KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

# Authorization middleware for Software as a Service

**Maarten Decat**

Supervisor:
Prof. dr. ir. W. Joosen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science: Computer Science

January 2016

# Authorization middleware for Software as a Service

**Maarten DECAT**

Examination committee:
Prof. dr. A. Bultheel, chair
Prof. dr. ir. W. Joosen, supervisor
Dr. B. Lagaisse
Prof. dr. ir. F. Piessens
Dr. E. Truyen
Prof. dr. B. Crispo
Prof. dr. ir. B. Preneel
Prof. dr. ir. F. De Turck
(University of Ghent)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science: Computer Science

January 2016

Uitgegeven in eigen beheer, Maarten Decat, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Voor Opa. Je zou zo trots geweest zijn.

# Acknowledgments

While this thesis carries my name, it would not have been possible without the support, feedback and guidance of the numerous wonderful people around me. To these people, I want to express my sincere gratitude.

First of all, I want to thank my supervisor Wouter Joosen. If I would have to summarize my last years at DistriNet in one word, it would be "opportunities" and this is largely due to the efforts of Wouter. Because of him, I have joined DistriNet, worked with some of the most intelligent people I have ever met, discussed my research with industry, visited Canada, China and the USA and dug into entrepreneurship, all within an atmosphere of professionalism. Not all PhD researchers are granted these opportunities, for which I am grateful.

Secondly, I want to thank my coach Bert Lagaisse. He has guided me throughout this whole journey and has lifted my work to the next level. In addition, he has introduced me into the wonderful world of Burger King, Quick, KFC, Starbucks, Burgerfolie and many more. Thank you for the very enjoyable collaboration, this thesis is also your accomplishment.

Next, I want to thank the other members of my jury for the interesting feedback and insights on my work: Adhemar Bultheel, Frank Piessens, Eddy Truyen, Bruno Crispo, Bart Preneel and Filip De Turck. Also many thanks to the others who have spent time on proof-reading this thesis.

I also want to thank my many great colleagues at DistriNet. It truly is an awesome group of people, leading to inspiring discussions and fun DRADS days. A special thanks goes to Jasper Bogaerts, thank you for a rewarding collaboration the last years, which amongst others resulted into the Amusa middleware. A second special thanks goes to the members of Lantam for the often-intense-but-always-constructive meetings and discussions. A third special thanks goes to my many cool office mates of the last years such as Philippe, Kim, Stefan, Wouter, Bart, Bert, Jasper, Arnaud, Jasper, Ansar and Jan. Coming to work would not have been as much fun without you. A final special thanks goes to the administrative and technical staff of the department, you have made my life much easier.

The last years I have also taken some steps into entrepreneurship with Battle of Talents and SpaceBillboard. While these projects in hindsight look like baby steps, the effort at the time was substantial and it would not have been as enjoyable without my partners in crime Tjorven, Jeroen, Sofie and Piet. Buzzword bingo for the win!

Next I would like to express my gratitude towards my parents, grandparents, family and family in law. You have always supported me in everything I do, have sparked my curiosity, have taught me how to live. I would not be writing this if it was not for all of you, for which I am eternally grateful.

And last but definitely not least, I want to thank my wife Kathleen. Thank you for your patience and support in all my endeavors, it is not easy to live with a computer scientist. Thank you for making me a better person, I hope I can do the same for you.

– Maarten Decat

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

# Abstract

This thesis focuses on access control for Software-as-a-Service (SaaS) applications. Access control is the part of security that aims to constrain which users can access which data in an application by enforcing access rules. SaaS makes up a novel and promising type of applications in which a customer organization rents access to an entire application hosted *in the cloud* for use through a web browser.

Because SaaS applications are typically designed to be used by multiple customer organizations at the same time, application-level access control is of big importance to them. However, SaaS applications also pose new and specific challenges for access control. For example, SaaS access control should enable the provider of the application to control which of its customers can access which parts of the application and should enable these customers to control which of their employees can access which part of their data in the application. In addition, while this functionality by itself is non-trivial, SaaS access control is further complicated by the fact that every customer wants to express its access rules in terms of its own organizational structure, by the fact that SaaS applications are offered to a large amount of customers, and by the fact that these customers do not necessarily trust the provider completely.

As such, the goal of this thesis is to design access control techniques for SaaS applications that are able to cope with these challenges. In addition, these techniques should impose low performance overhead on the application, should be easy to use and should be easy to integrate into a SaaS application.

In this regard, this thesis provides four distinct contributions: (i) a reusable middleware for efficient access control management of SaaS applications, (ii) the concept of federated authorization, which externalizes access control from a SaaS application, (iii) the technique of policy federation, which improves the performance of federated authorization, and (iv) a technique to enable access rules to be securely evaluated in parallel to support large amounts of requests per second.

For each of these contributions, we build upon the state-of-the-art technologies of policy-based access control, attribute-based access control and tree-structured policies.

In addition, these contributions have been validated in four distinct case studies of realistic SaaS applications in the domains of automated document processing, automated workforce management and e-health. Finally, we have systematically evaluated these contributions in terms of performance and engineering overhead based on extensive prototypes.

# Beknopte samenvatting

Deze thesis behandelt toegangscontrole voor zogenaamde Software-as-a-Service (SaaS) applicaties. Toegangscontrole is het deel van beveiliging dat door het toepassen van toegangsregels bepaalt tot welke gegevens gebruikers toegang hebben. SaaS vormt een nieuw en veelbelovend type van applicaties waarbij een organisatie toegang huurt tot een gehele applicatie die aangeboden wordt vanuit *de cloud* en gebruikt wordt met een web browser.

Omdat SaaS applicaties typisch ontworpen worden om door meerdere klanten tegelijkertijd gebruikt te worden is toegangscontrole op het niveau van de applicatie van groot belang. SaaS applicaties bieden echter ook nieuwe en specifieke uitdagingen voor toegangscontrole. Zo moet toegangscontrole voor SaaS de aanbieder van een applicatie toelaten om te controleren welke van zijn klanten toegang hebben tot welk deel van de applicatie en moet toegangscontrole deze klanten op hun beurt toelaten om te controleren welke van hun werknemers toegang hebben tot welk deel van hun gegevens in deze applicatie. Hoewel deze functionaliteit op zich al niet triviaal is, wordt toegangscontrole voor SaaS nog complexer doordat elke klant zijn toegangsregels wilt uitdrukken in termen van de structuur van zijn eigen organisatie. Bovendien worden SaaS applicaties meestal aangeboden aan een groot aantal klanten en vertrouwen deze klanten de aanbieder van de applicatie mogelijk niet volledig.

Het doel van deze thesis is daarom toegangscontroletechnieken voor SaaS applicaties te ontwerpen die aan deze uitdagingen voldoen. Deze technieken mogen daarenboven slechts weinig invloed hebben op de performantie van de SaaS applicatie, moeten gemakkelijk te gebruiken zijn en moeten gemakkelijk te integreren zijn in een SaaS applicatie.

Binnen deze context biedt deze thesis vier contributies: (i) een herbruikbare *middleware* voor efficiënt beheer van toegangscontrole voor SaaS applicaties, (ii) de techniek van gefedereerde autorisatie, dewelke toegangscontrole externaliseert vanuit een SaaS applicatie, (iii) de techniek van *policy federation*, dewelke de performantie van gefedereerde autorisatie verbetert, en (iv) een techniek om meerdere toegangsregels zonder fouten parallel te evalueren om zo een groot aantal evaluaties per seconde te bereiken.

Voor elke van deze contributies bouwen we op de recente technologieën van policy-gebaseerde toegangscontrole, attribuutgebaseerde toegangscontrole en boomgestructureerde toegangsregels. Onze contributies zijn bovendien gevalideerd in vier uiteenlopende gevalanalyses van realistische SaaS applicaties in de domeinen van geautomatiseerd documentbeheer, geautomatiseerd personeelsbeheer en e-health. Daarnaast hebben we de performantie van deze contributies systematisch geëvalueerd op basis van uitgebreide prototypes.

# Abbreviations

| | |
|---|---|
| a | Action. Employed in Chapter 5. |
| ABAC | Attribute-based access control |
| DAC | Discretionary access control |
| e | Environment. Employed in Chapter 5. |
| EBAC | Entity-based access control |
| IaaS | Infrastructure as a Service |
| LBAC | Lattice-based access control |
| MAC | Mandatory access control |
| PaaS | Platform as a Service |
| PDP | Policy decision point |
| PEP | Policy enforcement point |
| PIP | Policy information point |
| PKI | Public Key Infrastructure |
| r | Resource. Employed in Chapter 5. |
| RBAC | Role-based access control |
| ReBAC | Relationship-based access control |
| RPDP | Remote policy decision point |
| s | Subject. Employed in Chapter 5. |
| SaaS | Sofware as a Service |
| UCON | Usage control |

# Glossary

Access control — The part of security that constrains the *actions* that *subjects* can perform on the *resources* in an application. Access control is generally divided into *authentication*, *authorization* and *audit*.

Access control policy — A software artifact in which access rules are declaratively expressed independently of how they are enforced.

Action — An operation performed on a *resource* in an application by a *subject*, e.g., reading a file or updating a database entry.

Attribute — A key-value property of a *subject*, a *resource*, an *action* or the environment, e.g., the roles of the *subject*, the owner of a *resource*, the identifier of an *action* or the current time.

Audit — The part of *access control* that checks the past *actions* that *authenticated subjects* have performed on the *resources* in an application and corrects any unauthorized actions.

Authentication — The part of *access control* that ensures that the *subject* is who he, she or it claims to be, for example by verifying the combination of a username and password.

Authorization — The part of *access control* that checks whether an *authenticated subject* is permitted to perform the requested *action* on the requested *resource* by evaluating access rules and blocking the action if not.

Federation — An organizational structure in which multiple organizations have set up collaboration agreements, but do not necessarily trust each other completely and remain separate domains in terms of security and administration.

Obligation

A statement of an operation that must be executed in conjunction with enforcing an access decision. For example, such an obligation can specify that a user should agree to terms and conditions before being granted access, that the system should write out a log of the access decision or that an attribute should be updated.

Policy

See *access control policy*.

Policy-based access control

An approach to *access control* in which the specification of the access rules is separated from the mechanisms that enforce them by expressing the rules in declarative *access control policies*.

Resource

An entity on which *actions* can be performed in an application, e.g., a file, a database, a database entry or a socket.

SaaS

See *Software-as-a-Service*.

Software-as-a-Service

One of the service models of cloud computing. With Software as a Service, *tenants* rent access to an entire application hosted by a provider for use through a thin client such as a web browser. Two well-known SaaS offerings are Google Drive and Salesforce.

Subject

An entity that can perform *actions* on the *resources* in an application, e.g., a human user, a remote machine on the internet or a process acting in name of a user.

Tenant

An organization that rents access to a cloud application. Every tenant represents and manages multiple end-users.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

During the last decades, information technology and software have penetrated every aspect of our society. We now use IT systems to pay our taxes over the web, perform wire transfers from our smart phones and share our daily lives with our friends. Behind the scenes, software now drives cars, flies planes, trades on the stock market and manages power plants. More and more information is digitized in order to ease administrative procedures and to enable applications such as the electronic health record.

As more operations are automated and more information is digitized however, security becomes of higher importance as well. For example, it is clear that we do not want any person or system to turn off the brakes of our car. Similarly, we only want ourselves, our spouse or authorized bank clerk to perform a wire transfer from our accounts. And finally, we do not want our insurance company to read all the details in our medical record. In summary, any application that manages data or infrastructure of some value requires some form of security.

There are many different facets to security and of these, this thesis focuses on *access control.* Access control aims to constrain which users can access which data in an application by enforcing access rules. As a simple example, such a rule can express that the account managers of a certain company are only permitted to access documents of the customers that are assigned to them.

Over time however, access control has grown increasingly complex and challenging. For example, organizations such as modern hospitals currently treat thousands of patients every day and employ nurses, physicians and supporting staff that are structured in teams, projects and shifts and are possibly spread across multiple physical and autonomous departments. This leads to the challenge of designing access control techniques that are powerful enough to express all of these concepts and support these complex scenarios. At the same time however, these techniques should still be easy to manage, should not slow the software down, should be easy to incorporate in the underlying code and should be guaranteedly secure. As such, access control is faced with the continuous challenge of increasing expressivity and supporting more complex usage scenarios while maintaining usability, performance and security.

This thesis contributes to further addressing this challenge. More specifically, this thesis focuses on access control for Software as a Service or SaaS. SaaS makes up a novel and promising type of applications in the domain of cloud computing, but also poses specific challenges for access control. As such, the goal of this thesis can be summarized as:

*This thesis aims to provide techniques that enable and facilitate access control in Software-as-a-Service applications.*

In the rest of this chapter, we first introduce access control and Software as a Service. We then identify the challenges for access control in SaaS and present the goals of this thesis. Next, we present the approach taken in this research and discuss the four major resulting contributions. Finally, this chapter concludes with an overview of the structure of the rest of this thesis.

## 1.1 Access control and Software as a Service

This thesis focuses on access control for Software as a Service. In this section, we first introduce access control, then introduce Software as a Service and finally discuss the need for access control in Software-as-a-Service applications.

### 1.1.1 Access control

Access control is the part of security that constrains the actions that are performed on the data in a system by enforcing access rules.

Because of the importance of access control in software, it has been subject to research for decades. As a result, the research on access control is both broad and deep. Amongst others, a lot of effort is spent on designing models to efficiently and correctly specify the permissions of users in a system, e.g., lattice-based access control [143], role-based access control [100], attribute-based access control [123] and more recent advances such as usage control [171] and relationship-based access control [115, 104].

These models have then led to formal definitions of their properties (e.g., [44, 49]), to supporting administrative models (e.g., [183]) and methods such as role mining [138].

Also, a lot of research effort is spent on making it easier to incorporate access control in application code. Amongst others, this has led to the approach of policy-based access control [190, 181] in which the access rules are declaratively specified in so-called policies. This in turn has led to research on languages for expressing these policies (e.g, Ponder [79], XACML [117] and SecPAL [42]), on combining policies of multiple parties (e.g., [54]) and on expressing specific rules such as separation-of-duty [56].

In addition, access control research also encompasses secure authentication (e.g., using cryptographic keys [191]), performance of access control (e.g., [152, 134, 76]), automatic placement of access control code in application code (e.g., [166]), formal models of access control systems (e.g., [186, 111]), enabling efficient access control across multiple organizations (e.g., [173, 2]), supporting concepts such as dynamic delegation of rights (e.g., [75]) and applying all of these concepts to specific technologies such as workflows (e.g., [73]) and databases (e.g., [124]). Of course, this listing is still far from complete.

Because of our background in industrial case studies and research projects, this thesis focuses on whether the existing techniques in this broad domain can be applied to address the access control challenges specific to SaaS and how they should be improved to do so. Moreover, this thesis focuses on *application-level access control*, as opposed to for example language-level access control or database-level access control. In application-level access control, the access rules are expressed in terms of the concepts that are present in the application and the organizations that employ it. Chapter 2 goes deeper into the access control research on which this thesis builds.

### 1.1.2 Software as a Service

Software as a Service is one of the three service models identified in cloud computing [161]. Briefly said, cloud computing is a recent paradigm in which one or more customer organizations called *tenants* employ computing resources ("the cloud") hosted and operated by a *provider* over the internet as a service. The latter means that tenants are able to rapidly sign up for the service, provision new resources with minimal or no human interaction of the provider and only pay for the resources that they have consumed (a "pay-per-use" cost model). Depending on the type of computing resources offered by the provider, three service models are identified:

1. With *Infrastructure as a Service* or *IaaS*, the provider offers a pool of fundamental computing resources such as storage or processing. A well-known IaaS offering is Amazon EC2 [20], which allows tenants to build their own infrastructure on top of the physical infrastructure of Amazon in the form of virtual machines.

2. With *Platform as a Service* or *PaaS*, the provider offers a platform with specific libraries and services for tenants to deploy and run their own applications on. A well-known PaaS offering is Google AppEngine [19], which allows tenants to run their applications written in Java, Python, PHP or Go and provides specific services to easily scale out these applications.

3. With *Software as a Service* or *SaaS*, the provider offers an entire application for use through a thin client. Two well-known SaaS offerings are Google Drive [22] and Salesforce [21], which respectively provide an office suite and a suite for customer relationship management, both for use in a web browser.

This thesis focuses on the third of these three service models, i.e., SaaS. The market for SaaS has been growing substantially in the previous years and is expected to continue to do so (e.g., see [13]), amongst others because of the ease of use of SaaS applications and the ease of providing a SaaS application to multiple customers around the world. SaaS can also play an important role for the industry in Flanders and is gaining traction, as illustrated by the growing number of SaaS providers in this region in industries such as document processing [16], workforce management [16], e-health [15, 11] and security [26, 8].

### Key characteristics of SaaS

In essence, SaaS applications are characterized by three characteristics: (i) the fact that they employ application-level multi-tenancy, (ii) their scale and (iii) their nature as outsourcing.

**Characteristic 1: application-level multi-tenancy.** Firstly, SaaS applications are typically provided to and employed by multiple tenants at the same time, a concept called *multi-tenancy* [119, 67]. More precisely, multi-tenancy is an architectural strategy that aims to lower the operational costs of a service by means of economies of scale. In earlier paradigms, the provider of a service would have developed and set up a dedicated instance of this service for each individual tenant, which leads to sub-optimal usage of the underlying hardware and multiple variants of the same code base that all have to be maintained. Multi-tenancy attempts to address these issues by having multiple tenants share the underlying resources of the service in order to increase hardware sharing and maintain only a single code-base. More specifically for SaaS, multi-tenancy aims to have tenants share the full application stack, a concept called *application-level multi-tenancy* (see Figure 1.1).

**Characteristic 2: large scale.** Secondly, SaaS applications are aimed for large amounts of tenants. Amongst others, this is a consequence of the fact that multi-

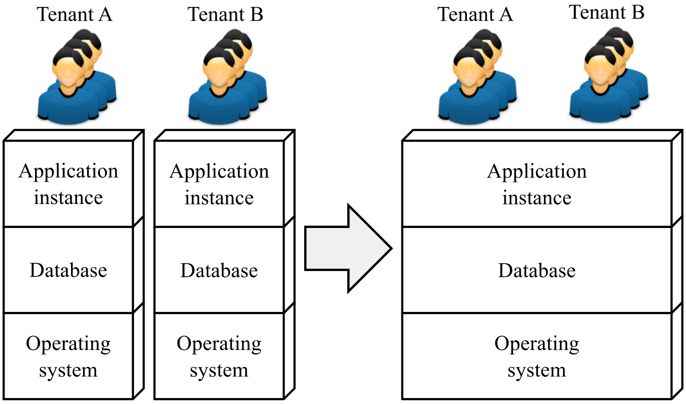


Figure 1.1: An illustration of application-level multi-tenancy: instead of developing and setting up a dedicated instance of the SaaS application for each individual tenant, multi-tenancy aims to have multiple tenants share a single application stack in order to increase resource sharing and maintain only a single code-base [119, 67].

tenancy lowers the overhead of accepting a new tenant for the provider and thereby enables the provider to aim for a large amount of smaller tenants [67]. This focus on a large amount of tenants has two consequences. Firstly, in order to be able to manage this large amount of tenants, SaaS applications rely on self-management. More precisely, new tenants should be able to sign up for the application and manage it themselves with as little human interaction of the provider as possible. Secondly, a large amount of tenants leads to a large amount of users and a large amount of requests to the application. In order to support this throughput, SaaS applications are deployed on a distributed infrastructure and the software is engineered to be able to scale out with low latency overhead.

**Characteristic 3: SaaS is outsourcing.** Finally, SaaS is a form of outsourcing as the tenants essentially outsource the hosting and management of the SaaS application to the provider. For the tenants, this lowers their required in-house IT infrastructure and IT personnel. This characteristic also has two consequences. Firstly, the data of the tenants in the SaaS application and the software that manages this data are both located in the infrastructure of the provider, i.e., outside of the premises of the tenant. Secondly, while the tenants trust the SaaS provider with the data in the application itself, they do not necessarily trust the SaaS provider with any more than that and may want additional assurances from the provider, such as whether no data was lost.

As we will see, these three characteristics all pose challenges for SaaS access control.

### 1.1.3 The need for security and access control in SaaS

Our focus is on security for SaaS. For the tenants, this security should protect their data in the application from being read or altered by unauthorized persons and systems. For the provider, this security should make sure that only the appropriate, e.g., paying, tenants are able to access the SaaS application and should protect it from being made unavailable.

