on the other side than P_{Comp}, a policy evaluation request is needed, which has a cost $C_{PR} \simeq C_R$. To take this into account, we define $K_{i,P}$ as the minimum of the cost of evaluating P_i when evaluating P_{Comp} provider-side, thereby actually deciding on the optimal evaluation location of P_i:

$$K_{i,P} = min(C_{i,P}, C_{i,T} + C_{PR}) \qquad \text{(CF5)}$$

$K_{i,T}$ is defined similarly:

$$K_{i,T} = min(C_{i,P} + C_{PR}, C_{i,T}) \tag{CF6}$$

For atomic policies, $C_{i,P}$ and $C_{i,T}$ are defined as CF1 and CF2; for composed policies, $C_{i,P}$ and $C_{i,T}$ are defined recursively as CF3 or CF4.

5.3.3 Results from the case study

The policies from the case study all require more tenant attributes than provider attributes, except for P_9. As a result, most of the policy tree will be deployed tenant-side, starting from the root and only P_9 (or more precisely, the policy tree resulting from normalizing P_9) is still deployed provider-side. Because the root policy P_0 is deployed tenant-side, a provider-side policy reference is inserted as new root.

5.4 Step 3: combination

Finally, the third step of the algorithm tries to combine remote policy references in order to minimize the number of policy evaluation requests between tenant and provider (see Algorithm 4). More precisely, the algorithm combines multiple policies referenced in a single composed policy into a larger equivalent composed policy and combines their remote policy references into a reference to the new combined policy. For this, the algorithm employs transformations T5, T6 and T7 as defined in Equations (T5–T7). In case of FirstApplicable, only consecutive remote policy references in the sub-policies can be combined; in case of PermitOverrides or DenyOverrides, all remote policy references can be combined since these algorithms are commutative as shown by transformations T8 and T9 of Equations (T8–T9).

Algorithm 4 Definition of the `combine()` method. S_T and S_P are as defined in Algorithm 1.

```
def combine(Policy p):
  if p is AtomicPolicy: return p
  else:
    Policy[ ][ ] groups = p.getCombinableSubpolicies()
    for Policy[ ] group in groups:
      ComposedPolicy cp =
          new ComposedPolicy(p.target, p.pca, group)
      $S_P$.replace(group, cp) // no effect if group not in $S_P$
      $S_T$.replace(group, cp) // no effect if group not in $S_T$
      p.subpolicies.replace(group,
          new RemotePolicyReference(cp)
    return p
```

5.4.1 Results from the case study

The policy tree resulting from normalizing and decomposing the policies from the case study does not allow to combine multiple remote policy references. The final policy tree is shown in Figure 5.

5.5 Discussion: policy equivalence

An important property of the policy federation algorithm is that the federated policy gives the same results as the original policy. To make this more concrete, we here introduce the notion of policy equivalence.

Definition: Policy equivalence Two policies P_1 and P_2 are equivalent iff for every request R and context Ctx, evaluating P_1 leads to the same decision as evaluating P_2. The context Ctx is a collection of attribute values of the subject, the object, the action and the environment: $Ctx = (A_S, A_O, A_A, A_E)$. The request R is a subset of the context: $R \subset Ctx$.

Our policy federation algorithm maintains policy equivalence because (1) only step 1 and step 3 transform the policy tree and every applied transformation (see Equations (T1–T9)) maintains policy equivalence and (2) both the original policy and the federated policy share the same context since the policies deployed provider-side will only require provider attributes and non-sensitive tenant attributes and all non-sensitive attributes are available to both tenant and provider. An equivalent decomposition also leads to an equivalent distribution, except for the fact that distributed policy evaluation can introduce network exceptions.

6 Performance evaluation

In this section, we evaluate policy federation in terms of performance. For the performance evaluation, we can evaluate the impact of policy federation on policy evaluation time and the performance of the algorithm itself. The policy federation algorithm is meant to be run at policy deployment time, i.e., independently of the policy evaluation flow, and therefore does not introduce run-time overhead. For the policies presented in the case study, the algorithm takes about 11 ms; for policies of one order of magnitude larger[b], the algorithm still takes less than 2 seconds. Because these durations fit the asynchronous execution of the federation algorithm, we do not provide details about the algorithm and focus on the impact of policy federation on policy evaluation time.

6.1 Prototype

To measure the performance impact of policy federation, we implemented a prototype of both the federation algorithm (2KLOC) and a middleware system supporting policy federation (6KLOC). Both build upon the SunXACML policy evaluation engine. The source code is publicly available at [13].