To some extend, SaaS security can build on similar security techniques as other (web) applications. For example, secure channels such as HTTPS can be used to protect the data sent to and from the SaaS application. Additionally, network-level access control techniques such as firewalls are required to protect the SaaS infrastructure from unrestricted access. Also, secure authentication techniques such as multi-factor authentication can be used to protect against password theft. And finally, existing countermeasures can be applied to protect the application against well-known code-level vulnerabilities such as SQL injection.

What is specific to SaaS however is the large need for application-level access control. Similar to other types of applications, the provider requires this access control to restrict the access of tenants to the application and the tenants require it to restrict which of their users can access which of their data in the application. The additional reason for the need for application-level access control in SaaS however is application-level multi-tenancy. More specifically, when an application is shared by multiple tenants, an important security requirement is to make sure that one tenant cannot access the data of another tenant, a form of tenant isolation [119]. Because all tenants of a SaaS application share the same code, the same application instances and the same underlying database, this separation cannot be enforced architecturally any more, i.e., it is not possible any more to provide dedicated application instances and a dedicated database to each tenant and isolate these on different dedicated networks. As a result, SaaS applications rely on application-level access control to separate tenants.

## 1.2 Challenges for access control in SaaS

Access control is an essential part of security for SaaS. However, SaaS access control also faces specific challenges.

### 1.2.1 Functional challenges

Based on the discussion of the previous section, SaaS access control should support the following combination of functionality:

1. Tenant isolation: Access control for SaaS should separate the different tenants in a SaaS application from each other, except when explicitly permitted.
2. Constraining tenants: Access control for SaaS should allow the provider to restrict which tenants can access the application[1].
3. Constraining end-users: Access control for SaaS should allow the tenants to restrict which of their own users, e.g., its employees or customers, can access which part of their data in the application.

### 1.2.2 Non-functional challenges

While it is non-trivial to support the above functionality in the first place, each of the three essential characteristics of SaaS listed in Section 1.1.2 leads to additional non-functional challenges for access control as well.

**Non-functional challenges from multi-tenancy.** While multi-tenancy leads to the functional requirement of enabling the tenants to restrict their own users, every tenant in addition wants to be able to enforce access rules specific to its own domain and organization. For example, a hospital will reason about physicians and departments, while a bank will reason about bank clerks, assigned customers and shifts. As a result, multi-tenancy requires to *support access rules consisting of a wide variety of access control concepts, of which some are even domain-specific.* This challenge is a specific instance of the broader challenge of tenant variability [47, 119, 195]. Moreover, as a SaaS application can be used by a large number of tenants, it would not be feasible with respect to availability to have to restart or even recompile the application for every new tenant or every change in access rules. As a result, multi-tenancy requires to be able to *specify the access rules independently of the application code* and to be able to *modify them without having to recompile or restart the application.*

**Non-functional challenges from the large scale.** SaaS applications aim for a large amount of tenants. As a result, SaaS applications are designed to scale out on a distributed infrastructure and they rely on self-management. Both characteristics lead to challenges for access control. Firstly, access control should also be designed to be able to *scale to the throughput of the application itself.* The reason for this is that access control should be enforced on most, if not any, request to the application

[1] Notice that most SaaS providers also have their own users that access the application, e.g., employees such as help desk operators. SaaS access control should also allow the provider to restrict the access of these users to the SaaS applications. In this thesis, we do not explicitly take this requirement into account as this can be addressed by modeling the provider as a tenant that can restrict its own users as per requirement 3 above.

and should not hinder the scalability of this application. Secondly, access control should *function correctly in the context of a distributed system*, e.g., in the presence of partial failures and concurrency. Thirdly, the *self-management* should also hold for access control. More precisely, tenants should be able to configure their own users and their rights autonomously. However, this self-management should still be *secure* and guarantee, for example, that the tenants cannot configure access rules that allow them to access the data of other tenants.

**Non-functional challenges from outsourcing.** SaaS in essence is a form of outsourcing. As a result, the data in a SaaS application is located outside of the premises of the tenant and the tenant does not necessarily trust the provider with data other than that in the SaaS application. In this case, especially the latter leads to challenges for access control. More precisely, in some cases, the tenant may want to enforce certain access rules on the SaaS application, but does not want to disclose the *sensitive access control data* required to evaluate them or even may not be permitted to by law. This is most clear in privacy-sensitive application domains such as e-health where policies often reason about patient-physician relationships, consent or the condition of a patient. In addition, the tenant may not want to disclose *sensitive access rules* themselves, for example in case of rules that reason about its competitors.

### 1.2.3 Additional concerns

The previous listed the SaaS-specific non-functional challenges for SaaS access control. In addition to these, we also take into account three concerns that should hold for every software system: (i) low management overhead, (ii) low performance overhead and (iii) low software engineering overhead.

**Concern: low management overhead.** If an access control technique requires a large management effort of the provider or of a tenant, it is less likely to be used in practice. Additionally, while a SaaS application is used by multiple tenants, it should also be feasible for each tenant to use multiple SaaS applications. Therefore, we take into account the concern that the techniques designed in this thesis should *impose only low management overhead* and if possible, should even *lower the management overhead of access control*. Finally, in order to enable organizations to efficiently express their access rules, the technologies developed in this thesis should also *support fine-grained access rules* that combine multiple access control concepts such as identity-based permissions, groups, hierarchic roles, time, location and separation of duty (we explain these concepts in more detail in Chapter 2).

**Concern: low performance overhead.** Similarly to the management overhead, if an access control technique damages the performance or usability of the application on which it is enforced, that technique will not be used in practice. We already explained the requirement of scalability and throughput to cope with a large amount of tenants. In addition, we also take into account the concern that the techniques designed in this thesis should *impose limited latency overhead.*

**Concern: low engineering overhead.** Finally, in a similar reasoning as before, we also take into account the concern that the techniques designed in this thesis should *require little engineering effort for the developer of a SaaS application.* Especially for complex functionality such as multi-tenant access control, our work should make it easier to support and incorporate this functionality in a SaaS application.

## 1.3 Goals of this thesis

The goal of this thesis is to enable and facilitate access control in Software-as-a-Service applications by studying, designing and evaluating access control techniques that address the challenges discussed in the previous section. More specifically, this thesis focuses on four sub-sets of these challenges, leading to four more specific goals:

**Goal 1: address the challenges from multi-tenancy.** Our first goal is to design an access control technique that addresses the challenges from multi-tenancy. In the first place, this technique should enable the provider and all the tenants to enforce their specific access rules on the shared multi-tenant SaaS application and should isolate the multiple tenants in this application. In addition, this technique should support this functionality with low performance overhead and it should impose low management overhead on the tenants and provider. As such, this goal focuses on the challenge of multi-tenancy and the concerns of low performance and management overhead.

**Goal 2: lower the management overhead for tenants.** The previous goal effectively is to enable tenants to express their access rules on a SaaS application. However, when the tenants have to configure their access rules in the SaaS application itself, their overall access management is scattered across the multiple SaaS applications that they use, which in turn leads to a large administrative overhead and eventually to inconsistencies and security bugs.
Therefore, our second goal is to design an access control technique that lowers the management overhead for tenants. More precisely, this technique should enable

tenants to centrally manage access control for all SaaS (and on-premise) applications that they employ. As such, this goal focuses on the concern of low management overhead for the tenants.

**Goal 3: limit the disclosure of sensitive access control data and rules.** Our first goal was to enable tenants to express their access rules on the SaaS application. If the provider also evaluates the rules of its tenants, these tenants are forced to disclose their access rules and the data required to evaluate them to the SaaS provider. However, tenants do not necessarily trust the provider with these rules or data.
Therefore, our third goal is to design an access control technique that enables tenants to enforce their access rules on a SaaS application without having to disclose these rules nor the data required to evaluate them with the provider of that application. In addition, this technique should provide this functionality with low performance overhead. As such, this goal focuses on the challenge of limited trust in the SaaS provider.

**Goal 4: enable policy evaluation to securely scale out.** Finally, our fourth goal is to enable policy evaluation to scale to the size of the application that it constrains. To achieve this, policies should be evaluated concurrently and distributedly. However, for certain classes of policies such as history-based policies, one access decision depends on the previous ones. As a result, concurrency can be exploited to achieve incorrect access decisions and privilege escalation. Moreover, general techniques for concurrency control in databases are not able to address this issue and scale to the size of current applications at the same time.
Therefore, our fourth goal is to design an access control technique that avoids incorrect access decisions due to concurrency, is able to scale to a large number of machines and incurs only a limited latency overhead. As such, this goal focuses on the challenge of the large scale of SaaS applications and the concern of low performance overhead.

In addition to these four goals, we also aim to provide the designed access control techniques as **reusable middleware**, i.e., as software that fulfills and encapsulates complex (distributed) functionality in a way that can easily be employed by an application through well-defined APIs. This additional goal stems from the concern of low engineering overhead. As a result of this goal, the techniques designed in this thesis encompass end-user interfaces and domain-specific languages as well as the underlying run-time environment and software engineering techniques.

As we will see later on, these goals have led to four distinct contributions of this thesis.

## 1.4 Research approach

Our approach for achieving the goals listed in the previous section is characterized by three key properties: (i) the requirements analysis and validation is driven by case studies, (ii) we deliberately build upon existing state-of-the-art technologies and (iii) we employ prototype-based evaluations.

### 1.4.1 Case studies

As a first characteristic of our approach, our research is heavily driven by realistic case studies of SaaS applications. Each of these case studies resulted from a research project in collaboration with other research partners and companies from industry. This approach allowed us to gain insights in the access control challenges of real-life SaaS applications. In essence, the challenges stated in the previous section were identified in these case studies. In addition, this approach allowed us to validate our contributions, amongst others based on a realistic set of access rules derived from these case studies and the feedback from the industry partners in the research projects.

More precisely, this research is driven by four distinct case studies:

**Case study 1: eDocs – automated document processing.** Our first case study is a SaaS application provided by a company that we call eDocs in this thesis. This SaaS application enables large companies such as banks and press agencies to efficiently generate and distribute large amounts of digital documents such as pay checks and invoices to their respective users and customers. In addition, the recipients can read and manage all their received documents using the application.
This case study is based on the ICON research project called PUMA (Permission, User Management and Availability for multi-tenant SaaS applications [16]). In combination with eWorkforce (see next), it mainly influenced our work on Amusa by identifying the complexity of multi-tenant access control and the challenge for SaaS providers to build an access control system that supports this functionality. In addition, this case study has led to an extensive set of realistic access rules that illustrate the importance of history-based policies. This case study is further explained in Chapter 3 and was analyzed in-depth in a separate technical report [81].

**Case study 2: eWorkforce – automated workforce management.** Our second case study is a SaaS application provided by a company that we call eWorkforce in this thesis. This SaaS application enables its tenants to automatically plan the workflows for their product and service appointments, e.g., install and repair jobs for

telecom operators, utility companies and retailers. eWorkforce assigns the resulting appointments to technicians who receive their appointments through a mobile application and afterwards report task progress and consumed resources such as cables and devices.
Similarly to eDocs, this case study is based on the PUMA research project [16], it mainly influenced our work on Amusa and it has led to an extensive set of realistic access rules. This case study is also further explained in Chapter 3 and was analyzed in-depth in a separate technical report [82].

**Case study 3: home patient monitoring.** Our third case study is a SaaS application provided to hospitals for monitoring patients at their homes. More precisely, the patients are monitored using a chest band and the measurements are sent to the SaaS application. Telemedicine operators then check the patient's status and notify the patient's physician at the hospital in case of important evolutions. A patient's status can also be viewed by other physicians and nurses and by the patients themselves.
This case study is based on a number of research projects [11, 12]. This case study mainly influenced our work on federated authorization and policy federation by illustrating the need for security and privacy in the domain of e-health, the need for scalable access control management in organizations such as large hospitals and the complex access rules that result from these organizations. This case study is further explained in Chapter 4 and Chapter 5 zooms in on a set of rules from this case study.

**Case study 4: collaborative care.** Finally, our fourth case study is a collaborative software platform that aims to streamline home care by improving the communication between the multiple involved organizations such as general practitioner practices, elder homes, home nursing organizations, hospitals and catering services. The platform therefore digitizes the information that these organizations share, such as prescriptions of and notes about the care receiver.
This case study was based on the ICON research project called O'CareCloudS (Organizing Care through trusted Cloudy-like Services, [15]). In essence, this case study extends the concept of SaaS because more than two organizations collaborate in this application. This leads to more complex relationships between the organizations and thus different challenges for access control. This case study has mainly influenced our work on federated authorization and is further explained in Chapter 4.

### 1.4.2 Supporting technologies

As a second characteristic of our research approach, it is incremental in the sense that we deliberately build upon three technologies available in the state of the art: policy-based access control, attribute-based access control and tree-structured policies.

These technologies already partially address some of the challenges described in Section 1.2 and thereby allow us to focus on the challenges specific to SaaS. The next chapter discusses each of these technologies in more detail, here we summarize them briefly:

**Policy-based access control.** Policy-based access control is an approach in which the specification of the access rules is separated from the mechanisms that enforce them in the application. As such, they can be modularized and externalized from the application that they constrain to express them in declarative *access control policies* [181]. We employ policy-based access control to enable the tenant and provider to specify their own rules without having to recompile or restart the application. In addition, we employ the reference architecture for policy-based access control systems (see Section 2.3.2) to define and discuss the architecture of our contributions.

**Attribute-based access control.** Attribute-Based Access Control (ABAC, [123]) is a recent model to express access rules in terms of key-value properties of the subject, the resource, the action and the environment. These properties are called *attributes* and include for example the subject identifier, the subject roles, the resource type and the time. We employ ABAC because attributes provide a simple abstraction that enables users to be managed in terms of their properties and because attribute-based access rules can express a wide variety of possibly domain-specific access control concepts.

**Tree-structured policies.** Finally, tree-structured policies or *policy trees* are a means to structure multiple rules into one well-defined policy and reason about possible conflicts between these rules [74, 148]. The rules are the leaves of the tree and decisions of children are combined using combination algorithms such as *FirstApplicable* and *PermitOverrides*. We employ policy trees because they provide an interesting approach to efficiently and correctly combine a large number of rules in a single policy and because they allow to reason about combining the policies of the tenants and the provider.

In practice, these three technologies are supported by the policy language called XACML [5]. As a result, we employ XACML in our prototypes for expressing policies. However, XACML is known for being hard to use. Because this hindered our research, we defined our own simplified attribute-based tree-structured policy language called STAPL [89] and employ this in the latter part of our work. However, as we do not regard STAPL as the focus of this thesis, we only briefly discuss it in the next chapter.

### 1.4.3 Research prototypes

The third and final characteristic of our research approach is that we systematically evaluate our contributions based on prototypes. This approach allowed us to validate and evaluate the behavior of our contributions in practice, in the first place in terms of performance. Moreover, this approach allowed us to gain practical experiences with the employed technologies.

In total, we developed prototypes for each of the major contributions of this work. These prototypes encompass user front-ends, background algorithms and supporting distributed middleware. The source code of each of these prototypes, the resulting performance measurements and optionally a live demo are available on-line. The details of these prototypes are given in each individual chapter, an overview is given in Appendix D.

## 1.5 Contributions

The results of our research can be summarized as four distinct contributions: (i) the Amusa middleware for efficient access control management of multi-tenant SaaS applications, (ii) the concept of federated authorization, (iii) the technique of policy federation and (iv) a technique for scalable and secure concurrent evaluation of history-based access control policies. These four contributions correspond to the four goals described in Section 1.3, with Contributions 2 and 3 both stemming from both Goals 2 and 3.

**Contribution 1: the Amusa middleware for access control in a multi-tenant context.** Our first contribution stems from the goal to address the challenges from multi-tenancy. More precisely, we present the Amusa middleware for access control of multi-tenant SaaS applications. Amusa enables both the provider and all tenants to specify their access rules for the SaaS application, combines these securely and enforces them at run-time with low performance overhead. Moreover, Amusa simplifies the overall access control management using an incremental three-layered approach. Finally, Amusa encapsulates this functionality as reusable middleware.
In this thesis, we elaborate on the three-layered access control management of Amusa and the architecture of the supporting middleware. In addition, we evaluate its performance behavior and integration effort based on a prototype.

**Contribution 2: federated authorization.** As our second contribution, we investigate and validate the concept of *federated authorization.* This second contribution stems from both the goal of lowering the management overhead for tenants as well as from the goal of limiting the disclosure of sensitive access control data and rules. More precisely, federated authorization externalizes policy evaluation from a remote application, dually to federated authentication (see Section 2.4.3). As such, this approach allows to evaluate and enforce an access control policy on a SaaS application without having to disclose this policy nor the data required to evaluate it. Additionally, federated authorization can also facilitate scalable access control management by enabling to centralize the access control management of a tenant for all remote applications that it employs.

In this thesis, we define federated authorization, present a generic middleware architecture for federated authorization, evaluate its performance behavior and validate the potential of this concept in collaborative federated applications based on the case study of the collaborative e-health platform.

**Contribution 3: policy federation.** As our third contribution, we present the technique of *policy federation*. Similarly to the previous contribution, this contribution stems from both the goal of lowering the management overhead for tenants as well as from the goal of limiting the disclosure of sensitive access control data and rules. In addition, this contribution focuses specifically on performance. More precisely, policy federation optimizes the performance of federated authorization by building on the observation that these policies require both data located at the tenant and data located at the provider. Policy federation therefore automatically decomposes and deploys the policies of a tenant to evaluate the resulting parts near the data they require as much as possible while keeping sensitive access control data and policies on the premises of the tenant.

In this thesis, we define the technique of policy federation, present an algorithm for policy federation based on attribute-based tree-structured policies, define supporting middleware and evaluate its performance behavior.

**Contribution 4: scalable and secure concurrent evaluation of access control policies.** Finally, as our fourth contribution, we present an efficient concurrency control scheme specifically for access control. This final contribution stems from the goal to enable policy evaluation to securely scale out with low performance overhead. In this regard, we focus on the concurrency issues when evaluating policies such as history-based policies concurrently or distributedly. More precisely, we present a concurrency control scheme that leverages the specific structure of a policy evaluation so that this scheme can avoid incorrect access control decisions due to concurrency and can scale to a large number of machines with limited latency at the same time.

In this thesis, we model history-based policies using attribute-based tree-structured policies, present our scheme for concurrency control both for a centralized as well as a distributed deployment and demonstrate that this scheme is able to scale to a large number of machines based on our prototype.

## 1.6 Outline

The remainder of this thesis is structured as follows.

Chapter 2 presents the background of this work and thereby discusses access control, policy-based access control, performance tactics for policy-based access control and federated access control.

Chapter 3 presents and evaluates the Amusa middleware for access control in a multi-tenant context. This chapter is based on our publication at the ACM Symposium on Applied Computing 2015 [83].

Chapter 4 investigates the concept of federated authorization and validates its applicability in collaborative applications. This chapter is based on our publications at On The Move 2013 [88] and at HealthInf 2015 [90].

Chapter 5 presents and evaluates the technique of policy federation. This chapter is based on our publications at MW4NG 2012 [84] and in the Journal of Internet Services and Applications (JISA) [85].

Chapter 6 presents and evaluates our domain-specific scheme for concurrency control in policy evaluation. This chapter is based on our publications at MW4NG 2013 [87] and at ACSAC 2015 [86].

Finally, Chapter 7 concludes this thesis and discusses remaining challenges for SaaS access control and policy-based access control in general.

# Chapter 2

# Background

This chapter presents the background of this thesis. In the previous chapter, we already briefly introduced access control and discussed the specific challenges for access control in Software as a Service. In this chapter, we further elaborate on access control in general. Section 2.1 first introduces access control. The following sections then elaborate on the different aspects of access control that this thesis builds upon. More precisely, Section 2.2 discusses common models for access control. Section 2.3 introduces policy-based access control, thereby zooming in on languages for expressing such access control policies and the reference architecture for policy-based access control systems. Section 2.4 describes techniques for federated access control, i.e., access control for applications involving multiple organizations. Section 2.5 discusses existing performance techniques for policy-based access control. Section 2.6 positions the contributions of this thesis in terms of the background presented in this chapter. Finally, Section 2.7 concludes this chapter.

## 2.1 Access control

Access control is the part of security that constrains the *actions* that are performed on the data in a system by enforcing access rules. Access control can be enforced on multiple levels of a system. Of these levels, his thesis focuses on *application-level access control* as opposed to for example network-level access control or database-level access control. In application-level access control, the access rules are expressed in terms of the concepts that are present in the application and the users that employ it. For example, such a rule can state that account managers are only permitted to access documents of the customers for whom they are assigned responsible.

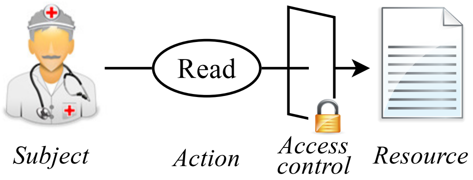


Figure 2.1: Access control constrains which *subjects* can perform which *actions* on which *resources* in a system.

The entities on which actions are performed are called *resources* [5] (sometimes also *objects*). Examples of such resources are files in a file system, patient records in a document management system, a database entry, a database as a whole and a socket. Certain actions can be performed on each of these resources, such as reading a file, adding a diagnose to a patient record, updating a database entry or opening a socket. The entities that perform these actions on these resources are called *subjects* [5]. Examples of such subjects are human users, a remote machine on the internet and a process acting in name of a user. The goal of access control is thus to constrain which subjects can perform which actions on which resources in which circumstances (as illustrated in Figure 2.1).

The process of access control is generally divided into authentication, authorization and audit. *Authentication* is the process of ensuring that the subject is who he, she or it claims to be. This can for example be achieved by verifying the combination of a username and password, the combination of a public and a private cryptographic key, or a biometric property such as a fingerprint. *Authorization* is the process of checking whether an authenticated subject is permitted to perform the requested action on the requested resource by evaluating access rules and blocking the action if not. Finally, *audit* is the process of checking the past actions that authenticated subjects have performed on the resources in the system and correcting any unauthorized actions. In essence, both authorization and audit rely on authentication, but authorization is a form of *a priori* access control while audit is a form of *a posteriori* access control. As a side remark, notice that while authentication is commonly used to prove the identity of a subject, it can also be used to prove a property of the subject (e.g., his or her age) without releasing the identity, an approach that benefits privacy.

Of these three parts, this thesis focuses on authorization. As such, the work in this thesis assumes that the subject has been correctly authenticated, but does not rely on any specific technology or method of authentication. Similar to the bulk of related work, this thesis employs the term “access control” while actually referring to the more specific concept of authorization.

## 2.2 Access control models

Because access control is an important aspect of most applications, researchers soon started to design models in order to be able to reason about this access control. In this section, we discuss some of the most important and relevant access control models for this thesis. Firstly, we discuss the access control matrix, which represents a fundamental overview of the permissions of the subjects in a system. Afterwards, we discuss models that reason about who can assign these permissions and how these permissions can be assigned more effectively. Finally, we discuss the concept of obligations, which extends access control with the possibility to execute operations in conjunction with enforcing the access decision.

### 2.2.1 The basics: the access control matrix

One of the earliest access control models is the *access control matrix*, which was proposed by Lampson [142] in 1971. An access control matrix defines the permissions of the subjects on the resources of the system, i.e., which actions a certain subject can perform on a certain resource, and does so for every subject and every resource in this system. As illustrated in Figure 2.2, this mapping from subjects to resources can be represented as a matrix, hence the name. In essence, the access control matrix provides a fundamental overview of the permissions of the subjects in a system at a specific point in time, even when higher abstractions are used to assign these permissions (see Section 2.2.3).

| | File A | File B | File C |
|---|---|---|---|
| Subject 1 | read | read | read |
| Subject 2 | read,<br>write | | |
| Subject 3 | | read,<br>write | read,<br>write |

Figure 2.2: An example of the access control matrix based on a simple file system with three resources, i.e., the files, and three subjects.

### 2.2.2 Who can assign permissions

The access control matrix represents the permissions of the subjects in the system, but does not define who can assign these permissions.

In general, there are two approaches to this. Firstly, *discretionary access control* (*DAC,* [143]) allows subjects themselves to assign permissions to other subjects for certain resources. For example, in Unix-like file systems, the owner of a file has the ability to specify whether others can read, write or execute it. Contrarily, with *mandatory access control* (*MAC*, [143]) only a central administrator has the ability to assign permissions to subjects. This approach used to typically be discussed in a military context, but is now also incorporated in recent operating systems, e.g., SELinux [25]. Moreover, an organization enforcing access rules on its members is in essence also a form of mandatory access control.

In practice, both models are often combined in order to allow users to discretionarily manage their own resources within the boundaries of mandatory rules. As we will see, this also applies to our work in the sense that we allow the provider of a SaaS application to mandatorily constrain its tenants and each tenant to discretionarily constrain its own users (which again is a form of mandatory access control from the point of view of these users). In addition, more fine-grained models exist to define who can assign permissions based on the administrative operations supported by the models that define *how* permissions are assigned (e.g., ARBAC [183]). We discuss these models next.

### 2.2.3 How permissions are assigned

In addition to *who* can assign permissions, there are also multiple models that define *how* permissions are assigned. In this section, we describe four of the most well-known of such models: identity-based access control, lattice-based access control, role-based access control and attribute-based access control.

#### Identity-based access control

In essence, the access control matrix can be seen as the most basic model to assign permissions to subjects: permissions are assigned to each individual subject for each individual action on each individual resource. This approach can be called *identity-based access control.*

In order to represent the permissions of the subjects in practice, the access control matrix is often implemented and stored in parts. One possible approach are *access control lists.* These contain the permissions of all subjects in a system on a certain resource and are stored with that resource. The dual approach is storing the list of permissions of a certain subject on the resources in the system as *capability lists*.

The approach of identity-based access control is straightforward, but also poses several disadvantages. For example, this approach leads to large management overhead when the number of subjects and resources grows. In addition, this approach lacks protection against untrusted code that runs in name of a subject. To address these challenges, other models provide higher abstractions for assigning permissions, which we discuss next.

### Lattice-based access control

A first well-known access control model that aims to address some of the shortcomings of identity-based access control is *multi-level access control* or *lattice-based access control* [143].

In lattice-based access control, subjects and resources are assigned a security level. These levels are structured in hierarchies, e.g., highly confidential, confidential and public, and multiple such hierarchies are combined into a lattice. To ensure confidentiality, access control then enforces that subjects are not permitted to read resources of higher security levels, nor write resources of lower security levels (the model of Bell and LaPadula [44]). The opposite access rules, i.e., denying reading resources of lower security levels and denying writing resources of higher security levels, ensure integrity (the model of Biba [49]). Lattice-based access control can also protect against untrusted code that runs in name of trusted subjects by assigning the security level of a subject to the processes that he or she starts and also enforcing access control on these processes. This well-known application of multi-level access control is the reason why it originally was and still often is discussed in combination with mandatory access control.

The concept of the security levels in this model can be seen as the foundation of the more general concept of *information flow control [179].* However, in a broader context, lattice-based access control fails to support the access control concepts that are present in the application-level access rules that this work focuses on, such as ownership or roles.

### Role-based access control

In addition to protecting against untrusted code, lattice-based access control in essence also provides more scalable access control management than identity-based access control by assigning labels to subjects and resources instead of assigning individual permissions. However, lattice-based access control does not provide a suitable model

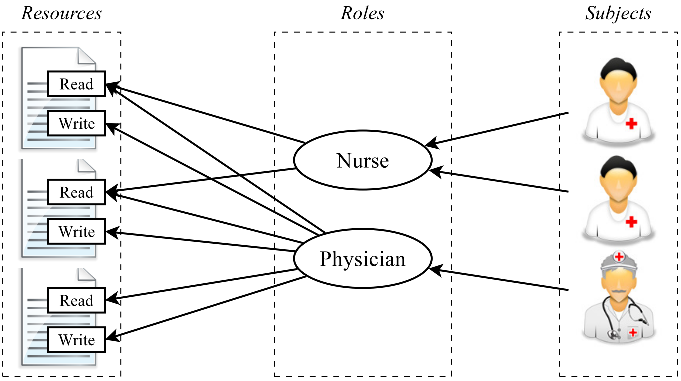


Figure 2.3: An example instantiation of role-based access control [100] with three subjects, two roles and three resources with each two actions. As illustrated, a role bundles multiple permissions and thereby provides more scalable access control management than identity-based access control. This example defines that nurses can only read two specific documents while physicians can read and write any document.

for easily expressing common application-level access rules. As a result, another concept for scalable access control management arose in the 1990s: roles.

Roles form an intermediary between the subjects and their permissions on the different resources (see Figure 2.3): a permission represents a specific action on a specific resource, a role bundles multiple permissions and a subject is assigned one or more such roles. Roles are a natural concept for grouping subjects and in addition, these roles can be modeled to represent the structure of an organization.

Eventually, the concept of roles was ratified into *role-based access control* (*RBAC*) as a NIST standard by Ferraiolo et al. [100] in 2001. In addition to the basic model described in the previous paragraph, this standard also describes how roles can be structured in hierarchies and how roles can be used to enforce separation of duty rules (as first described by Brewer [56]) both statically and dynamically. RBAC is currently employed in practice (e.g., in Spring Security [27]) and has received a lot of attention in research, amongst others with a focus on role mining, formal properties, role administration and practical applications of roles. Fuchs et al. provide a good overview of this field in their survey [109].

However, while RBAC has many advantages compared to previous approaches, it also has the disadvantage that certain rules cannot be easily modeled as roles. For example, expressing that a subject can only access its own resources (an ownership rule) requires one role per user. Similarly, expressing negative rules in which certain subjects are excluded from certain resources requires separate roles for every role minus these subjects. More generally speaking, expressing fine-grained rules in RBAC leads to a large amount of roles (as illustrated by for example Schaad et al. [185]), a problem informally known as *role explosion* (e.g., as used by Jin et al. [128]). In addition, expressing that a subject can only access a resource during a certain time period requires to activate and deactive roles dynamically. As a result of these limitations, a large amount of extensions to RBAC have been proposed, such as roles that incorporate time information [45], location information [46], ownership [102], multi-tenancy [29] or generic parameters [116].

### Attribute-based access control

Following the multiple extensions to RBAC, eventually *attribute-based access control* (*ABAC*) arose. As such, ABAC can be seen as a generalization of these extensions to RBAC as discussed by Sandhu [182].

ABAC (see Figure 2.4) expresses access rules in terms of generic key-value properties

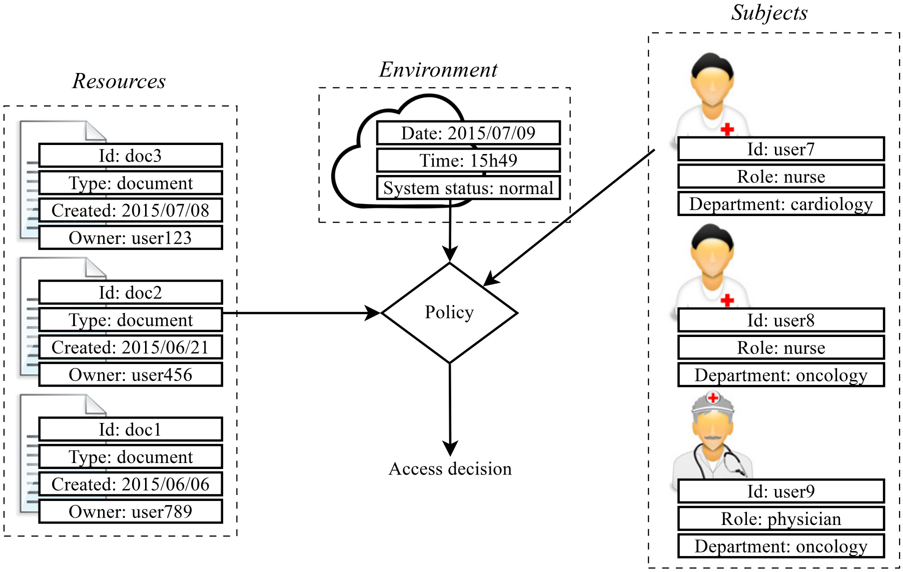


Figure 2.4: An example instantiation of attribute-based access control [100] with three subjects and three resources with each respectively three and four attributes. As illustrated, when user7 attempts to access doc2, the access decision is the result of evaluating the policy with the attributes of this subject, this resource and the environment.

of the subject, the resource, the action and the environment (i.e., the context). These key-value properties are called *attributes*. Examples of subject attributes are his, her or its identifier, location and roles; examples of resources attributes are its type, owner and security label; an example of an action attribute is its identifier, and examples of environment attributes are the time, date and state of the system. This distinction between attributes of the subject, resource, action and environment was ratified in the first XACML standard [117] and was later discussed in research as one of the first by Yuan and Tong [212].

ABAC has a number of interesting properties. Amongst others, attributes provide a simple yet versatile abstraction for managing users and for communicating between multiple organizations [2, 6, 3]. In addition, attributes are an interesting means to express access rules because they can express a wide spectrum of well-known access control concepts such as permissions, roles, ownership, time, separation of duty and location, as well as domain-specific access control concepts such as the assigned customers of a subject or the patients of a physician. Section 2.3.1 shows more elaborate examples of this. For these reasons, this work builds upon ABAC.

Because of its interesting properties, ABAC is gaining attention in both research and industry. In research, Jin et al. [127] have recently shown that ABAC can express the previous models of identity-based discretionary access control, lattice-based access control and role-based access control. Another interesting research track is the transition from RBAC to ABAC. In this context, Kuhn et al. [139] identify three alternatives for integrating attributes and roles: automatically assigning roles based on attributes, modeling roles as any other attribute and restricting the permissions resulting from a subject's roles using attributes. In industry, ABAC is growing into a more general methodology for managing access control for IT applications [123].

However, there are still challenges to be addressed for a successful adoption of ABAC in practice. Sandhu [182] and Hu et al. [123] both provide good overviews of these challenges, but to highlight a few, (i) the increased expressiveness of the model requires additional support for reviewing, testing and monitoring the access rules, (ii) the practical adoption requires qualitative and preferably certifiable attribute sources and (iii) the increased complexity of the access rules requires performance techniques to limit the impact on the application. Although it was not our direct goal, this thesis does aid in the maturation of ABAC by employing and evaluating it in the realistic setting of multi-tenant SaaS applications.

### Further advances

Following up on ABAC, other researchers have introduced more advanced access control models.

For example, one interesting evolution is the notion of *usage control* (*UCON*) as introduced by Park and Sandhu [171, 184]. UCON aims to extend "traditional" access control with features such as mutable attributes, obligations and continuous authorization for long-lived services. In addition, UCON aims to unify access control with trust management, digital rights management and privacy protection. While UCON was introduced over ten years ago, its advanced features still provide interesting techniques for current challenges, e.g., using continuous authorization in access control for cloud services [145] or using mutable attributes to model history-based policies (see Chapter 6). The work of Zhang et al. [213] demonstrates the potential of these techniques by applying them in a usage-based security framework for collaborative computing systems. For a broader overview of the work on UCON, we refer to Lazouski et al. [146].

Another interesting evolution is the concept of *provenance-based access control* as recently introduced by Park et al. [170] as well as by Bates et al. [39]. In this model, provenance data, i.e., data about the origin of other data and the actions taken on it, is used to evaluate access control rules. This data can prove an interesting source of information for access control, amongst others to enforce stateful access rules such as dynamic separation of duty [169].

Finally, we want to highlight the notion of *relationship-based access control* (*ReBAC*). ReBAC expresses access rules in terms of the inter-personal relationships between users. This model was first introduced in the context of social networks, for example in the work of Carrie and Gates [61], Fong [104] and Cheng et al. [66]. Interestingly, ReBAC has recently been generalized into *entity-based access control* (*EBAC*, [115, 78]). This model in turn expresses rules as constraints over subjects, resources, entities in between (e.g., contracts, treatments, teams, organizations etc) and the relationships between all these entities. This approach falls in line with efforts to employ ontologies or semantic web technologies for expressing and evaluating access rules, e.g., the work of Hachem et al. [120] and Giunchiglia et al. [114]. Towards the future, EBAC can be an interesting evolution from ABAC in which even more fine-grained rules can expressed. For example, the common e-health rule "*a physician can only access data of patients whom he treated in the last six months*" is hard to express in ABAC, but can easily be expressed in this model.

### 2.2.4 Beyond permissions: executing operations with an access decision

The previous section described how we can express which subjects can perform which actions on which resources. An interesting extension to this is the concept of *obligations* [177, 117, 171].

Obligations are statements of operations that must be executed by a subject or by the access control system itself in conjunction with enforcing the access decision. For example, such an obligation can specify that the subject should agree to terms and conditions before being granted access. Similarly, an obligation can be used to specify that the system should write a log of the access decision or that it should notify a system administrator that access has been denied to an extremely confidential resource. As a final and important example, an obligation can also specify that an attribute should be updated because of the access decision, e.g., the number of times that a resource has been accessed.

This work takes into account obligations and Chapter 6 builds upon them to model history-based policies. While Park and Sandhu [171] make a distinction between obligations that should be executed before permitting or denying an action, during the action or after the subject has performed it, this work only employs the former.

## 2.3 Policy-based access control

After access rules have been defined in one of the models described in the previous section, these rules have to be expressed in a machine-readable format in order to actually enforce them.

The most straight-forward option is to incorporate these rules in the code of the application. This approach however has several disadvantages. The first disadvantage is that in order to update the access rules, the application code has to be changed and the application has to be recompiled and redeployed every time. Secondly, as a result of the previous, the access rules cannot vary dynamically at run-time either, which is needed for some applications. Thirdly, this approach has the disadvantage that the application developers are occupied with both application logic as well as access control logic. This violates the software engineering principle of *separation of concerns* [172], which can be disadvantageous for security as discussed by De Win et al. [80]. Finally, this approach makes it hard to analyze the complete set of access rules that are enforced on an application.

The approach of *policy-based access control* [190, 181] aims to address these disadvantages by modularizing the access rules with regard to the application code. To achieve this, policy-based access control separates the specification of the access rules from the mechanisms that enforce them by expressing the rules in declarative *access control policies* (or just *policies* in this thesis). Whenever an access decision is needed, these policies are interpreted (or "evaluated") by a specialized component in the application called the *policy evaluation engine* or *policy decision point* [5]. As a result of this approach, the access rules can be specified and updated without having to change the application code and an application developer can focus on application logic while a security expert can focus on the access rules themselves. In addition, this approach enables selecting the appropriate access rules at run-time. This is especially interesting for applications with run-time variability in the access rules, such as the SaaS applications considered in this thesis. Finally, because the access rules are now represented as a separate software artifact, this approach enables the (automated) analysis of the access rules that hold for an application.

The concept of policy-based access control stems from the 1990s. Notable early work in this domain is the work of Sloman [190]. He describes a system and language for policy-based access control that supports both positive and negative authorization and obligation rules. Afterwards, this work evolved into the influential Ponder policy specification language [79] (also see Section 2.3.1). Also notable is the work of Blaze et al. [50] who combine the concept of policy-based access control with security credentials (e.g., X.509 certificates [191]) into the notion of "trust management" for internet applications and provide a practical implementation called PolicyMaker. Finally, we want to highlight the formal work of Schneider [186] who introduces the concept of *security automata* to describe and reason about which rules certain types of policies can effectively enforce.

In a broader sense, policies represent a fundamental software engineering tactic that separates semantics from enforcement and describes these semantics declaratively. As such, policies are being applied for a large variety of goals in the domain of distributed systems in addition to access control. Amongst others, Bacon et al. [36] employ policies for information flow control in multi-domain applications, Wun and Jacobson [211] for managing content-based publish/subscribe middleware, Kumar et al. [140] for describing self-management behavior and Walraven et al. [202] for customizing applications. Moreover, policies also have a long history of being used for network management [121]. Finally, policies also align to other software modularization techniques, such as aspect-oriented software development [101]. As a result of this, policies are faced with similar challenges such as the correct composition with application code as discussed by Lagaisse and Joosen [141]. In addition, policies and aspects can be combined, for example to automatically weave policy enforcement points into application code, as discussed by Verhanneman et al. [201].

As a side remark, notice that the term "access control policy" is overloaded in literature and is used to denote at least two concepts [192]. On the one hand, this term is used to denote the laws and regulations of an organization about security and access control, independent of how they are written down or enforced. On the other hand, this term is used to denote the software artifact in which these regulations are effectively realized so that they can be interpreted by a software system. As we employ policy-based access control mainly as a software engineering tactic for modifiability and separation of concerns, this thesis follows the latter meaning.