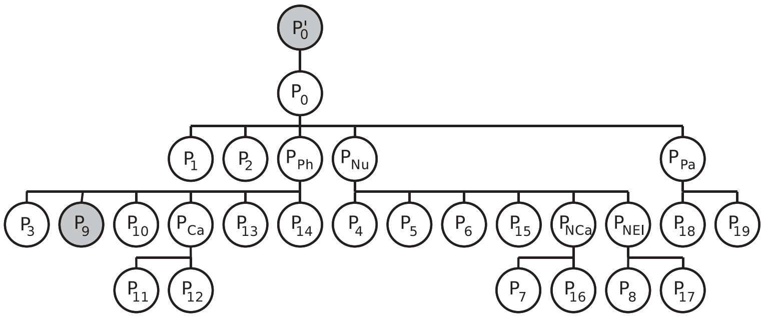

Figure 5 Final result of the applying the policy federation algorithm to P_0. Grey policies are deployed provider-side, white policies are deployed tenant-side. For readability reasons, the normalizations of P_9, P_{10}, P_{12} and P_{13} are not shown.

Figure 6 shows the architecture of the supporting middleware in terms of the XACML reference architecture for policy-based access control infrastructures (see Section 2). As shown, both the provider and the tenant have a PAP, a PDP, a context handler and one or more PIPs since both will evaluate policies. The provider hosts the SaaS application and therefore also the PEP. The provider hosts the attributes concerning the objects in the application (A_O) and the provider part of the environment ($A_{E,P}$) and the tenant hosts the attributes concerning the subjects of the application (A_S) and the tenant part of the environment ($A_{E,T}$). Non-sensitive attributes are made available to the other party by means of an attribute service, the PDPs by means of a Remote Policy Decision Point (RPDP). The RPDPs and attribute services are published as SOAP web-services implemented on top of Apache Tomcat 7 using the Apache CXF services framework. The Policy Federation Layer shown in Figure 6 is the focus of this work. This layer cooperates with the tenant and provider PAP in order to deploy the tenant policies after the initial decomposition step. For more information about the supporting middleware, we refer to [14].

6.2 Test set-up

The performance impact of policy federation can be expected to depend on the characteristics of the policy, e.g., its size, the number of required attributes, the location of these attributes etc. Thus, in order to give a realistic view of the performance impact of policy federation, we employ the policies from the case study and measure (i) the number of remote requests (i.e., attribute requests or policy evaluation requests) between tenant and provider needed for evaluating the policies and (ii) the total policy evaluation time. In the first place, we compare two cases: (i) provider-side evaluation: in this case the policies are completely evaluated provider-side and

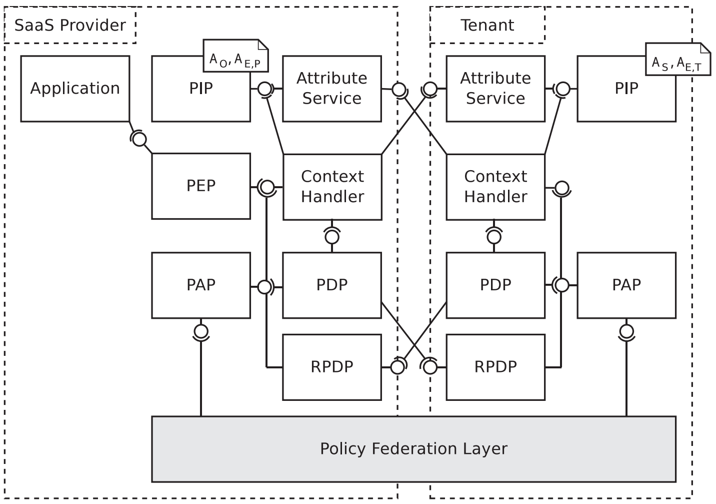

Figure 6 Architecture of the supporting middleware for policy federation in terms of the XACML reference architecture (see Section 2). The Policy Federation Layer is the focus of this work.

(ii) federated evaluation: in this case, the policies are deployed across tenant and provider as resulting from the federation algorithm. For completeness, we also compare the results to (iii) tenant-side evaluation: in this case the policies are completely evaluated tenant-side. We employ 26 different access requests that together cover every branch of the original policy tree. Notice that in the provider-side case, sensitive attributes are fetched from the tenant.