As a second side remark, notice that models such as lattice-based access control and role-based access control are sometimes referred to as examples of policy-based access control as well (e.g., in the work of Samarati et al. [181]). Indeed, these models allow to define the permissions of subjects at run-time without having to change the application code and thereby allow to *encode* the access regulations of an organization. However, in these models, there is no software artifact that contains the access rules themselves. In other words, these models allow to configure access control data such as roles, but the rules themselves are hard-coded in the approach, e.g., “permit if the subject has the appropriate permission through activated roles, deny otherwise” for RBAC. In this thesis, we employ the more specific meaning of policy-based access control in which the access rules themselves can be configured and changed at run-time, an approach that provides more flexibility.

In the rest of this section, we elaborate on languages for expressing access control policies and discuss the reference architecture for policy-based access control systems.

## 2.3.1 Policy languages

In order to express the access rules in a policy, a policy language is required. During the last two decades, multiple such languages have been proposed. In this section, we first give an overview of these languages and then zoom in on XACML [5] and STAPL [89]. Appendix A provides an extensive example of a STAPL policy from the case study of electronic document processing (see Section 1.4.1) and illustrates the wide range of access rules that can be expressed using XACML or STAPL.

### Overview

In the early work on policy-based access control, multiple policy languages were proposed for the specific purposes of the authors (e.g., [190, 50]). Following on that, a large number of more generic policy languages was proposed in the beginning of the 2000s, such as the SPL policy language in 1999 by Ribeiro et al. [177], Ponder in 2001 by Damianou et al. [79], EPAL in 2003 by the W3C consortium [35], XACML in 2003 by the OASIS standardization committee [117], Cassandra in 2004 by Becker and Sewell [43] and SecPAL in 2006 by Becker et al. [42] (for an broader overview of such policy languages, we refer to [121]).

All of these languages allow to express access control policies, although EPAL has a specific focus on privacy and SecPAL has a specific focus on trust relationships in distributed systems. All of these languages are also attribute-based and allow to express multiple access control concepts such as identity, ownership roles, and separation of duty, and domain-specific concepts. In addition, Ponder and Cassandra provide specialized features for roles, while the others model roles as any other attribute. Cassandra and SecPAL differ from the others in their foundation on formal logic, i.e., they are implemented based on DataLog. Moreover, as opposed to the others, Cassandra and SecPAL only support positive rules, i.e., they only allow to express which actions are permitted and not which actions are explicitly denied in certain situations. Finally, it is worth noting that although SPL is the earliest language in this overview, it contains advanced features such as typed entities with existential and universal quantifiers that resemble the current evolution towards relationship-based access control (see Section 2.2.3).

Of these languages, this thesis primarily builds upon XACML [5].

### XACML

we focus on the more practical policy languages (as opposed to the more formal languages). More specifically

This thesis builds upon XACML [5]. There are multiple reasons for this. First and foremost, this thesis aims to achieve applied and industry-relevant contributions. In this regard, XACML currently is the de facto standard for expressing access rules in practice and is also widely used in literature (e.g., [157, 92, 152, 160, 176, 158, 33, 69]). Secondly, XACML also provides a number of interesting and state-of-the-art features. For example, XACML is attribute-based and thereby allows a wide spectrum of rules to be expressed, including domain-specific concepts. In addition, XACML allows both positive and negative rules to be expressed, which facilitates more fine-grained rules [177]. Finally, XACML supports obligations, which amongst others are required to express history-based rules (see Chapter 6). A third reason why this thesis builds upon XACML is that it was also supported by a practical and open-source policy evaluation engine [7] when this work started.

In addition to expressing policies using attributes, XACML also structures policies as *policy trees* [53]. In such policy trees, multiple rules are combined into one well-defined policy by having each intermediate node of the tree specify to which requests its children apply by means of a *target.* As such, these policy trees provide a technique to modularly structure and edit a policy.

In XACML, the policy trees consist of three main elements: Rules, Policies and Policy Sets. A Rule specifies an effect (Permit or Deny) and an attribute-based condition for this to hold. A Policy contains one or more Rules and a Policy Set contains one or more Policies or other Policy Sets, which enables policy trees of arbitrary depth. Each of these elements can specify an attribute-based target that specifies to which requests it applies. If this target evaluates to false, the result of evaluating the element is NotApplicable and the children or the condition are not evaluated further. In case this target evaluates to true, the children of a Policy or Policy Set are evaluated depth-first; for a Rule, the condition is evaluated and the result is the specified effect in case it evaluates to true or NotApplicable otherwise. In essence, this behavior makes XACML a three-valued logic. After the result of all its children is known, a Policy or Policy Set combines these results into its own result that bubbles up in the tree. In addition, every Rule, Policy and Policy Set can specify obligations (see Section 2.2.4). These obligations are defined to be fulfilled on a certain result (Permit or Deny) and because the final result is only known at the end of the policy evaluation, these obligations are only fulfilled after the whole policy is evaluated as well.

An important feature of XACML is that the Rules can specify both a Permit and a Deny. In other words, XACML allows to express both positive rules, i.e., rules that permit a certain action, as well as negative rules, i.e., rules that deny certain actions. As a result of this, the XACML policy trees can contain conflicting rules, i.e., a request can be permitted by some rules and denied by others. In order to address such conflicts, the Policies and Policy Sets combine the results of their children by means of specialized *combination algorithms* such as DenyOverrides (which gives precedence to a Deny over a Permit) or FirstApplicable (which returns the result of the first applicable child in the order in which the children were specified). As a result, the policy trees of XACML also enable and force the policy author to reason about possible conflicts between rules in addition to enabling modular policies.

In essence, the concept of a policy tree with the combination algorithms was already present in literature before XACML (e.g., [177, 53, 54]). Afterwards, it received additional attention because of its interesting properties (e.g., [160, 148, 176, 32, 77]). The first version of XACML itself was defined as an OASIS standard in 2003 [117], followed by v2.0 in 2005 [165] and v3.0 in 2013 [5]. This thesis builds upon the core model of XACML, which remains unaltered across these three versions.

### STAPL

While XACML encompasses a powerful model for expressing access control policies, it encodes these policies in XML. While this format in theory improves transportability and interoperability, it also makes XACML policies verbose and hard to read and write for human users.

A possible approach to address this issue is to generate XACML policies from a more human-friendly format. For example, the companies WSO2 and ForgeRock both provide a graphical interface to construct XACML policies as part of their products WSO2 Identity Server [10] and OpenAM [9]. Similarly, the company Axiomatics provides a more human-friendly language to generate XACML called the Axiomatics Language for Authorization (ALFA, [57]). A drawback of this approach however is the abstraction gap between the policies declared in the editor and the policies evaluated at run-time. For example, any run-time errors or logs have to be translated back into the abstraction of the editor, a feature that is not supported yet by these technologies.

Because XACML is hard to use and because of the disadvantages of these existing approaches, we defined and developed an easy-to-use attribute-based tree-structured policy language ourselves. This language is called the Simple Tree-structured Attribute-based Policy Language or STAPL [89]. STAPL is a user-friendly representation of the core concepts of XACML, i.e., policy trees with attribute-based expressions and obligations. As a result, any basic STAPL policy has an equivalent representation in XACML. STAPL is currently defined as a DSL in Scala and is supported by an efficient parser and evaluation engine.

#### Supported access control concepts

Both XACML and STAPL allow to express a wide spectrum of access control concepts (or *policy idioms* [43, 42]). For example, these languages support roles, ownership, time and location, as well as domain-specific concepts such as assigned customers. Appendix A illustrates these concepts based on an extensive policy from the case study of electronic document processing (see Section 1.4.1).

### 2.3.2 The reference architecture for policy-based access control systems

In addition to a policy language, the XACML standard also defines a reference architecture for policy-based access control systems. This architecture was in essence already implicitly present in early work on policy-based access control (e.g., [50, 190]), but was ratified IETF and DMTF [209] and later refined in the XACML standard [117].

This reference architecture is illustrated in Figure 2.5 and consists of the following components:

- The *Application* represents the application logic from the point of view of the access control system.
- The *Policy Decision Point* (*PDP*) is responsible for actually making the access control decision by evaluating the applicable policies loaded from the PAP.
- The *Policy Enforcement Point* (*PEP*) is responsible for requesting an access decision from the PDP when the application requires one. The PEP can be closely integrated with the application (e.g., as an API or using more advanced techniques such as an inline reference monitor [98] or aspect weaving [201]) but can also be an external interceptor such as a firewall.

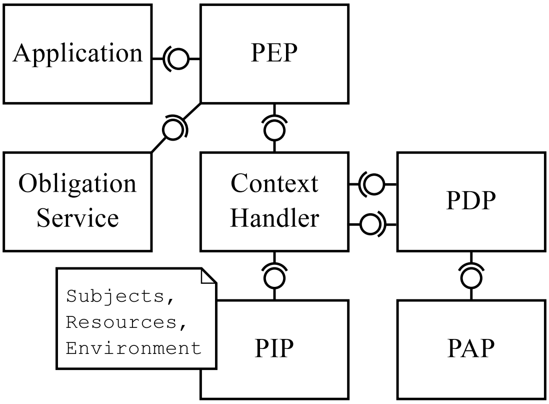


Figure 2.5: The reference architecture for policy-based access control systems [117].

- The *Policy Information Point* (*PIP*) is responsible for providing values for the attributes of the subjects, the resources and the environment. For example, a PIP can be a database, a directory, a web service or even just a piece of software that returns the current date.

- The *Context Handler* is responsible for managing the communication between the PEP, PDP and PIP. More precisely, the context handler converts decision requests, decisions, attribute requests and attribute values from their native formats into the XACML format and vice versa.

- The *Policy Administration Point* (*PAP*) is responsible for managing the policies in the access control system and providing them to the PDP. For example, a PAP can be a file system of policy files, a database, a web service or a combination of these with an administration interface for end-users.

- The *Obligation Service* is responsible for executing any obligations in an access decision. The exact instantiation of an Obligation Service depends on its supported type(s) of obligations, for example a log service for writing logs, an e-mail service for sending e-mails or the same attribute database that is behind a PIP for updating values of attributes.

Notice that this architecture is a *logical* architecture in the sense that it defines the components of a policy-based access control system, but does not define the number of instances of these components in this system or how these are deployed. For example, a large-scale application can employ a policy evaluation engine as PDP that is hosted on a dedicated machine in the network and that employs three attribute databases as PIPs. A firewall on the other hand can be regarded as a PEP, PDP and PAP at the same time and may not even employ a single PIP.

**Resulting authorization protocol.** The authorization protocol that results from the architecture of Figure 2.5 is as follows. When the PEP requires an access decision, it sends a decision request to the PDP through the context handler. This request consists of information about the subject, the resource, the action and the environment in the form of attributes. In order to allow the PDP to request other attributes during the policy evaluation, this request should at least specify the subject that wants to perform the action and the resource on which the subject wants to perform it, both by means of their identifiers. If possible, the PEP can also provide statically known attributes, such as attributes of the subject that are cached in the user session in the application. The context handler converts the native decision request of the PEP into the XACML format and forwards this to the PDP. The PDP then evaluates the policies loaded from the PAP for this request. 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 the PDP can request additional attributes from the PIP through the context handler when needed. Eventually, the PDP reaches an access decision and returns this to the PEP through the context handler. The PEP then fulfills any obligations in this decision using the Obligation Service and if all obligations were fulfilled correctly, it enforces the decision.

## 2.4 Federated access control

In addition to attribute-based access control and policy-based access control, Chapters 4 and 5 of this thesis build upon techniques for federated access control.

Federated access control is concerned with access control for applications in which multiple organizations collaborate. This organizational structure is called a *federation* [94]. More specifically, a federation can be defined as an organizational structure in which multiple organizations have set up collaboration agreements, but do not necessarily trust each other completely and remain separate domains in terms of security and administration. Because multiple organizations collaborate with limited trust in federated applications, access control is of great importance to them.

SaaS applications are a specific and contemporary type of federated applications. In this case, the federation comprises the SaaS provider and its tenants. In addition to SaaS, other types of federations exist as well. For example, a large body of research exists on *grid computing*, which aims to facilitate resource sharing across multiple organizations [106]. More recently, the maturation of web applications has lead to the need for scalable and secure access control management when employing remote applications. As many of these challenges align with the challenges of this thesis, these domains contain research and access control techniques related to this work.

In this section, we give an overview of this research. More precisely, we give an overview of the access control research performed in the domain of grid computing and then describe some of the more recent and standardized technologies that we employ from the domain of web applications. First, we highlight two technologies that these domains have built upon: Kerberos and the Public Key Infrastructure.

### 2.4.1 Early techniques for federated access control: Kerberos and the Public Key Infrastructure

In this section, we highlight two influential examples of early authentication technologies: Kerberos and the Public Key Infrastructure (PKI).

Firstly, Kerberos [168] is a distributed authentication system developed at MIT in the 1980s. Kerberos mainly addresses the problem of strongly authenticating users in a distributed network of servers without this user having to provide his or her password to every server as this lowers usability and facilitates password theft. To do so, Kerberos introduces an authentication server to this network (see Figure 2.6). Every server in the network knows the cryptographic key of the authentication server and vice versa.

The resulting authentication protocol is as follows: When a client, e.g., a user, wants to make a request to a server, it first contacts the authentication server (step 1). The authentication server then authenticates the client locally (steps 2 and 3) and returns a Kerberos ticket (step 4). This ticket contains information about the client, a unique session id and an expiration time. The ticket is encrypted using the cryptographic key of the server so that the client cannot modify it and the session key is encrypted with the key of the authentication server so that the application servers can verify

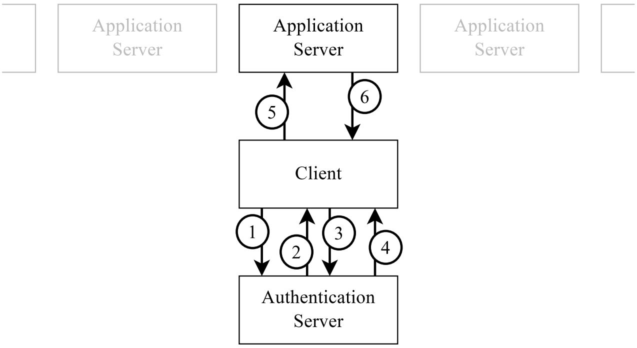


Figure 2.6: The Kerberos protocol for authenticating a client across a network of servers [168].

the validity of the ticket. As such, the client can present this ticket to an application server as proof of its authentication (step 5). This server then checks the validity of this ticket and returns the requested resource (step 6).

In summary, this flow can authenticate users to servers without sharing their passwords while at the same time protecting against eavesdropping and replay attacks. For cross-organizational authentication, this flow can be extended with a ticket granting service that converts a local ticket from the home organization of the user to a ticket that can be used to access the servers in the other organization. Thus, in essence, trust between multiple servers or organizations is established in Kerberos by configuring keys. As we will see later on, this approach and this authentication flow is taken on in more recent technologies for federated authentication.

As originally described by Neuman, Kerberos employs symmetric keys [168]. Amongst others, this has the disadvantage that the key of the authentication server has to be shared with every server in this network, increasing the chances of stealing this key. An alternative approach to symmetric keys is public key cryptography. In this approach, every entity has a private key and an accompanying publicly disclosed key. These two keys are constructed so that when entities encrypt data with their private key, others can then decrypt this data with the public key of this entity. Amongst others, this approach can be used to prove that an entity owns a certain private key without disclosing it, which is a powerful form of authentication. The problem then becomes how to prove that a certain public key belongs to a certain entity.

The Public Key Infrastructure (PKI, [191]) addresses this problem by binding public keys to entities using signed certificates that can be requested from certificate authorities. In turn, the validity of these certificates can be recursively verified based on the public key of these authorities until a trusted root certificate is reached. As such, the PKI can be used to achieve secure and authenticated communication between two parties without requiring a direct trust relationship on beforehand. In addition, the PKI can be used to authenticate entities by itself, i.e., by linking the identity of an entity to that entity using a certificate. This approach can be extended to link any generic attribute to an entity, an approach that forms the basis of many authorization approaches in grid computing.

### 2.4.2 Access control in grid computing

As defined by Foster el al. [106], grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-organizational *virtual organizations*. In other words, multiple organizations set up a collaboration in which they open up their resources such as computing power or research data for direct access by other organizations. This leads to sharing relationships ranging from client-server to peer-to-peer, and from long-lived and static to ad-hoc and highly dynamic.

Because resource providers want to clearly state which resources can be shared with whom under which circumstances, access control is of great importance to grid computing. However, access control in this domain is faced with additional challenges compared to traditional access control, as for example discussed by Foster et al. [105] and Welch et al. [208]. For example, an organization can employ resources or services of multiple other organizations, but having to duplicate the access control management for every such service will incur too much management overhead. As a result, these federated application require techniques for scalable, e.g., centralized, access control management. As another example, both the resource owner, the home organization of a user as well as a the virtual organization want to constrain the actions of this user on these resources. As a result, grids require techniques for securely combining the access control of multiple organizations. As a final example, it is not realistic to assume that all organizations in a virtual organization employ the same technology. As a result, grids require standardized techniques for interoperability.

Kerberos and the PKI have laid the basis for cross-organizational authentication in grid computing. Building on this, the first approach for authorization is the so-called *grid-map file*. This file maps the fully-qualified names of the authorized grid users to accounts in the local operating system. This approach provides simple authorization, but has the disadvantages of requiring all resource providers to update their grid-map file when users enter or leave the virtual organization and only allowing the resource owner to enforce access control.

Following on the grid-map file, multiple improvements have been proposed that have influenced this work. Firstly, PERMIS [64] takes the step from identity-based authorization to attribute-based authorization. More precisely, the resource provider evaluates its policies based on the attributes of the authenticated subject that are retrieved from the PKI. In addition, PERMIS specifies a policy language (that is later superseded by XACML) and an authorization architecture using the components of the reference architecture for policy-based access control systems (see Section 2.3.2). Because of the attribute-based access control and the PKI, PERMIS avoids the need for globally known subject identities and provides more scalable access control management.

A second influential system for grid access control is the Community Authorization Service (CAS, [173]). CAS was developed in parallel to PERMIS and aims to enable a virtual organization to enforce access control on the actions of its members while at the same time achieving scalable access control administration. The authors argue that the solution to these challenges is that a resource owner should just grant access to a virtual organization as a whole and that this virtual organization should restrict which of its users can access which resources in a more fine-grained manner. To achieve this, CAS takes on an authorization flow similar to the authentication flow of Kerberos: when a user wants to access a resource, he or she contacts the CAS Server of the virtual organization. This server authenticates the subject locally and evaluates its policies. If the subject is permitted to access the requested resource, that server returns a capability token that expresses the identity of the virtual organization and the permissions of the subject. The resource owner then verifies this capability token and enforces its policies for this virtual organization as a whole. As such, CAS combines the access control of both parties, but leaves the final decision to the resource owner. Compared to PERMIS, CAS still opts for centralized user management at the CAS server of the virtual organization, but on the other hand allows both the resource owner and the virtual organization to express and enforce their policies.

Apart from PERMIS and CAS, other systems have been proposed. Firstly, VOMS [30] can be seen as an extension to CAS in which the central server can return subject attributes instead of only permissions. Secondly, PRIMA [156] takes a similar approach as PERMIS and additionally focuses on self-management by users and dynamic mapping of subjects to local accounts with the appropriate privileges. Thirdly, Akenti [197] aims to enable multiple stakeholders to express access policies on a resource by storing these as signed certificates in a virtual organization-wide Akenti server. While this could be an interesting generalization of CAS, the authors discuss their proposed policy language, but do not discuss the user management infrastructure or how the different policies are correctly combined. Finally, Cardea [147] focuses on access control from the point of view of the resource provider and opts for XACML policies combined with SAML attribute queries (see later on) instead of the PKI for interoperability. A similar approach is taken later on by Welch et al. [207].

In general, the multiple access control systems for grid computing together have laid the basis for current federated access control techniques. Moreover, these systems demonstrate the advantages and disadvantages of different deployments of the components in the reference architecture, e.g., centralized versus decentralized [189], and of related tactics, e.g., pushing versus pulling attributes. As a result, these approaches have influenced this work, especially our work on federated authorization in Chapter 4.