Each of the main components of the prototype runs on a separate machine with 1GiB RAM and a single core of 2.40GHz running Ubuntu 12.04. Attributes are stored locally on the machine that requires them. Using fixed network delays, the round-trip time of a request between tenant and provider is set to 10 ms. Tests are run sequentially and PDP evaluation is done in a single thread. Each test starts with 500 warm-up requests and is repeated until the confidence interval lies within 2% of the sampled mean for a confidence level of 95%.

6.3 Results

Figure 7 shows the results of the performance tests. Because the federation algorithm does not take into account the frequency of each request, we do not state means over all requests, but list the results for each access request separately.

We can make several observations from the figure. First, provider-side evaluation requires the same or larger number of remote requests than tenant-side and federated evaluation in all cases, leading to longer evaluation times in most cases. This is caused by the fact that the policies from the case study require more tenant attributes than provider attributes. Request 13 is the most extreme case, where all required attributes are stored tenant-side and 7 attribute requests are replaced by a single policy evaluation request.

Second, in most cases, federated evaluation leads to the same or smaller number of remote requests than tenant-side evaluation. The same number is achieved if P_9 (i.e., the part of the policy tree that is deployed provider-side) is not required to reach an access control decision, e.g., for requests 13 to 16. Smaller numbers are achieved in the other cases, e.g., requests 4 to 7. In these cases, multiple attribute fetches from tenant to provider are replaced by a single policy evaluation request. This shows the intended results of the federation algorithm. However, the smaller number of remote requests does not lead to proportionally shorter evaluation times, e.g., for requests 4, 5 and 6. This is caused by the larger overhead of a policy evaluation versus an attribute fetch, while the federation algorithm assumed both to be equal. In requests 24 and 25, tenant-side and federated evaluation even perform worse than provider-side evaluation because of this.

Finally, for requests 8 and 22 to 26, federated evaluation leads to larger numbers of remote request and longer evaluation times than tenant-side evaluation. This is caused by the fact that P_9 is evaluated, but all attributes required to come to a decision are already cached. Thus, federated evaluation requires a policy evaluation request, while tenant-side evaluation does not require any attribute fetches.

7 Discussion

In the previous sections, we presented the technique of policy federation, which aims to decompose access control policies over multiple parties for confidentiality and improved performance. In this section, we discuss the

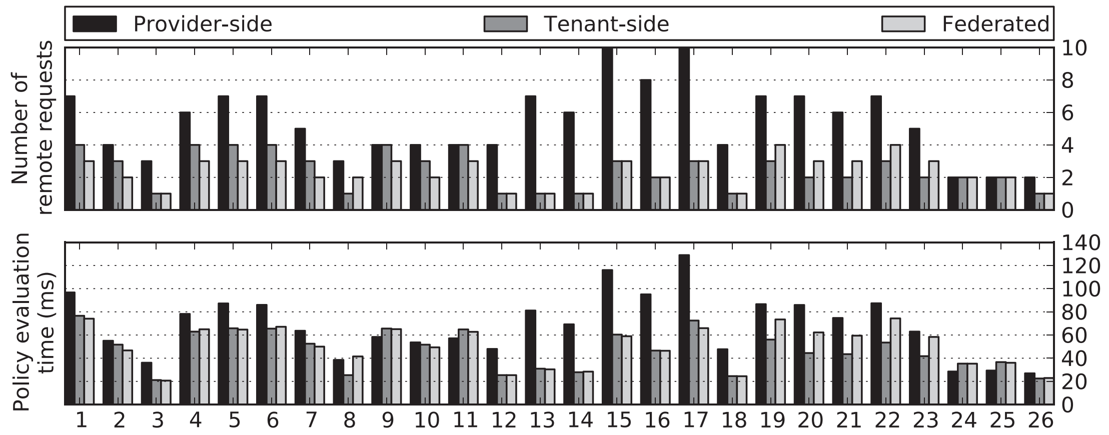

Figure 7 Results of the performance tests. The upper chart shows the number of remote requests needed for evaluating the policies (lower is better), the lower chart shows the resulting policy evaluation time in milliseconds (lower is better). For each access request, we show the results for provider-side evaluation, tenant-side evaluation and federated evaluation. As shown, the federated policy provide the best results for most access requests.

results of this work and in which ways it can be refined and extended.

7.1 Confidentiality

Policy federation effectively succeeds in keeping the sensitive tenant attributes and policies confidential. However, two potential threats to this work are (i) the increased attack surface of the tenant by the introduction of the RPDP service and (ii) possible inference of policies or attributes by the provider using the complete set of access requests and decisions. For the former, we argue that the risk of the increased attack surface is low since only the provider should be given access to the RPDP service. For the latter, we argue that the possibly inferred knowledge is limited since both the tenant policies and the required attributes remain confidential and the provider can only request the tenant to evaluate the policies resulting from the federation algorithm. However, future work is required to answer this question more quantitatively, for example using techniques such as logical abduction.

Towards the future, the employed confidentiality model can be refined. The algorithm now assumes that an attribute or policy is labeled sensitive or non-sensitive. In a more extensive case, a sensitivity policy could express more complex rules, for example, limiting attribute release to some parties based on their identity or defining a certain combination of multiple attributes as confidential.

7.2 Performance

The performance evaluation showed that policy federation has the ability to improve policy evaluation performance. With the maturation of policy-based and attribute-based access control, access control policies will only grow in both size and complexity and the performance gain of policy federation can be expected to increase as well.

In order to achieve further improved results, the algorithm can be refined in several ways. First, remote policy references can be extended with local targets in order to avoid the unnecessary policy requests mentioned in Section 6.3. Second, the algorithm achieves sub-optimal results because of the overhead of a policy evaluation versus an attribute fetch. While policy evaluation engines are expected to provide improved performance towards the future (e.g., [18]), the cost functions in the algorithm can be refined to take into account this overhead. As a further extension, performance properties of the provider and tenant infrastructures can be taken into account as well. Finally, the algorithm now only statically reasons about policies. In order to further optimize towards common access requests, the algorithm can be applied at run-time, thereby incorporating run-time statistics.

7.3 Obligations and attribute updates

Another part of future work is to incorporate obligations, i.e., actions which should be performed in conjunction with enforcing the access control decision [12]. For example, obligations can be used to specify that the user should agree to a license agreement or that the policy infrastructure should write out a log, send an e-mail to an administrator or update an attribute value. In [14], the impact of incorporating obligations in federated authorization is described. However, similar to attributes and policies in the policy tree, the tenant can regard certain obligations as sensitive and thus, obligations should be incorporated in the process of policy federation as well.

An interesting subset of obligations are attribute updates. Attribute updates can be used to model history-based policies [19], e.g., a separation-of-duty policy that states that a member of the help desk cannot view both insurance and financial documents of a single organization or a policy that limits the number of views of a document. Both attribute updates and history-based policies introduce extra complexity in policy federation because (1) attribute updates require concurrency control in case of distributed policy evaluation [17] and (2) history-based policies are known to have a large impact on performance [19]. Both are therefore interesting tracks for future research.

7.4 Generalization to N > 2 parties

A final possible extension of this work is a generalization to more than two parties. This paper focused on a tenant renting access to a SaaS application and that tenant wanting to enforce tenant-specific access control policies on that application. This situation can be extended to more than two parties, e.g., a patient monitoring system provided to multiple hospitals which collaboratively provide care to the same patient. In our experience, this situation reduces to each hospital applying its specific policies to the shared application, in which case the algorithm can separately be applied to each hospital policy without change. Should a situation arise that does not show this pattern (i.e., a federation in which a single policy reasons about data of more than two parties), the algorithm should be extended. However, we do expect the techniques in this paper to apply to this situation as well.

8 Related work

This work describes rewriting and optimizing access control policies. In general, it has been inspired by the work on query optimization in database systems, which similarly discusses transformation rules, heuristic-based optimization and cost-based optimization for distributed execution. In essence, this work applies these techniques to the domain-specific tree-structured policy model described in Section 4. For an overview of this large body of work, we refer to [20]. Specifically in the domain of policy-based access control, several other authors have also focused on the problem of policy decomposition and distribution. Bauer et al. [21] describe a distributed system for constructing formal proofs, aimed at access control. Amongst others, they also briefly discuss tactics to take into account confidentiality of input data and to improve performance based on the location of the input data. This work extends and applies the general principles discussed in their work on practical policy trees to achieve an algorithm for policy federation. Ardagna et al. [16] focus on controlled disclosure of sensitive access control policies and also discuss policy decomposition and transformation rules. However, their goal is to provide a limited view on sensitive policies. Therefore, their approach does not maintain policy equivalence and does not directly apply to our goal. Finally, the work of Lin et al. [22] sketches a theoretical framework for policy decomposition and distribution based on performance and confidentiality requirements. Their goal is similar to ours and their work has been an important influence. However, they describe a theoretical approach based on a simplified policy model, limiting applicability. Thus, this work extends theirs with a more widely-applicable policy model, a description of supporting middleware and a real-life evaluation.