However, one important difference between grid computing and SaaS applications is the goal of access control. More specifically, the goal of most of the systems discussed in this section is to enable the resource provider to protect *its* resources based on its own access rules. This is similar to the provider wanting to protect its SaaS application as a whole. However, SaaS access control should also enable a tenant to protect its resources located at the provider. This is a fundamentally different trust model that amongst others leads to the need to enable tenants to configure their own users and enforce their own policies on the SaaS application. Because this set-up is similar to the set-up of the resource provider and the virtual organization in CAS, CAS can be seen as the most important influence to our approach for SaaS access control, especially to the work in Chapter 4.

For more information about the systems mentioned in this section, we refer to the multiple surveys of access control techniques for grid computing, such as the one of of Colombo et al. [70] or Jie et al. [126].

### 2.4.3 Federated access control in web applications

Grid computing was an important example of federated applications in the early 2000s. During that period, web applications matured and as larger organizations started to use them, the need for security and scalable access control management in this domain grew as well. As a result, multiple important technologies were developed in the domain of web applications, building on the previous experience in amongst others grid computing. Because SaaS applications often employ web technology in practice, these technologies can provide the basis for the work in this thesis.

In this section, we discuss two of the most important technologies in this domain: federated authentication and OAuth. Note that we do not discuss techniques for federated access control for web services such as WS-Security [144], WS-Policy [37], WS-Trust [167] and WS-Federation [38]. These techniques mainly target service-oriented systems and are not used in SaaS or web applications. Moreover, the technologies that are discussed in this section can be seen as the current instantiation of the core concepts discussed in these standards.

#### Federated authentication

A first important federation technique from the domain of web applications is *federated authentication* [2, 6]. Authentication is the part of access control that confirms the identity of a user, for example by checking the combination of a username and password. Federated authentication externalizes this authentication from an application so that it can be performed by a trusted external party. Amongst others, this approach can be used to centralize authentication and the related user data at the premises of a single organization. For a web application, this approach enables an organization to employ the application without having to duplicate their user data or the management of this data. Additionally, this approach allows the organization to employ their preferred means of authentication and the credentials of the members of the organization are not disclosed to the application.

In order to achieve federated authentication, the application and the organization should be able to communicate. There are currently two major standards for this communication: SAML [2] and OpenID [6]. Of these two, SAML is the most extensive. More precisely, OpenID only supports a fixed set of attributes to be shared between both parties and only supports a single protocol. SAML on the other hand specifies

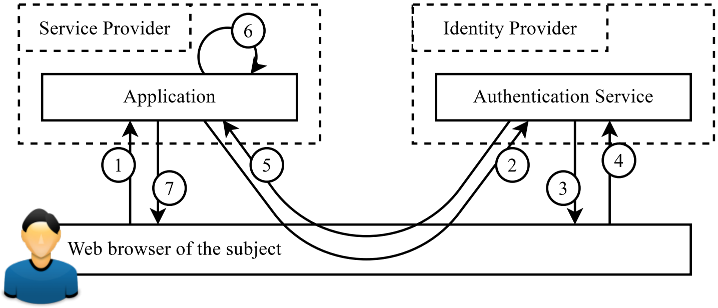


Figure 2.7: A common protocol for federated authentication through the web browser of the user [6, 2].

an extensible XML format for the messages exchanged between the two parties and defines a set of protocols and their bindings to common technologies such as HTTP.

A common example of the protocols supported by SAML (and the one also supported by OpenID) is illustrated in Figure 2.7. In this figure we employ the terminology of SAML, which talks about two parties: the *Service Provider* (*SP*) and the *Identity Provider* (*IdP*). For a web application or a SaaS application, this Identity Provider most commonly is the home organization of the subject or a third party that manages the members of this organization. The protocol between both parties to authenticate a subject is then as follows: The subject makes a request to the application of a certain Service Provider (step 1). In order to authenticate the subject, this application redirects the end-user to the authentication service of his or her Identity Provider accompanied by an authentication request (step 2). Two example approaches to achieve this are an HTTP redirect or returning and automatically submitting an HTML form. The authentication service authenticates the subject locally using any preferred method (steps 3 and 4). If the authentication was successful, the authentication service redirects the end-user to the application accompanied with an (encrypted) authentication statement (step 5). This statement contains at least an identifier for the subject and possibly his or her attributes. The application logs the subject in, for example by setting a cookie (step 6). Finally, the application checks the authorization of the user and returns its response to the subject if permitted (step 7). Notice that if the subject has an active session with the authentication service, steps 3 and 4 can be omitted in order to achieve *Single Sign-On* across the multiple applications that the organization employs.

The approach explained in Figure 2.7 is employed in practice by amongst others Google and Facebook to allow users to log into other applications using their Google and Facebook accounts, e.g., the “Login with Facebook” button. Apart from enabling centralized user management and the other advantages for organizations discussed above, this approach also has the ability to lower password reuse and relieves the service provider of credential management.

Notice that this approach aligns closely with attribute-based access control. More precisely, federated authentication can be used to share the attributes of the subject for use in authorization later on. Moreover, federated authentication can be sued to share the attributes of the subject without sharing its identity, which benefits privacy.

### OAuth

A second important federation technique in the domain of web applications is OAuth [122]. While SAML and OpenID target authentication, OAuth can be regarded as a basic form of federated authorization.

The goal of OAuth is to securely enable third-party applications called *clients* to access a specific part of the resources of a user in a web service without requiring that user to be present. Common examples of these clients are mobile apps or other web applications. Before OAuth, the only option to achieve this was for the user to share his or her credentials for the web service with this client. This approach has the disadvantages of increasing the chance of password theft, of only being able to grant a client access to all of the resources of the user and of only allowing to revoke access for all clients at once. To address these shortcomings, OAuth provides these clients with a unique *authorization token* that the service knows of to which resources of which user it grants access.

The OAuth protocol (illustrated in Figure 2.8) bears some similarity to Kerberos and is as follows: First of all, every client that wants to access a certain web service has to agree on a credential to authenticate itself, e.g., a unique client id obtained from the service provider. When a client then wants to access the resources of a certain user in a certain web service for the first time, e.g., when installing an app or registering for a web application, it requests an authorization grant for these resources from the authorization server of this web service, e.g., by redirecting the user to the authorization server or by presenting a web page in the app (step 1). The authorization server authenticates the user and asks him or her for consent, thereby presenting information about the client and the resources that it wants to access (step 2). If the user consents (step 3), the authorization server generates a unique authorization token, stores for which client, user and resources this holds and returns this token to the client (step 4). When the client wants to effectively access the resources of the user, it presents its client credentials and the obtained authorization token to the authorization server (step 5). The authorization server checks these credentials and the authorization token and if the user has not revoked the consent for this client, the authorization server returns a temporary access token to the client (step 6). The client then requests the resource from the web service accompanied by the received

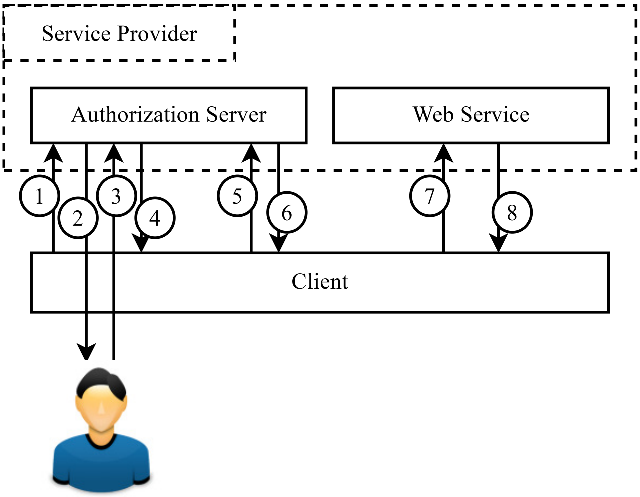


Figure 2.8: The OAuth protocol for granting a client access to the resources of a user in a web service without having to share the credentials of that user [122].

access token (step 7). The web service checks the access token and if valid, returns the resource to the client (step 8). Notice that the access token is valid for only a limited time. After that time, the client has to request a new access token using a similar protocol as before.

OAuth is currently used by amongst others Google, Facebook and Twitter to allow mobile apps and web applications to interact with their services. Compared to the techniques explored in this thesis, the possible access rules that can be enforced however are rather limited because of the limited shared information and the one-time authorization.

### Recent advances: OpenID Connect

Following on OpenID and OAuth, OpenID Connect was recently proposed [180]. OpenID Connect provides similar functionality as OpenID, i.e., obtaining information about the subject in a federated setting, but does so by building on OAuth 2.0. As such, while OpenID could only be used from a web application, OpenID Connect can be used by web applications as well as by mobile or native applications. We do not go deeper into OpenID Connect as we do not build upon it.

## 2.5 Performance of policy-based access control

Access control research is continuously trying to facilitate efficient access control management, amongst others by enabling more fine-grained access rules. However, access control should be enforced on most, if not all requests to an application. Therefore, the latency overhead of access control should be limited to avoid obstructing the intended use of this application. As a result, access control research is faced with the continuous challenge of supporting increasingly fine-grained access rules while incurring only a limited latency overhead. This challenge is further enlarged by the approach of policy-based access control which improves modularity by adding indirection and possibly network communication. Because of the importance of performance, strong and acceptable performance also is a requirement for the contributions of this thesis.

Apart from our work, other authors have also focused on the problem of efficient access control. As a result, this thesis fits into a growing body of work on access control performance. In this section, we give an overview of this work structured by the different employed performance tactics.

### Tactic 1: Caching

A first and well-known tactic to improve the performance of a system is caching. In policy-based access control, there are multiple targets for caching. For example, one can cache the attributes so that less attributes have to be fetched from the database in subsequent policy evaluations. Similarly, one can cache access decisions in order to completely avoid evaluating the policy again for a certain subject, resource or request.

An important observation for this technique is that caching can impede security. More specifically, attributes can change value over time, e.g., because of out-of-band attribute updates such as administrator actions or because of in-band history updates for dynamic separation of duty policies. In this case, caching an attribute can result into using stale information and therefore making incorrect decisions. As a result, cache invalidation and cache consistency are important topics when applying caching to access control, as also discussed by Gheorghe et al. [113].

Surprisingly, little general work exists about caching in access control. Amongst others, Borders et al. [55] describe a general framework for single-node decision caching and spend explicit attention to customizable cache invalidation. Minami and Kotz [163] then focus on caching of formal statements (which can be attributes and decisions in our terminology) in a system for distributed policy evaluation. They employ a publish-subscribe system with periodical freshness messages to decrease the inconsistency windows of the cache.

### Tactic 2: Decision inference

A tactic related to caching that has received more attention in literature is decision inference. Decision inference extends decision caching by trying to infer new decisions from those in the cache without having to contact the PDP. This can be applied to further improve the throughput, latency and availability of the system.

Decision inference was first proposed by Crampton et al. [76] in their Secondary and Approximate Authorization Model (SAAM). In their original paper [76], they apply this approach to lattice-based access control where the lattices of security labels can be used as basis for inferring new decisions. Alternatively, Wei et al. [204] apply this technique to RBAC systems, in which decisions can be inferred based on the role hierarchies. Tripunitara and Carbunar [198] then extend this work with efficient active cache updates using Bloom filters. In addition, Wei et al. [205] also extend SAAM to a distributed system of cooperating decision caches.

### Tactic 3: Pre-evaluating policies

A third possible tactic is to pre-evaluate a policy for a future request and cache the decision so that the actual authorization check for that request later on does not require a policy evaluation any more. As suggested by Beznosov [48], this tactic can also be combined with SAAM for increased effectiveness.

The challenge for this tactic is to know for which requests to pre-evaluate the policy. In this respect, Kohler and Schaad [134] tackle this problem in the context of business processes in which case users often follow multiple steps expressed as workflows. In [132], the authors go deeper into the employed caching strategy and in [133] they compare the performance of this technique with SAAM and traditional caching. Compared to the work of Kohler and Schaad, Kini and Beznosov [131] take a more generally applicable approach by predicting future requests of the user based on its previously observed behavior.

Notice that while these approaches can decrease the latency of policy evaluation from the point of view of the end-user, the PDP will have to process more requests. As a result, the PDP will have to be able to scale out or these tactics should only be applied when the load allows it.

### Tactic 4: Refactoring the policies

Another possible tactic to improve performance is to refactor the policies to a more efficient form as these policies are often not written with performance in mind. Of course, this refactoring should maintain the semantics of the policy, meaning that the refactored policy should provide the same decisions as the original policy.

In this space, Miseldine [164] proposes to statically refactor a policy tree for minimizing the overhead of matching the targets in the tree. Similarly, El Kateb et al. [96] propose to split a policy into multiple smaller policies based on the employed attributes. Finally, Marouf et al. [158, 159] propose to reorder the rules in policies based on run-time statistics of employed attributes in order to reach a decision faster. Complementary to these techniques, Kolovski et al. [135] propose a formalization of XACML using description logic that can, amongst others, be used to prune redundant rules from policies, i.e., rules that are always overridden by another rule in the tree.

### Tactic 5: Optimized policy evaluation engines

Next to optimizing the policies, the internal workings of the policy evaluation engine itself can also be optimized. An influential example of this approach is XEngine [152, 151]. XEngine converts string elements in a policy to equivalent numerical representations for more efficient comparison and flattens a policy tree for more efficient evaluation. This approach however can only be applied to a sub-set of XACML [174]. As an alternative approach, Pina Ros et al. [174] propose to optimize policy evaluation using binary decision diagrams, which poses less constraints on the policies. Complementary, Griffin et al. [118] propose to translate XACML policies into JSON and employ JavaScript features such as non-blocking I/O for more efficient evaluation. Finally, Marouf et al. [158, 159] specifically focus on the problem of efficiently locating matching targets in large sets of policies or rules by clustering these based on the applicable subjects.

### Tactic 6: Efficient attribute fetch

In addition to the previous, a more specific target to optimize a policy evaluation engine is the fetching of attributes. The idea that fetching the required attributes during policy evaluation has a large impact on its latency was posed by Brucker and Petritsch [58]. In their idea paper, they suggest to pre-compute sets of required attributes by statically analyzing the policies in order to fetch multiple attributes at once. Their evaluation suggests that this tactic has the ability to gravely improve policy evaluation time. The observation that attribute fetch has a large impact on policy evaluation time has had a large impact on this work.

### Tactic 7: Distributed policy evaluation

A final tactic to improve the performance of policy-based access control is to distribute policy evaluation. For example, this tactic can be employed to concurrently evaluate a policy on multiple machines for multiple requests, which leads to increased throughput of the system. By then increasing the number of machines and dividing the load over these machines, the capacity of the system can be increased when needed.

In this regard, Chadwick [62] was one of the first to describe a distributed policy decision point. He proposes to achieve coordination by using attribute updates specified in obligations. Alzahrani et al. [31] extend his approach into a decentralized network of policy decision point peers. On the more formal side, Gay et al. [111] discuss *service automata*, which can be regarded as policy decision points that can communicate with each other for coordinated distributed policy evaluation.

A more specific use case of distribution in access control is distributed policy evaluation in the presence of distributed access control data. In this specific case, the goal is to decrease the latency of evaluating a policy by bringing the policy evaluation closer to the data that it requires instead of bringing the data to the evaluation. Our technique of policy federation (see Chapter 5) falls into this category. Before that, Bauer et al. [40] described this approach for access control policies in the form of formal proofs. In their approach, the multiple sources of access control data each evaluate part of the policy locally to reach a final decision more efficiently. In [41] these authors discuss further performance optimizations of their system, such as chaining and delaying expensive parts of the policy. Additionally, Lin et al. [149] sketch a theoretical framework for policy decomposition and distribution for optimized performance when taking into account the location of the attributes required by this policy and their confidentiality constraints. This work has been an important influence to our work on policy federation. A similar approach is discussed by Minami and Kotz [162, 163].

All of the work mentioned in the previous paragraphs focuses on policies that only read data. In case policies do update data however, concurrency issues can arise, which can result into incorrect access decisions. Amongst others, this is the case for history-based policies such a dynamic separation of duty policies. In this regard, Janicke et al. [125] propose a formal model for the possible concurrency issues with policy evaluation. On the more practical side, Dhankhar et al. [92] address these issues using locking, but report on policy evaluation times of seconds. As an alternative approach, Kelbert and Pretschner [130] discuss a decentralized system for policy evaluation and build on the underlying database for concurrency control. Because databases however inherently cannot avoid all inconsistencies due to concurrency and scale to the size of SaaS applications at the same time, we describe an efficient and scalable domain-specific scheme for concurrency control in Chapter 6.

### Comparative performance evaluations

Finally, performance evaluations are required in order to evaluate the impact of each of the performance tactics describe above. While most of the authors of these tactics include performance evaluations for their specific tactics, some authors have extended these into comparative performance evaluations that cover and compare multiple tactics. For example, Komlenovic et al. [136] assess six approaches for RBAC caching in the architecture of Wei et al. [204]. As another example, Turkmen and Crispo [200] compare the performance of three open source XACML evaluation engines in terms of policy loading and policy evaluation time. Finally, Butler et al. [60] build and validate a performance model for policy evaluation engines in order to predict the impact of new performance tactics.

One important issue in these performance evaluations is that the performance of evaluating a policy depends heavily on the structure of this policy, e.g., the number of elements in the policy tree, the number of required attributes and the number of attributes that still have to be fetched from the database. As a result, some authors opt for employing policies used by other authors to achieve a fair comparison (e.g., Komlenovic et al. [136]). In our work, we take a different approach and opt for employing a set of realistic policies that resulted from our case studies. A more sustainable solution for the whole domain of access control however would be the definition of a set of policies to be used in performance benchmarks, as also proposed by Butler et al. [59]. In the past, this task was made difficult by the large number of access control models and paradigms, but currently, such a benchmark seams feasible with the growing adoption of the XACML standard that can express multiple of these previous models.

## 2.6 Positioning of our contributions

The previous sections discussed the background technologies of this thesis. To wrap up this chapter, we now position the contributions of this thesis in terms of these technologies. Figure 2.9 illustrates an overview of this.

**Contribution 1: the Amusa middleware for access control in a multi-tenant context.** Our first contribution is the Amusa access control middleware for multi-tenant SaaS applications. Amusa provides three-layered incremental access control management based on attribute-based policies (see Section 2.2.3). In addition, Amusa securely combines the policies of the provider and all tenants by employing policy trees (see Section 2.3.1). Amusa is novel in the way that it employs these technologies to provide efficient, secure and performant multi-tenant access control management.

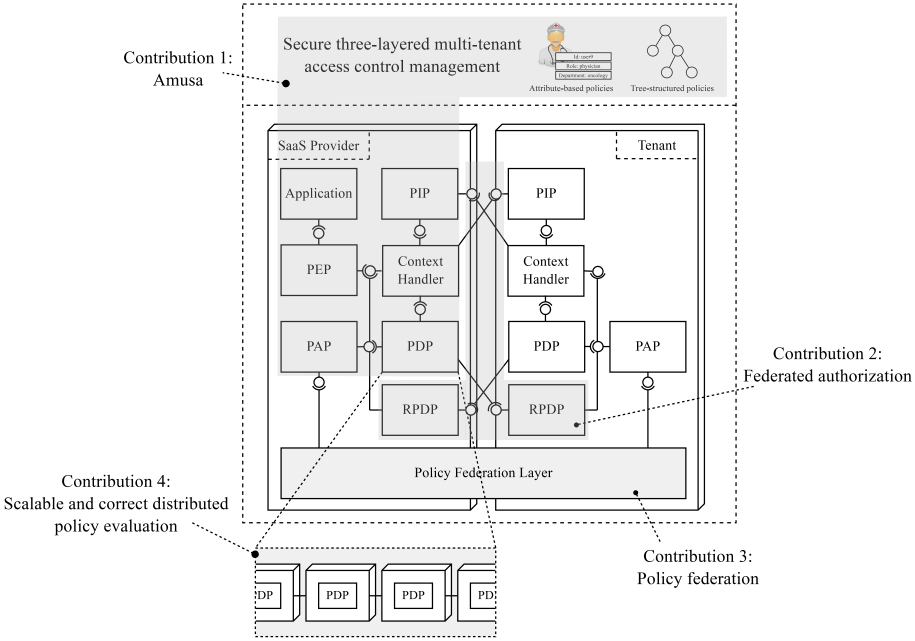


Figure 2.9: Overview of the contributions of this thesis based on the background discussed in this chapter. RPDP stands for Remote Policy Decision Point and is introduced in Chapter 4.