Several other authors have also investigated the problem of confidentiality-aware access control for outsourced applications and other solutions exist. For example, Asghar et al. [23] employ attribute and policy encryption, extending the work of di Vimercati et al., e.g., [24]. This approach is dual to policy federation and should allow all tenant data to be securely shared with the provider, but also introduces performance overhead and is still limited in policy expressivity, for example only being able to compare attributes with literal values.

Finally, this work fits in a growing collection of performance-enhancing tactics for policy-based and attribute-based access control. This work builds upon the idea of improving policy evaluation performance by focusing on attribute fetching, as first introduced by Brucker and Petritsch [25]. Policy federation can be complemented with the work of several other authors, e.g., Wei et al. [26], who focus on decision caching and Gheorghe et al. [27], who focus on infrastructure reconfiguration for optimal attribute retrieval and cross-request attribute caching.

9 Conclusions

In this paper we described access control for SaaS applications and focused on the challenges of confidentiality-aware and efficient policy evaluation, as motivated by an e-health case study. We proposed to address these challenges by decomposing and distributing the tenant-specific policies across tenant and provider in order to keep sensitive tenant data local while evaluating parts of the policies near the data they require as much as possible. This process, we call *policy federation*. We defined a widely-applicable attribute-based policy model, described an algorithm for policy federation in detail and elaborated on the design of supporting technology. Our approach succeeds in keeping the sensitive tenant data confidential and has the ability to improve policy evaluation time as well. This work fits in a growing collection of performance techniques for policy-based and attribute-based access control. With the maturation of these technologies and the growing ecosystem of service-oriented business coalitions, we believe that the need for federated access control and for policy federation in particular will only grow.

Endnotes

[a]We first discussed this concept in [28].

[b]For this, we randomly constructed an artificial policy tree of five levels, each composed policy having a branching factor of three and each policy requiring five random attributes.

Abbreviations

ABAC: Attribute-based access control; HPMS: Home patient monitoring system; PAP: Policy administration point; PDP: policy decision point; PEP: Policy enforcement point; PIP: Policy information point; RPDP: Remote policy decision point.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

MD carried out the definition of the policy model and the implementation of the prototype. MD and BL collaboratively carried out the design of the policy federation algorithm and the performance evaluation. MD, BL and WJ collaboratively carried out the case study analysis and the the conceptual and architectural design of our solution. All authors read and approved the final manuscript.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, by the EU FP7 project NESSoS and by the Agency for Innovation by Science and Technology in Flanders (IWT). With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

Received: 10 May 2013 Accepted: 13 January 2014
Published: 11 February 2014

References

1. Mell P, Grance T (2009) The NIST definition of cloud computing. Natl Ins Standards Tech 53(6): 50
2. Centralizing Information on a Global Scale: Cisco Deploys Salesforce to 15,000 Users with Siebel Integration and PRM Capabilities. http://www.salesforce.com/uk/customers/hi-tech-hardware/cisco.jsp (2009)
3. E-Health Information Platforms (E-HIP). http://distrinet.cs.kuleuven.be/research/projects/E-HIP (December 2013)
4. Healthcare professional's collaboration Space (Share4Health). http://distrinet.cs.kuleuven.be/research/projects/Share4Health (December 2013)