Finally, Amusa also defines a supporting middleware architecture in terms of the reference architecture for policy-based access control systems (see Section 2.3.2) and employs attribute caching and policy deployment across multiple PDPs as configurable performance tactics to lower the performance overhead.

**Contribution 2: the concept of federated authorization.** Our second contribution is the definition and validation of the concept of federated authorization. Complementary to federated authentication (see Section 2.4.3), federated authorization externalizes policy evaluation from a remote application so that it can be performed at the premises of another organization. For SaaS, this technique enables a tenant to enforce access rules on a SaaS application without having to disclose these rules nor the data required to evaluate them. In this thesis, we define a generic policy-based and attribute-based middleware architecture for federated authorization. This architecture employs SAML (see Section 2.4.3) and extends both the reference architecture for policy-based access control systems (see Section 2.3.2) and the XACML policy language (see Section 2.3.1).

**Contribution 3: the technique of policy federation.** Our third contribution is the technique of policy federation. Policy federation automatically decomposes and deploys the tenant policies across tenant and provider to evaluate the resulting parts near the data they require as much as possible while keeping sensitive tenant data and policies at the premises of the tenant. Again, this contribution employs attribute-based tree-structured policies similar to XACML (see Section 2.3.1) to achieve correct results and we define supporting middleware in terms of the reference architecture for policy-based access control systems (see Section 2.3.2). With regard to the performance tactics discussed in Section 2.5, policy federation falls in the categories of refactoring the policies and distributed policy evaluation.

**Contribution 4: scalable and secure concurrent evaluation of history-based access control policies.** Our fourth contribution is an efficient concurrency control scheme specifically for access control. By leveraging the domain-specific behavior of evaluating a tree-structured attribute-based policy (see Section 2.3.1), this scheme is able to avoid incorrect access control decisions due to concurrency while at the same time being able to scale to a large number of multi-core machines with limited latency overhead. As a result, this third contribution enables a scalable distributed PDP that supports history-based policies. As such, this contribution enables a scalable deployment of the PDP component of the reference architecture (see Section 2.3.2) and again builds upon the basic technologies of policy-based access control (see Section 2.3) with attribute-based tree-structured policies (see Section 2.3.1). With regard to the performance tactics discussed in Section 2.5, this concurrency control scheme falls in the category of distributed policy evaluation.

## 2.7 Conclusion

This chapter presented the background of this thesis. This thesis builds on policy-based access control combined with attribute-based tree-structured policies. Moreover, SaaS applications are a type of federated applications and performance is a general concern throughout this thesis. Therefore, this chapter first introduced access control in general and then elaborated on common models for access control, the concept of policy-based access control, supporting policy languages, the reference architecture for policy-based access control systems, techniques for federated access control and performance tactics for policy-based access control. Following on this background, the next chapter presents the first contribution of this thesis, i.e., the Amusa access control middleware for multi-tenant SaaS applications.

# Chapter 3

# Amusa: access control in a multi-tenant context

This chapter presents the first contribution of this thesis: the Amusa middleware for efficient access control management of multi-tenant SaaS applications. Application-level access control is a key component of SaaS security. However, it is also inherently complex because of the multiple parties that want to enforce access rules on the shared SaaS application. In this regard, Amusa facilitates managing SaaS applications by securely enabling the provider and the tenants to enforce their specific access rules expressed in terms of attributes. In addition, Amusa also facilitates building SaaS applications by encapsulating this complex functionality in easy-to-employ middleware with low performance overhead.

This chapter is mainly motivated by the two industrial case studies of eDocs and eWorkforce (see Section 1.4.1) and stems from the goal of addressing the challenges from multi-tenancy. As such, this chapter focuses on the challenge of multi-tenancy and the concerns of performance, low management overhead and low engineering overhead (see Section 1.2). This chapter is based on our publication at the ACM Symposium on Applied Computing 2015 [83].

## 3.1 Introduction

SaaS applications employ application-level multi-tenancy to lower their operational costs (see Chapter 1). In this approach, multiple tenants concurrently employ the same code base, application instances and data store [119]. As a result, tenants cannot be architecturally isolated from each other any more and application-level access control is required to make sure that tenants cannot access each other's resources in the application. Moreover, this access control should also enable the provider to constrain its tenants and should enable the tenants to constrain their own end-users, preferably without requiring interaction with the provider (i.e., self-management).

As a result of these requirements, application-level access control is a highly important part of security for SaaS, but also inherently complex. Providing this functionality is even more a challenge because the access rules of the provider and the tenants vary from case to case. For example, some applications require tenants to be fully isolated while others require business partners to access each others resources. Also, some providers want to constrain tenants based on a pre-paid model, others based on a post-paid model. Similarly, some tenants want to constrain their employees based on their region, others based on contracts, skills or departments. In addition, when all these parties can express their access rules, these rules should also be combined securely. For example, tenants should not be able to specify rules that override the provider's policies or give the tenant access to the resources of other tenants. Finally, the appropriate rules should be dynamically bound and enforced in the shared application at run-time with low performance overhead.

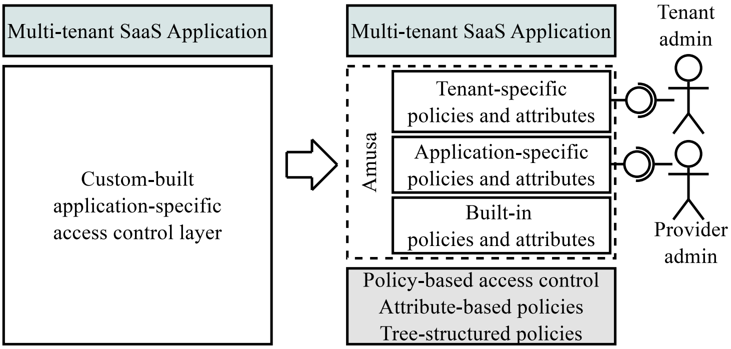

Multi-tenant SaaS Application
Custom-built application-specific access control layer
Multi-tenant SaaS Application
Amusa
Tenant-specific policies and attributes
Application-specific policies and attributes
Built-in policies and attributes
Policy-based access control
Attribute-based policies
Tree-structured policies
Tenant admin
Provider admin

Figure 3.1: The Amusa middleware provides incremental three-layered management of multi-tenant SaaS applications based on policy-based access control with attribute-based tree-structured policies and encapsulates this functionality in reusable middleware.

As a result of this complexity, the state of practice in SaaS application development does not achieve these requirements. Firstly, the provider's policies about tenants are often hard-coded in the application. Secondly, the tenant-specific access rules are often supported using a simple role-based scheme. And thirdly, tenant isolation is either hard-coded in the application or relies on isolation at the level of the data-store such as using namespaced silos in Google App Engine [14]. As a result, the provider policies are hard to customize, tenants are able to express only a limited set of access rules and tenant separation is inflexible.

The state of the art in access control also does not meet these challenges. The combination of policy-based access control [181] with attribute-based [123], tree-structured [74, 148] policies supports the declarative specification of a wide range of rules independent of the application code and binding these at run-time. However, these bare technologies still require each SaaS provider to build a multi-tenant access control layer on top of them. And even when employing these technologies, achieving the requirements stated above is not trivial.

To address these issues, this chapter presents Amusa (Access control middleware for Multi-tenant SaaS Applications, illustrated in Figure 3.1). Amusa allows both the provider and all of its tenants to express their access rules for the SaaS application using expressive attribute-based policies. Amusa combines these policies securely and enforces them at run-time. In addition, Amusa encapsulates this complexity in reusable middleware that requires low development effort of the provider and simplifies the overall access control management using an incremental three-layered approach. Finally, Amusa introduces low performance overhead using efficient policy deployment and attribute fetching, configurable per provider and per tenant. Amusa is motivated by the two industrial case studies of eDocs and eWorkforce (see Section 1.4.1) and builds on a large body of work in access control and policy-based middleware.

The remainder of this chapter is structured as follows. Section 3.2 further elaborates on the case studies and summarizes the requirements of Amusa. Section 3.3 describes Amusa in terms of its employed technologies, its access control management, its supporting middleware architecture and its development API. Section 3.4 evaluates Amusa in terms of security, performance and integration effort. Section 3.5 discusses our experience. Section 3.6 gives an overview of related work and Section 3.7 concludes this chapter.

## 3.2 Problem statement

In this section we elaborate on the challenge of access control for multi-tenant applications based on two case studies of industrial SaaS providers, i.e., eDocs and eWorkforce. We first describe these case studies, then illustrate the challenges for access control and conclude with the resulting requirements.

### 3.2.1 Industrial case studies

This work was performed in collaboration with two industrial SaaS providers in the domains of automated document processing and workforce management, respectively called eDocs and eWorkforce in this thesis.

**eDocs.** eDocs (see Figure 3.2) offers a service to its tenants to efficiently generate and distribute large numbers of personalized documents to their respective users and customers. Typical examples of these tenants are large companies such as banks and press agencies, which distribute pay checks and invoices. Tenants upload the raw data of these documents, e.g., as comma-separated values, after which eDocs generates the actual documents and distributes these to their destinations using e-mail or a print shop. Tenants can group and search their submitted documents and can track the receipt of these documents. On the receiving side, recipients can also employ the eDocs platform to read and manage all their received documents.

**eWorkforce.** eWorkforce (see Figure 3.3) offers a service to automatically plan the workflows for the product and service appointments of its tenants. Typical examples of these appointments are install and repair jobs for tenants such as large telecom operators, utility companies and retailers. The employees of the tenants, e.g.,

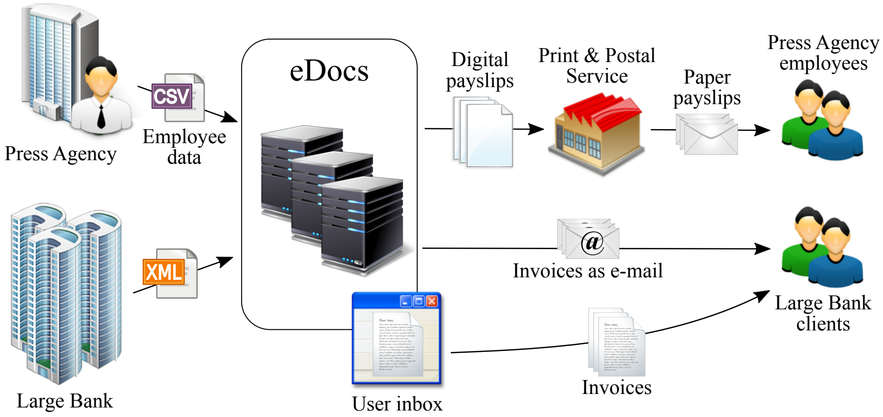


Figure 3.2: A high-level view of the first case study that inspired this chapter: the eDocs service that enables tenants to efficiently generate and distribute large numbers of digital personalized documents to their respective users and customers.

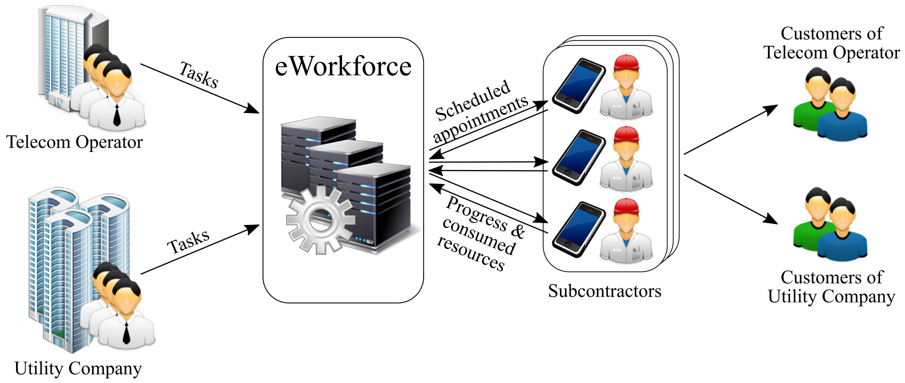


Figure 3.3: A high-level view of the second case study that inspired this chapter: the eWorkforce service that automatically plans the workflows for the product and service appointments of its tenants.

helpdesk operators, insert new tasks into the eWorkforce application. eWorkforce then schedules these optimally, e.g., based on location and estimated time, and assigns the resulting appointments to the appropriate technicians of its subcontractors, which are the companies executing the actual task. The technicians receive these appointments using the mobile eWorkforce application and afterwards report task progress and consumed resources, such as cables and devices. Based on this information, eDocs reschedules appointments where needed and sends updates to the technicians.

Both eDocs and eWorkforce were actively involved in the elicitation of the requirements for the Amusa access control middleware and assisted in its validation. For the interested reader, the detailed description of these two case studies is available in two technical reports [81, 82].

### 3.2.2 Problem illustration

Access control is an important part of security for both eDocs and eWorkforce. This access control should provide three main functionalities:

1. In the first place, such access control should enable the provider to constrain its tenants, e.g., make sure that only paying tenants can access the application.
2. Moreover, most SaaS applications require that the tenants cannot access each other's resources in the shared application (a form of *tenant isolation* [119]).

3. And finally, both eDocs and eWorkforce have large companies as customers and these companies have stringent security requirements themselves. As a result, the tenants should be able to constrain their own users of the SaaS application.

While this functionality by itself is not trivial, it is made even more challenging by the fact that each party wants to apply its own access rules based on its own specific concepts:

1. In terms of provider rules, the access rules differ between eDocs and eWorkforce: eDocs requires the credit of a tenant to be sufficient to access the application, while eWorkforce charges tenants afterwards.
2. In terms of tenant isolation, both eDocs and eWorkforce require that tenants are separated by default, but also require application-specific and tenant-specific exceptions to this tenant isolation. For example, eWorkforce requires that subcontractors are able to access the tasks assigned to their respective subcontractors, eDocs requires that resellers of the service are able to access the documents of their own customers and some tenants of eDocs require that their business partners can access documents of shared projects.
3. In terms of tenant rules, the access rules also differ between tenants. For example, for eDocs, a large bank only permits its users to read documents belonging to their assigned customers and a press agency only permits members of the European region to access the application. For eWorkforce, the tenants constrain their employees based on their skills, interim contracts, projects and internal departments. This variability challenge is related to the known problem of tenant variability in SaaS [47, 119, 195], but is enlarged because the access rules of a tenant depend heavily on its organizational structure and therefore *inherently* vary from tenant to tenant.

In addition, when all these parties can express their access rules, these rules should also be combined securely. For example, tenants should not be able to specify rules that override the provider's policies or give the tenant access to the resources of other tenants. Finally, the appropriate rules should be dynamically bound and enforced in the shared application at run-time with low performance overhead.

As a result of this complexity, both eDocs and eWorkforce and in extension other SaaS providers are all faced with the challenge of setting up a complex access control system.

### 3.2.3 Resulting requirements

The goal of this work is to facilitate building and managing multi-tenant SaaS applications by addressing the challenges illustrated in the previous section. More precisely, the goal of this work is to provide middleware for multi-tenant access control management. This middleware should provide efficient access control management to the SaaS providers and their tenants, should incur low performance overhead and should be reusable by multiple SaaS providers.

As such, the functional requirements for this middleware are five-fold:

1. the middleware should separate the different tenants from each other, but also allow for provider-specific and tenant-specific exceptions,
2. the middleware should enable the provider to constrain its tenants in terms of application-specific concepts,
3. the middleware should enable each tenant to constrain its users in terms of its tenant-specific concepts,
4. the middleware should combine the policies of all involved organizations securely, and
5. the middleware should dynamically enforce the appropriate access rules for each request in the shared application.

In addition, the middleware should adhere to the following three non-functional requirements:

1. the middleware should require little management effort of the provider and the tenants,
2. the middleware should introduce low performance overhead for the SaaS application, and
3. the middleware should require little integration effort of the developers of the application.

## 3.3 The Amusa middleware

To address the requirements stated in the previous section, we present the Amusa middleware. Amusa enables both the provider and all of its tenants to express their access rules on the shared multi-tenant SaaS application, combines these rules securely and enforces them at run-time with low performance overhead. To allow all parties to express their rules in terms of their own concepts, Amusa leverages the expressive model of attribute-based access control. Moreover, in order to simplify the overall access control management of all parties involved, Amusa provides incremental three-layered management in which the provider builds on the attributes and policies defined by Amusa, and the tenants build on the attributes and policies defined by the provider. Finally, Amusa provides a low-effort development API and two configurable performance tactics for lowering its performance overhead.

In the rest of this section, we describe (i) the access control management provided by Amusa, i.e., how administrators use Amusa, (ii) the middleware architecture of Amusa, i.e., how Amusa works under the hood, and (iii) the development API of Amusa, i.e., how developers integrate Amusa in their application code. First, we summarize the technologies on which Amusa builds.

**Key scenario.** For the rest of this chapter, we focus on an illustrative scenario of the eDocs case study employing two tenants: Large Bank and Press Agency. In this scenario, eDocs requires the credit of a tenant to be sufficient to access the application and also relaxes the default tenant isolation policy to permit resellers of its application to view the documents of their respective tenants. In addition, Large Bank only permits its users to read documents belonging to their assigned customers and Press Agency only permits members of the European region to access the SaaS application. Moreover, Large Bank relaxes the tenant isolation policy to permit its business partners to access the documents of shared projects.

## 3.3.1 Enabling technologies

To achieve the requirements of the previous section, Amusa builds on three state-of-the-art access control technologies that each support part of these requirements, i.e. policy-based access control, attribute-based access control and tree-structured policies. We already discussed these technologies in detail in the previous chapter, here we summarize them for clarity:

**Policy-based access control.** Policy-based access control is an approach in which the specification of the access rules is separated from the mechanisms that enforce them. As such, they can be externalized from the application that they constrain and be expressed in modular, declarative *access control policies* [181]. Amusa employs policy-based access control to enable the tenant and provider to specify their own rules without having to change the application code.

**Attribute-based access control.** Attribute-Based Access Control (ABAC, [123]) is a recent model to express access rules in terms of key-value properties of the subject, the resource, the action and the environment. These properties are called *attributes* and include for example the subject identifier, the subject roles, the resource type and the time. Amusa employs ABAC because attributes provide a simple abstraction that enables users to be managed in terms of their properties. Moreover, ABAC provides an expressive policy model that is able to express most of the rules of our case studies such as permissions, roles, ownership, time, separation of duty and location. However, ABAC by itself does not provide efficient access control management, e.g., each attribute for each subject or resource still has to be defined by the appropriate party.

**Tree-structured policies.** Tree-structured policies or *policy trees* are a means to structure multiple rules into one well-defined policy and reason about possible conflicts between these rules (e.g., [74, 148]). To achieve this, the rules are the leaves of the tree and decisions of children are combined using combination algorithms such as *FirstApplicable* and *PermitOverrides*. In addition, every element in the tree defines to which requests it applies by means of a *target*. Amusa employs policy trees to combine the policies of the tenants and the provider while guaranteeing important security properties, e.g., making sure that tenants cannot override the provider policies.

As a result, Amusa is an access control middleware that builds on these technologies, but adds a SaaS-specific layer that enables flexible and secure multi-tenancy. We discuss the access control management provided by Amusa in the next section.

### 3.3.2 Amusa's access control management

In this section, we describe the access control management provided by Amusa, i.e., how administrators manage access control using Amusa. Amusa divides this management over the three parties involved, i.e., the provider, the tenants and the Amusa middleware itself. This three-layered management lowers the overall management effort by means of reuse and gradual extension.

In terms of attribute-based access control, access control management consists of managing the attributes of the subjects and the resources as well as managing the policies that employ these attributes. In this case, Amusa divides the responsibilities as follows: (i) Amusa predefines common attributes and policies that can be reused across applications, providers and tenants. (ii) The provider offers the SaaS application. Therefore he knows the application domain and manages the application resources and actions that should be protected. This results in application-specific attributes and policies. The provider also offers the SaaS application to tenants and therefore also manages the policies that constrain them. (iii) The tenants manage their own users based on organization-specific attributes and policies. Each of the latter parties builds on the previous layer.

In the rest of this section, we describe the three-layered attribute management and three-layered policy management in more detail.

### Three-layered attribute management

Managing attributes entails two kinds of actions: (i) defining possible attributes for subjects and resources and (ii) assigning values to these attributes for these subjects and resources.

**Defining attributes.** With regard to defining attributes, Amusa itself pre-defines a fixed set of attributes, which the provider can extend for its own application and which the tenants in turn can extend for their organization. More precisely, the responsibilities are divided as follows:

*Amusa.* Amusa pre-defines the attributes that it requires for its correct functioning as well as a number of frequently-occurring attributes. The former include the subject and resource identifiers and the associated tenant of a subject or resource, the latter include the owner of a resource and the roles of a subject (albeit in a simplified form compared to concepts defined in the RBAC standard [100], see Appendix A).

*Provider.* The provider defines the attributes of the resources in its application and optionally some frequently-used application-specific subject attributes that can be used by all tenants. For example, in eDocs, each document has a sender and a destination, each user has an e-mail address and each tenant has a credit.

*Tenants.* The tenants define their tenant-specific subject attributes. For example, Large Bank defines attributes for departments, teams and projects, and Press Agency also for geographic regions.

To achieve this functionality, Amusa provides user interfaces to the administrators of the provider and each tenant. More specifically, Amusa requires every attribute to have a name, a type (e.g., a string or a number) and a multiplicity (i.e., single-valued or multi-valued). In addition, Amusa allows to specify whether Amusa should pull an attribute from its database during policy evaluation or whether it can be cached in the user's session (see Section 3.3.3).

The attribute definitions (only their names) for the key scenario are illustrated in Figure 3.4.

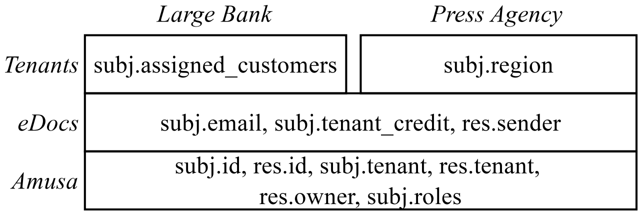

Large Bank
Press Agency
Tenants
subj.assigned_customers
subj.region
eDocs
subj.email, subj.tenant_credit, res.sender
Amusa
subj.id, res.id, subj.tenant, res.tenant, res.owner, subj.roles

Figure 3.4: Amusa enables attributes to be incrementally defined in three layers: Amusa, the provider and the tenants (illustrated for the key scenario).

**Assigning attribute values.** After defining the appropriate attributes, each stakeholder is responsible for specifying their values for the resources and subjects that it controls:

*Amusa.* Amusa automatically assigns attribute values where possible. For example, Amusa automatically assigns the identifier and the tenant of every new subject that it creates.

*Provider.* The provider assigns the appropriate attribute values to its resources. In essence, these values are already present in the SaaS application itself. For example, the application assigns the sender and destination of a document at the moment that it is sent. As such, these attributes are determined by the application code itself and they should be made available to the access control system as attributes (see later on).

*Tenants.* The tenants assign the subject attributes defined by Amusa, the provider or itself to their own subjects. For example, all tenants of eDocs have to specify the roles (defined by Amusa) and e-mail address (assigned by eDocs) of their users. In addition, Large Bank specifies the assigned customers of each of its subjects and Press Agency the region of each of its subjects.

There are multiple ways to assign values to attributes. As mentioned above, the values of the resource attributes are most likely determined by the application code. For the subject attributes however, the tenants should still explicitly provide these values. As the most straightforward approach to achieve this, Amusa provides a user interface for the administrators of the tenants to manually assign values to the attributes of their subjects. As an alternative approach, Amusa can also connect to previously-existing sources of subject data such as directories managed by other user management software.

### Three-layered policy management

The same three-layered model described for attribute management also applies to policy management. In general, the policies of each layer can be specified in terms of attributes that are available in that layer. More specifically, the responsibilities are divided over the three parties as follows:

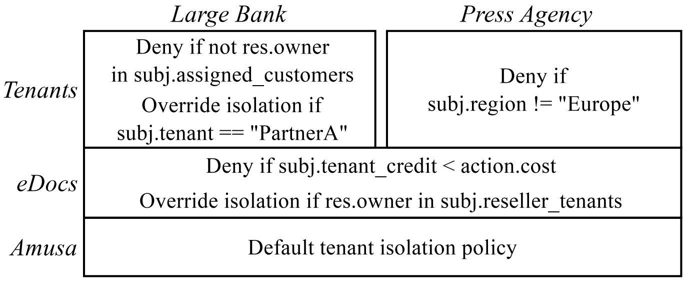
*Amusa.* Amusa itself has some policies built-in that apply to the provider and all tenants. The most important of these is the default tenant isolation policy. These policies can only reason about the attributes pre-defined by Amusa.

*Provider.* The provider can specify policies about the tenants as a whole. These policies can employ the attributes specified by Amusa and by the provider itself. For example, eDocs specifies that users belonging to a certain tenant cannot send any document if the credit of that tenant is not sufficient.

*Tenants.* The tenants can specify policies that apply to their own users. These policies can employ the attributes specified by Amusa, by the provider and by the tenant itself. For example, Press Agency only permits subjects of the European region to access the application.

In addition to these policies, the provider and the tenants can also specify selective exceptions to tenant isolation. For example, this enables eDocs to permit resellers to access the documents of their customers and Large Bank to permit its business partners to access documents of shared projects.

In order to achieve all of this functionality, Amusa provides user interfaces to the administrators of the provider and the tenants. These interfaces allow these users to specify their access rules, currently in the format of XACML or STAPL (see



| | *Large Bank* | *Press Agency* |
|---|---|---|
| *Tenants* | Deny if not res.owner in subj.assigned_customers<br>Override isolation if subj.tenant == "PartnerA" | Deny if subj.region != "Europe" |
| *eDocs* | Deny if subj.tenant_credit < action.cost<br>Override isolation if res.owner in subj.reseller_tenants | |
| *Amusa* | Default tenant isolation policy | |

Figure 3.5: Next to attributes, Amusa also enables policies to be incrementally defined in three layers: Amusa, the provider and the tenants (illustrated for the key scenario).

Section 2.3.1). Each of these users can only view and alter their own access rules, after which these are securely combined by Amusa into a single policy tree (see below).

The resulting policy definitions for the key scenario are illustrated in Figure 3.5. The policies for constraining tenants and users are combined using logical "and" so that the provider and tenants can incrementally restrict access. The isolation exceptions on the other hand are combined using logical "or" so that the provider and the tenants can incrementally expand access.

### Securely combining the policies

Amusa is responsible for combining the policies of all stakeholders in such a way that they are enforced correctly. This means that even though all stakeholders can customize Amusa by defining their own policies, Amusa must still guarantee certain security properties. Most importantly, tenants should not be able to leverage their own policies to override tenant isolation or the rules of the provider.

In order to achieve these properties, Amusa combines the rules of all parties using the policy tree shown in Figure 3.6. The leafs of this tree represent rules that return *Permit* or *Deny* on a certain condition. The intermediate nodes combine the effects of their children using a combination algorithm and specify to which requests they apply using a target. When the provider or a tenant adds or modifies a policy, Amusa constructs this policy tree as follows:

1. Build the sub-tree for tenant isolation:
   - Create a policy with target "*any*" and combination algorithm *PermitOverrides.*
   - Add the default rule for strict tenant isolation.
   - Add the provider exceptions to tenant isolation.
   - For each tenant, add its isolation exceptions wrapped in a policy with target "*resource.owner == <tenant_id>*".

2. Build the complete policy-tree:
   - Create a policy with target "*any*" and combination algorithm *DenyOverrides.*
   - Add the sub-tree for tenant isolation.
   - Add the provider policies about tenants.
   - For each tenant, add its policies about its own users, wrapped in a policy with target "*subject.tenant == <tenant_id>*" and the combination algorithm chosen by the tenant.

This policy tree ensures that the overall access decision is correct because of the employed policy combination algorithms. Firstly, the sub-tree for tenant isolation, the policies of the provider about tenants and the policies of each tenant about their users are combined using *DenyOverrides.* As such, a request is only permitted if both

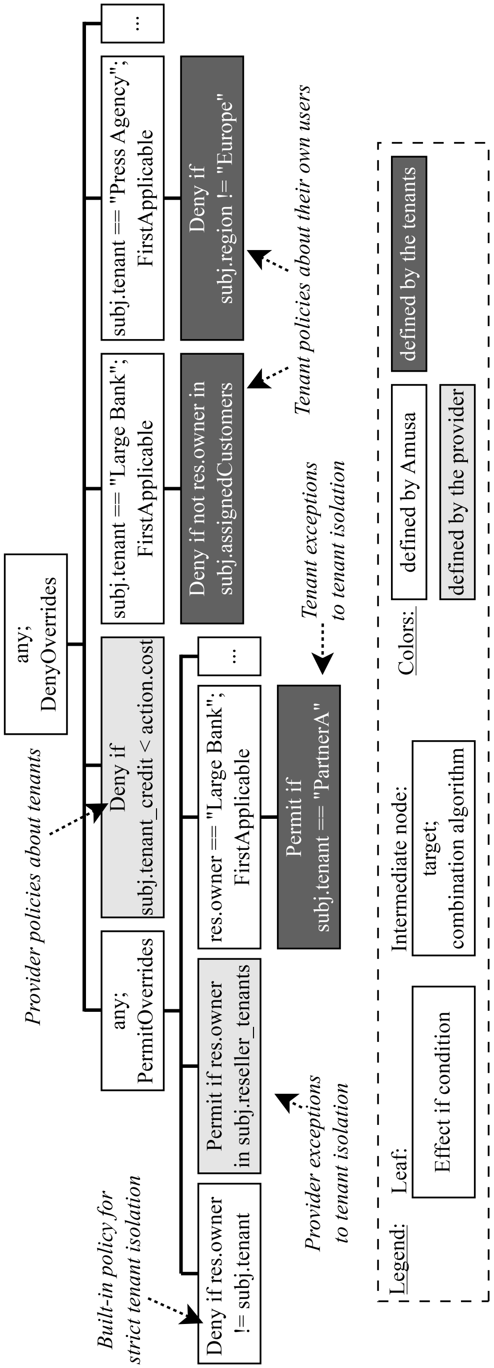


Figure 3.6: The policy tree that securely combines the policies of all stakeholders, illustrated for the key scenario. *subj* is subject, *res* is resource.

tenant isolation, the provider and the tenant permit it. On the other hand, the default tenant isolation rule and the exceptions of the provider and the tenants are combined using *PermitOverrides*. As such, a request is permitted if one exception permits it (and if the other top-level policies permit it). Additionally, the policies of each tenant are inserted in the tree below a target that only applies to the subjects or resources of that tenant and cannot be modified by the tenants. As such, only the policies of the appropriate tenant apply to the employees of that tenant. The security evaluation of Section 3.4.1 validates and illustrates the security properties that follow from this policy tree.

Notice that every element in the policy tree of Figure 3.6 is defined by exactly one stakeholder. This illustrates that three-layered access control management can effectively segregate the different stakeholders.

## 3.3.3 The middleware architecture of Amusa

Following the management provided by Amusa, this section describes the underlying architecture of the Amusa middleware that supports this management. We first discuss the logical decomposition of this architecture, then describe the two configurable performance tactics supported by this architecture, followed by one possible deployment and the resulting control flows for configuration, authentication and authorization.

### Decomposition

Figure 3.7 illustrates the decomposition of the architecture in terms of the reference architecture for policy-based access control systems. Section 2.3.2 explains this reference architecture in-depth, but to summarize for Amusa, the Policy Decision Point (PDP) takes the actual access decision by evaluating the access control policies while the Policy Enforcement Point (PEP) is integrated with the application code (e.g., as an API) and contacts a PDP for an access decision. The Policy Information Points (PIPs) host the attributes of the subjects and the resources. The PIPs are depicted more specifically as databases in Figure 3.7.

As Figure 3.7 illustrates, Amusa provides user interfaces to three types of actors: (i) The application end-users of the tenants and the provider, who use the application that employs the Amusa middleware and who communicate with Amusa itself for authentication. (ii) The administrators of the provider, who employ an administration dashboard to manage their tenants and policies. (iii) The administrators of the tenants, who employ an administration dashboard to manage their users and policies.

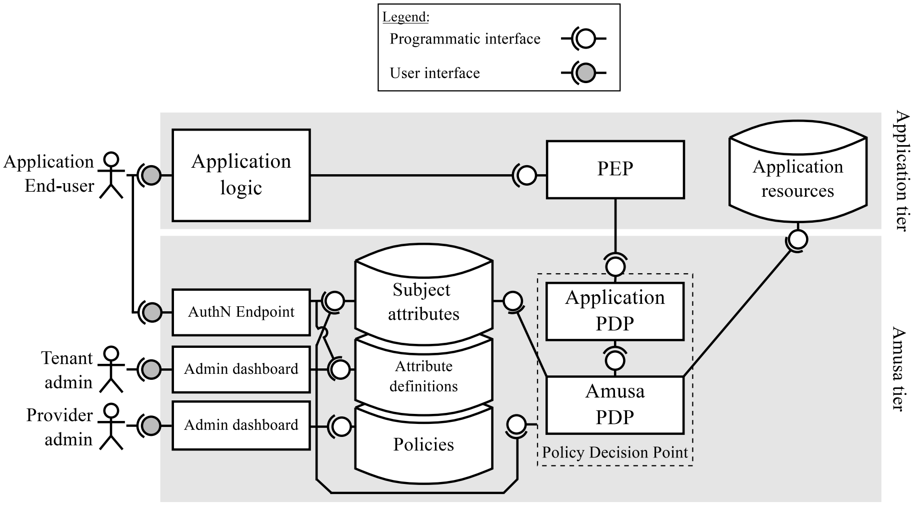


Figure 3.7: The decomposition of the architecture of the Amusa middleware. The policy decision point is the component that evaluates the policies and returns an access decision. AuthN is authentication, PEP is Policy Enforcement Point and PDP is Policy Decision Point.

As Figure 3.7 also illustrates, the architecture of Amusa consists of two tiers: the application tier and the Amusa tier. These two tiers are logically decoupled because of our goal to provide Amusa as reusable middleware.

Firstly, the application tier contains the SaaS application itself. It therefore primarily consists of the application logic and the database that contains the application data, i.e., the resources. Because this tier contains the application logic, it also contains the PEP, which is integrated into the application and forms the interface to the Amusa middleware from the point of view of the application. As a result, the application consists of two components from the point of view of Amusa: the application logic from which an access request is received and the database containing the resource data from which the resource attributes are retrieved.

Secondly, the Amusa tier provides the access control functionality and consists of seven major components: the policy decision point, the authentication endpoint, the provider administrator dashboard, the tenant administrator dashboard, the database containing attribute definitions, the database containing the policies and the database containing the subject attributes. Of these components, the policy decision point evaluates the access control policies and returns access decisions to the PEP. To do so, this component can fetch subject attributes and resource attributes from their respective databases during policy evaluation. As we will explain below, the policy decision point is further decomposed into the Application PDP and the Amusa PDP that can be separately deployed for improved performance. Apart from the policy decision point, the dashboards are used by the provider and tenant administrators to define and assign attributes, and deploy policies. Whenever an administrator makes a change to its policies, the dashboards construct the complete policy tree and deploy it to the policy decision point. Finally, the authentication endpoint provides a user interface for users to authenticate. The application redirects the user to the authentication endpoint provided by Amusa using a federated authentication technique such as SAML [2]. After successful authentication, the authentication endpoint redirects the user back to the application with an authentication statement and his or her subject attributes. The database containing the credentials of the subjects is not shown in Figure 3.7 for readability reasons.

### Configurable performance tactics

Amusa applies two major performance tactics to minimize its run-time performance overhead: (i) pushing or pulling subject attributes and (ii) the flexible deployment of policies across a multi-tier PDP. For these tactics, it is important to notice that their effect depends on the exact policies and attributes in use. Therefore, Amusa allows both tactics to be configured per SaaS application and per tenant. The performance evaluation in Section 3.4.2 evaluates the impact of each of these tactics.

**Pushing or pulling subject attributes.** As a first major configurable performance tactic, Amusa allows to configure for each type of subject attribute whether Amusa should pull this attribute from its respective database during policy evaluation or whether this attribute should be cached in the application and pushed to Amusa with the decision request from the PEP. Pushing an attribute can benefit performance: by caching these attributes in the application, a database request can be avoided during policy evaluation later on. However, caching trades consistency for performance: since the policy decision point will not fetch the latest value of a cached attribute, policy evaluation can be based on stale data. Because access control is a security feature, this inconsistency can lead to security holes, e.g. in case the attributes of a removed account are cached. As such, this tactic is best suited for infrequently-changing attributes such as the department of a subject. Therefore, Amusa allows the domain experts, i.e. the provider and tenant admins, to configure whether a certain subject attribute should be cached or not. Amusa will then return these attributes at successful authentication, after which the application is expected to store them in the user session and provide them to the PEP later on.

**Multi-tier PDP.** As a second major configurable performance tactic, Amusa allows to deploy the complete policy tree across a multi-tier policy decision point: as Figure 3.7 illustrates, the policy decision point is further decomposed into the Amusa PDP and the Application PDP that can be deployed separately. More precisely, the Amusa PDP is meant to be deployed close to the attribute databases, which has the performance advantage of fetching required attributes locally. The Application PDP on the other hand is meant to be deployed close to the application logic such that it can be contacted from the application without network communication. The latter is especially useful for evaluating parts of the policies that only require attributes that can be pushed with the decision request, which applies for example to the default tenant isolation policy, most provider policies, as well as other policies that are high in the policy tree. For this performance tactic, Amusa allows domain experts to manually configure which part of the complete policy tree of Figure 3.6 should be evaluated where.

### Deployment

The components of the decomposition of the Amusa architecture can be deployed in multiple ways. One possible and realistic deployment is shown in Figure 3.8. In this deployment, the application tier and the Amusa tier are also physically decoupled in order to allow both to be scaled out independently. More precisely, the application logic is deployed and replicated on multiple physical nodes in order to support the large amounts of requests per second that most SaaS applications are aimed for. As a result, each resulting application node also contains a PEP and each of these nodes connects to the central Amusa PDP. In Figure 3.8, this PDP is deployed on a dedicated node to achieve low-latency responses to access requests. Should this single node not be able to achieve the required throughput, the Amusa PDP can be scaled out securely using the techniques discussed in Chapter 6.

In addition to the two tiers, the deployment of Figure 3.8 also deploys the Application PDP and Amusa PDP for optimal performance. More precisely, the Amusa PDP is deployed on the same node as the database containing the subject attributes. This offers the performance advantage of fetching subject attributes locally during policy evaluation. The Application PDP on the other hand is deployed on each application node, i.e., close to the application logic. This in turn avoids network overhead in case the policy evaluation does not require any attributes apart from the ones given by the application, e.g. when evaluating the default tenant isolation policy.

Notice that we also deployed the database of application resources separately from the PDPs. This is a realistic assumption because Amusa is meant to be decoupled from the application as much as possible, but this also means that the Amusa PDP possibly still has to fetch some attributes from a remote node because of the physical separation between both tiers. However, the policies in our case studies show that

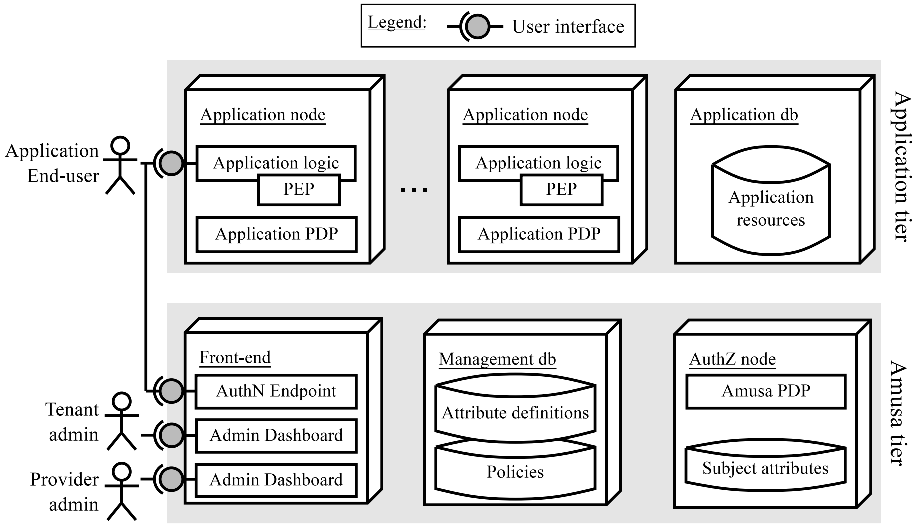


Figure 3.8: A possible and realistic deployment of the Amusa architecture illustrated in Figure 3.7. AuthN is authentication, PEP is Policy Enforcement Point and PDP is Policy Decision Point.

most resource attributes can be given with the decision request from the application, which often already has loaded the resources.