5. Security Assertion Markup Language (SAML) v2.0. http://www.oasis-open.org/standards#samlv2.0 (March 2005)
6. OpenID Authentication 2.0 - Final. http://openid.net/specs/openid-authentication-2_0.html (December 2013)
7. U. S. Department of Health and Human Services (1996) Health insurance portability and accountability act (HIPAA). Retrieved from http://www.hhs.gov/ocr/privacy/hipaa/understanding/index.html
8. European Commision (1995) Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Retrieved from http://old.cdt.org/privacy/eudirective/EU_Directive_.html
9. Latham D (1985) Department of Defense Trusted Computer System Evaluation Criteria. Tech. rep., US Department of Defense
10. Ferraiolo DF, Sandhu R, Gavrila S, Kuhn DR, Chandramouli R (2001) Proposed NIST standard for role-based access control. ACM Trans Inf Syst Secur 4(3): 224–274. http://doi.acm.org/10.1145/501978.501980
11. Jin X, Krishnan R, Sandhu R (2012) A unified attribute-based access controls model covering DAC, MAC and RBAC. In: Data and applications security and privacy XXVI. Springer, Berlin, Heidelberg, pp 41–55. http://dx.doi.org/10.1007/978-3-642-31540-4_4
12. Moses T (2005) eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
13. Maarten Decat - Policy Federation. https://distrinet.cs.kuleuven.be/software/policy-federation/
14. Decat M, Lagaisse B, Van Landuyt D, Crispo B, Joosen W (2013) Federated authorization for software-as-a-service applications. In: On the move to meaningful internet systems: OTM 2013 Conferences. Springer, Berlin, Heidelberg, pp 342–359
15. Crampton J, Huth M (2010) An authorization framework resilient to policy evaluation failures. In: Proceedings of the 15th European Conference on Research in Computer Security. Springer-Verlag, Berlin, Heidelberg, pp 472–487. http://dx.doi.org/10.1007/978-3-642-15497-3_29
16. Ardagna C, Capitani di Vimercati S, Foresti S, Neven G, Paraboschi S, Preiss FS, Samarati P, Verdicchio M (2010) Fine-grained disclosure of access policies. In: Soriano M, Qing S, Lopez J (eds) Information and communications security. lecture notes in computer science, vol. 6476. Springer, Berlin, Heidelberg, pp 16–30. http://dx.doi.org/10.1007/978-3-642-17650-0_3
17. Decat M, Lagaisse B, Crispo B, Joosen W (2013) Introducing concurrency in policy-based access control. In: Proceedings of the 8th workshop on middleware for next generation internet computing. ACM, New York, pp 3:1–3:6
18. Liu AX, Chen F, Hwang J, Xie T (2008) Xengine: a fast and scalable xacml policy evaluation engine In: Proceedings of the 2008 ACM SIGMETRICS. SIGMETRICS '08. ACM, Annapolis, MD, USA, pp 265–276. http://doi.acm.org/10.1145/1375457.1375488
19. Gama P, Ribeiro C, Ferreira P (2006) A scalable history-based policy engine. In: Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on. IEEE, pp 100–112. http://doi.ieeecomputersociety.org/10.1109/POLICY.2006.8
20. Elmasri RA, Navathe SB (1999) Fundamentals of database systems, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
21. Bauer L, Garriss S, Reiter M (2005) Distributed proving in access-control systems. In: Security and Privacy, 2005 IEEE Symposium on. IEEE Computer Society, Los Alamitos, pp 81–95
22. Lin D, Rao P, Bertino E, Li N, Lobo J (2008) Policy decomposition for collaborative access control. In: Proceedings of the 13th ACM SACMAT. ACM, New York, pp 103–112
23. Asghar M, Ion M, Russello G, Crispo B (2011) Espoon: Enforcing encrypted security policies in outsourced environments. In: Availability, Reliability and Security (ARES), 2011 Sixth International Conference on. IEEE Computer Society, Los Alamitos, pp 99–108
24. di Vimercati SDC, Foresti S, Jajodia S, Paraboschi S, Samarati P (2007) A data outsourcing architecture combining cryptography and access control. In: Proceedings of the 2007 ACM workshop on computer security architecture, CSAW '07. ACM, Fairfax, Virginia, USA, pp 63–69. http://doi.acm.org/10.1145/1314466.1314477
25. Brucker A, Petritsch H (2010) Idea: efficient evaluation of access control constraints. In: Engineering Secure Software and Systems. Springer, pp 157–165. http://dx.doi.org/10.1007/978-3-642-11747-3_12
26. Wei Q (2009) Towards improving the availability and performance of enterprise authorization systems. Ph.D. thesis, University of British Columbia
27. Gheorghe G, Crispo B, Carbone R, Desmet L, Joosen W (2011) Deploy, adjust and readjust: Supporting dynamic reconfiguration of policy enforcement 7049: 350–369. http://dx.doi.org/10.1007/978-3-642-25821-3_18
28. Decat M, Lagaisse B, Joosen W (2012) Toward efficient and confidentiality-aware federation of access control policies. In: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing. ACM, Montreal, Quebec, Canada, pp 4:1–4:6. http://doi.acm.org/10.1145/2405178.2405182

doi:10.1186/1869-0238-5-1
Cite this article as: Decat *et al.*: **Middleware for efficient and confidentiality-aware federation of access control policies.** *Journal of Internet Services and Applications* 2014 **5**:1.