## Resulting control flows

Finally, we here discuss the configuration, authentication and authorization flows that result from the architecture discussed before.

**Configuration flow.** The administrators of the provider and the tenants employ the administrator dashboards in order to define attributes, assign values to subject attributes and specify policies.

Firstly, when an administrator defines a new attribute, this definition is stored in the database of attribute definitions. Afterwards, these attributes can be employed in the access control policies and the appropriate parties can assign values to these attributes for their subjects.

Secondly, when an administrator updates the value of a subject attribute, this value is stored in the database of subject attributes provided by Amusa. The cache of subject attributes in the application is not explicitly updated because these attributes are expected not to be sensitive to inconsistencies as configured by the

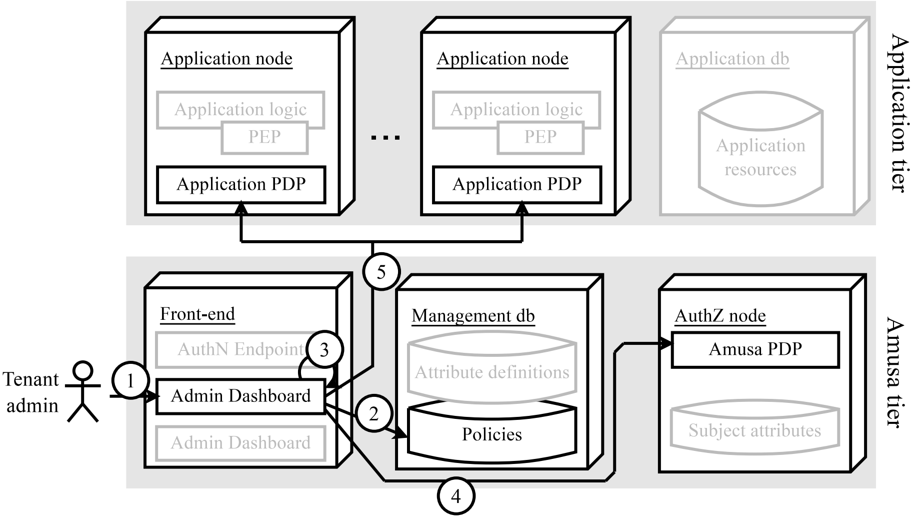


Figure 3.9: The configuration flow when a tenant administrator updates a policy resulting from the deployment of Figure 3.8.

administrator. Notice that it is not possible to specify values for resource attributes in the administrator dashboard. For these attributes, the developers of the provider have to implement attribute handlers to enable the Amusa PDP to fetch newly defined resource attributes from the application database during policy evaluation. This is discussed in the next section.

Finally, Figure 3.9 illustrates the configuration flow for updating a tenant policy. When a tenant administrator updates (a part of) its policy (step 1), the dashboard first stores the updated policy in the policy database (step 2). It then constructs the complete policy tree as explained in Section 3.3.2 (step 3) and deploys this tree across the Amusa PDP (step 4) and Application PDPs (step 5) as configured by the provider administrator. The configuration flow when a provider administrator updates a provider policy is similar.

**Authentication flow.** Figure 3.10 illustrates the authentication flow resulting from the deployment of Figure 3.8. When an unauthenticated user makes a request to the SaaS application (step 1), the application redirects this user to the authentication endpoint provided by Amusa using a federated authentication technique such as SAML [2] (step 2). This endpoint then authenticates the user (steps 3 and 4), e.g., based on the combination of a username and password. After successful authentication, the authentication endpoint fetches which attributes of this subject are configured to be cached in the application from the database of attribute definitions (steps 5 and 6),

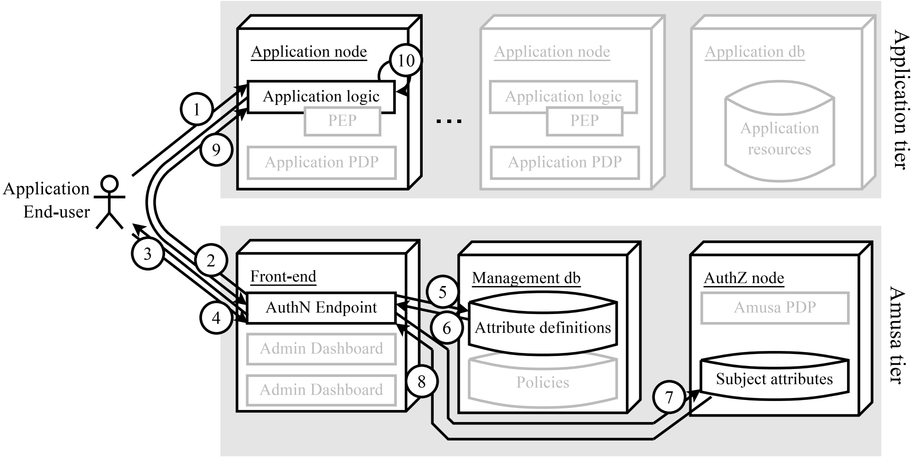


Figure 3.10: The authentication flow resulting from the deployment of Figure 3.8.

fetches the values of these attributes from the database of subject attributes (steps 7 and 8) and redirects the user back to the application with an authentication statement as well as these attributes (step 9). The application then stores these attributes in the session of this user (step 10) and should make these available to the PEP in order to include them in decision requests later on. After having authenticated the user, the application has to authorize the requested action, which is described next.

**Authorization flow.** Figure 3.11 illustrates the authorization flow resulting from the deployment of Figure 3.8. Before this flow starts, the attribute definitions and policies have been loaded into the Amusa PDP and Application PDPs as described before. When an authenticated user then makes a request to the application (step 1), the application (or more precisely, the PEP) sends an authorization request consisting of attributes to the local Application PDP (step 2). This request always contains the identifier of the subject and the resource for which an access decision is required and additionally contains attributes such as already-fetched data about the resource and attributes of the subject cached in the user session in the application. The Application PDP then evaluates its part of the policy tree based on this data (step 3) and contacts the Amusa PDP if its own policies do not lead to a decision (step 4). The Amusa PDP then evaluates its part of the policy tree (step 5), there fetching subject and resource attributes from their respective databases (steps 5a and 5b). Eventually, the Amusa PDP returns its access decision to the Application PDP that requested it (step 6). This Application PDP combines this decision with the rest of the policy tree and returns the final decision to the local PEP (step 7). If permitted, the application then returns the result to the user (step 8).

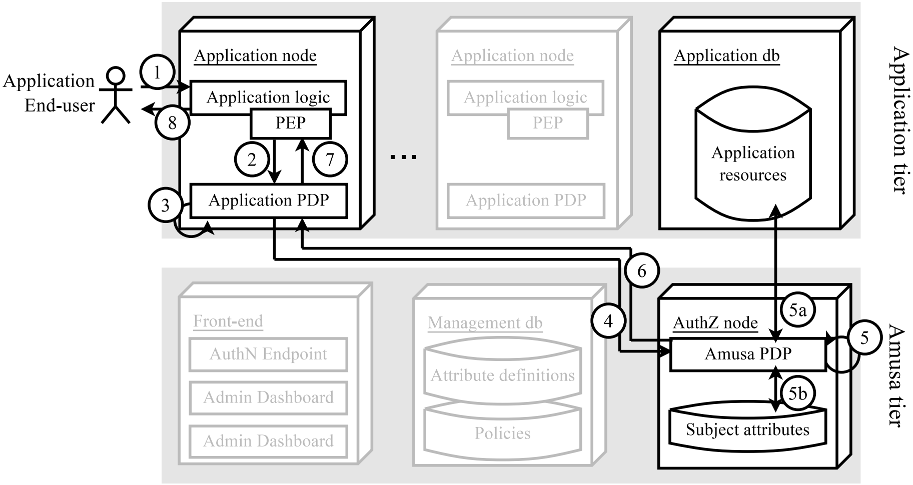


Figure 3.11: The authorization flow resulting from the deployment of Figure 3.8.

Note that in essence, the PDPs evaluate the complete policy tree containing all policies of all stakeholders for every request. However, because of the structure of the complete policy tree (see Figure 3.6), only the appropriate policies apply and are effectively evaluated. As such, this set-up effectively binds the correct policies at run-time, which was one of the explicit requirements for supporting multi-tenant access control.

### 3.3.4 How to integrate Amusa in an application

Finally, we discuss how developers of a SaaS application can integrate Amusa in the code of this application. While provider administrators can define attributes and policies without writing any code, the developers of the provider should still (i) instrument the application code for authentication, (ii) instrument the application code for authorization, i.e. to request access decisions from Amusa and enforce them, and (iii) optionally extend Amusa with attribute handlers for the resource attributes in the application database. Amusa provides a development API for each of these operations.

In this section, we discuss these development APIs. Because we do not regard authentication as a contribution of this work, we focus on the latter two points. Clearly, the effort to perform these modifications should be as low as possible because the goal of Amusa is to facilitate building and managing multi-tenant SaaS applications. Therefore, these APIs are designed to require low integration effort of the provider. In Section 3.4.3, we evaluate this integration effort.

### Authorization API

In terms of authorization, the application should request an authorization decision from Amusa for every action supported by the application. Therefore, the impact of authorization on the application code can be expected to grow with the size of the application. To minimize this impact, the Amusa API is designed to be as concise and easy-to-use as possible.

Listing 3.1 provides an example usage of the basic authorization API. This API closely aligns to the choice for attribute-based access control: the application asks Amusa whether a certain subject is permitted to perform a certain action on certain resource in a certain environment and each of these entities is represented in terms of its attributes. The request should at least contain the identifiers of the subject, the resource and the action in order to fetch other attributes assigned to them, but can also push other attributes for improved performance. The whole policy evaluation process is then handled by Amusa, which returns its decision. This decision is represented as a simple boolean for now, but can be extended towards the future, e.g. with application-level obligations such as requiring the user to agree to terms and conditions.

The basic API of Listing 3.1 can often be further simplified using technology-specific methods. For example, the Java Spring application framework provides functionality

```java
// an application method
public Document viewDoc(Document doc, HttpSession session) {
  Subject s = session.getSubject();
  Resource r = new Resource(doc.getId())
    .addAttribute(”type”, ”document”)
    .addAttribute(”tenant”, doc.getOwner());
  Action a = new Action(”view”);
  if(! pep.isAuthorized(s,r,a)) { return; }
  ... // application logic
}
```

Listing 3.1: Example usage of the basic authorization API in Java.

```java
@PreAuthorize(”#doc”, ”view”)
public Document viewDoc(Document doc, HttpSession session) {
  ... // application logic
}
```

Listing 3.2: Example usage of a simplification of the basic API using Spring-like annotations.

to easily store data in user sessions, automatically inject these data in controller methods and annotate these methods with simple authorization statements [18]. In addition, frameworks such as JEE allow to insert hooks without having to alter the application code. Finally, other authors have discussed methods to automatically insert the calls to the PEP in code, for example based on aspect weaving [201] or based on automated source code analysis [166]. All of these approaches can be used to further lower the instrumentation effort. An example of the resulting simplified API in the case of Spring is shown in Listing 3.2.

### Attribute handlers

As opposed to the subject attributes, the resource attributes are stored outside of Amusa, i.e. in the application databases. Therefore, if the provider wants to allow the Amusa PDP to fetch these attributes during policy evaluation, the provider should implement the appropriate attribute handlers that query these databases and return the resource data in the form of attributes.

Similar to the authorization API, the attribute handlers are designed for simplicity. As shown by Listing 3.3, each attribute handler supports fetching one or multiple attributes as identified by their type, their attribute identifier and the identifier of the resource. By using this simple interface, Amusa can easily be integrated with the application-specific resource databases. In the simplest case, an attribute represents a simple property, e.g. the type of a resource, but in essence, an attribute is an access control abstraction that can also represent a complex database query, e.g. the members of the project to which the resource belongs to.

```
interface AttributeHandler {
  public boolean supports(AttrType t, String attrId);

  public AttributeValue getValue(AttrType t, String attrId,
           String resourceId);
}
```

Listing 3.3: The interface of an attribute handler to be implemented by the provider.

## 3.4 Evaluation

The previous section presented the concept of three-layered access control management for SaaS and the supporting Amusa middleware. Section 3.3.2 already illustrated that this three-layered approach can effectively segregate the different roles in SaaS access control management. This section further evaluates Amusa in terms of (i) its security properties, (ii) its performance impact and (iii) its integration effort.

### 3.4.1 Security

Section 3.3.2 described the policy tree employed by Amusa to securely combine the policies of all the involved parties. In this section, we validate that this policy tree indeed keeps the complete policy secure. First we deduct a number of security properties guaranteed by the policy tree, then we illustrate how these properties mitigate certain misuse cases.

**Security properties.** In essence, the policy tree guarantees the following security properties:

*1. If the provider denies a request, a tenant can never override this.* The policy tree achieves this by combining the policies of all stakeholders using the *DenyOverrides* combination algorithm. As a result, a *Deny* of the provider policies can never by overridden by a tenant policy.

*2. Tenants can only enforce policies on their own users.* The policy tree achieves this by inserting the policies of a certain tenant below a target that only applies to its own *subjects*.

*3. Only the policies of the appropriate tenant are taken into account for a certain request.* The policy tree achieves this as a result of the previous guarantee, combined with the guarantees of Amusa that all tenant identifiers are unique, that the tenant is correctly assigned to subjects and that the assigned tenant cannot be changed by any subject.

*4. Tenants and provider can override the default tenant isolation policy.* The policy tree achieves this by combining the default tenant isolation policy with the isolation exceptions of the provider and the tenants using *PermitOverrides*. As such, the provider and its tenants can all override a *Deny* of the default isolation policy.

*5. Tenants can override tenant isolation only to permit others to view* its *application resources.* The policy tree achieves this by inserting the isolation exceptions of a certain tenant below a target that only applies to its own *resources*.

*6. Tenants cannot gain access to the resources of other tenants using their own policies.* The policy tree achieves this as a result of the previous security guarantees. More precisely, if a user of a tenant A tries to access a resource belonging to tenant B, the constraining policies of tenant A and the isolation exceptions of tenant B apply, respectively because of the subject and the resource of the request. However, unless the isolation exceptions of tenant B permit the request, the *Deny* of the default isolation policy will always overrule a possible *Permit* of tenant A.

Note that some of these guarantees also depend on the correctness of certain attributes. For example, it should not be possible for a user to change the attribute *subj.tenant_credit*. To guarantee this to the provider, the supporting middleware can enforce that certain attributes cannot be defined or assigned by tenants.

**Illustration.** To show that these properties keep the complete policy secure, take the following three examples:

*Example 1.* Imagine that Large Bank tries to gain access to the resources of Press Agency by configuring the following rule *R1: Permit if subject.tenant == "Large Bank" and resource.owner == "Press Agency"*. This is handled by Guarantee 6, keeping the resources of Press Agency secure.

*Example 2.* Imagine that Large Bank tries to perform more actions than its credit permits by configuring the following rule *R2: Permit if subject.tenant_credit == 0*. This is handled by Guarantee 1, which gives preference to the *Deny* of the provider policies.

*Example 3.* Imagine that Large Bank tries to deny the use of the application to Press Agency by configuring the following rule *R3: Deny if subject.tenant == "Press Agency"*. This is handled by Guarantee 2, i.e. *R3* will never apply to the users of Press Agency.

## 3.4.2 Performance

Next to the security properties of Amusa, we also evaluated its performance overhead on a request of a user to the application. More precisely, we evaluated (1) the behavior of Amusa with regard to a growing number of tenants, (2) the performance gain of the configurable performance tactics and (3) the overall resulting overhead of Amusa.

### Set-up

To evaluate the performance overhead of Amusa, we developed a prototype of both the Amusa middleware as well as the eDocs application running on top of this middleware. Both prototypes are written in Java and employ the Spring 3 Web MVC framework for the front-ends. The Amusa prototype employs SAML [2] for authentication, XACML2 [165] for policy specification and an extended version of the SunXACML engine for policy evaluation. The application prototype allows users to send documents to each other, as well as reading and managing these documents. A demo of Amusa, the code of both prototypes and the employed policies are available on-line[1].

The tests deploy the architecture of Figure 3.7 on three nodes, respectively hosting (1) the application logic and resource database, (2) the Amusa PDP and the Amusa attribute database, and (3) the client making the requests to the application. The Application PDP is compiled into the application for optimal performance. Each test was repeated until the confidence interval of the average policy evaluation time was situated within 2% of the sampled mean for a confidence level of 95%. We excluded the top 1% of the results because some of these were up to 100 times larger than the mean, presumably because of running the tests on a shared cloud platform.

The tests employ the policies of the eDocs application and its Large Bank tenant. Measuring the performance overhead of access control is not trivial because this overhead largely depends on the size and structure of the involved policy. In this regard, we opted for measuring the performance of a set of realistic policies instead of a set of artificial policies. The employed policy contains 32 rules, has a tree depth of 4, requires 26 different attributes and comprises 1119 lines of XACML in total. Because reaching an access decision often does not require to evaluate the complete policy, we report the results for 8 representative authorization requests that cover the complete policy. These requests trigger the tenant isolation policy, the provider policies and the different rules of Large Bank. Table 3.1 provides the most important properties these 8 representative requests.

### Impact of a growing number of tenants

With regard to a growing number of tenants, the only aspect of Amusa that grows with the number of tenants is the size of the complete policy tree. More precisely, each new tenant adds a branch to the policy tree of which the applicability will be checked during each policy evaluation. eDocs and eWorkforce both have around 50 tenants. For this size, testing the applicability of all tenant policies in XACML imposes a mean performance overhead of less than 0.15ms. This overhead is negligible as compared to the overall performance overhead, but grows linearly with the number of tenants. As such, it can become beneficial for larger SaaS applications to introduce a specialized policy primitive for more efficient matching of the tenant policies, e.g., a hash-map based on the tenant identifier.

[1] https://distrinet.cs.kuleuven.be/software/amusa/

| **Request:** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|---|
| **#Tree nodes** | 10 | 9 | 23 | 23 | 32 | 22 | 28 | 33 |
| ↪**#Rules** | 1 | 1 | 6 | 7 | 5 | 9 | 11 | 5 |
| **#Attributes** | 6 | 5 | 28 | 16 | 28 | 24 | 27 | 31 |
| ↪ **#Unique** | 5 | 4 | 9 | 9 | 10 | 11 | 10 | 9 |
| ↪ **#Resource** | 2 | 1 | 3 | 5 | 4 | 3 | 4 | 4 |
| ↪ **#Subject** | 3 | 3 | 6 | 4 | 6 | 7 | 5 | 5 |
| ↪**#Pushed** | 3 | 2 | 5 | 3 | 4 | 4 | 3 | 4 |

Table 3.1: Description of the test set-up: run-time properties of 8 representative requests for the employed policy, i.e. the number of nodes evaluated in the policy tree (can result into NotApplicable), the number of evaluated rules (leafs of the policy tree) and the number of attributes required to reach a decision. This total number of required attributes to reach a decision is further decomposed into the number of different required attributes (because the second request for the same attribute will be solved from the cache), the number of different resource attributes, the number of different subject attributes and the number of these that are pushed. These numbers do not take into account identifiers of the subject, the resource or the action.

### Impact of the performance tactics

To measure the impact of the performance tactics, we differentiate between 6 distinct cases based on the possible combinations of the configurable performance tactics. There are two possible configurations for attribute fetching: pushing or pulling certain attributes. There are three possible configurations for evaluating the policy: on the Application PDP, on the Amusa PDP or distributed over both PDPs, i.e., the multi-tier PDP. Notice that for this performance evaluation, the Application PDP can fetch attributes from the attribute databases, which was not shown in the architecture description.

For pushing attributes, we evaluate the impact of pushing the subject attributes that allow this and storing these in the user's session (see Table 3.1). The resource attributes are always given by the application. For the split evaluation, we opted for evaluating the provider policies at the Application PDP and the tenant policies at the Amusa PDP. The latter can be expected to require more subject attributes.

**Results.** Figure 3.12 shows the results of our performance evaluation. In order to provide insights in the behavior of the complete policy tree, we list the results for each individual request of Table 3.1. We can make several interesting observations from these results.

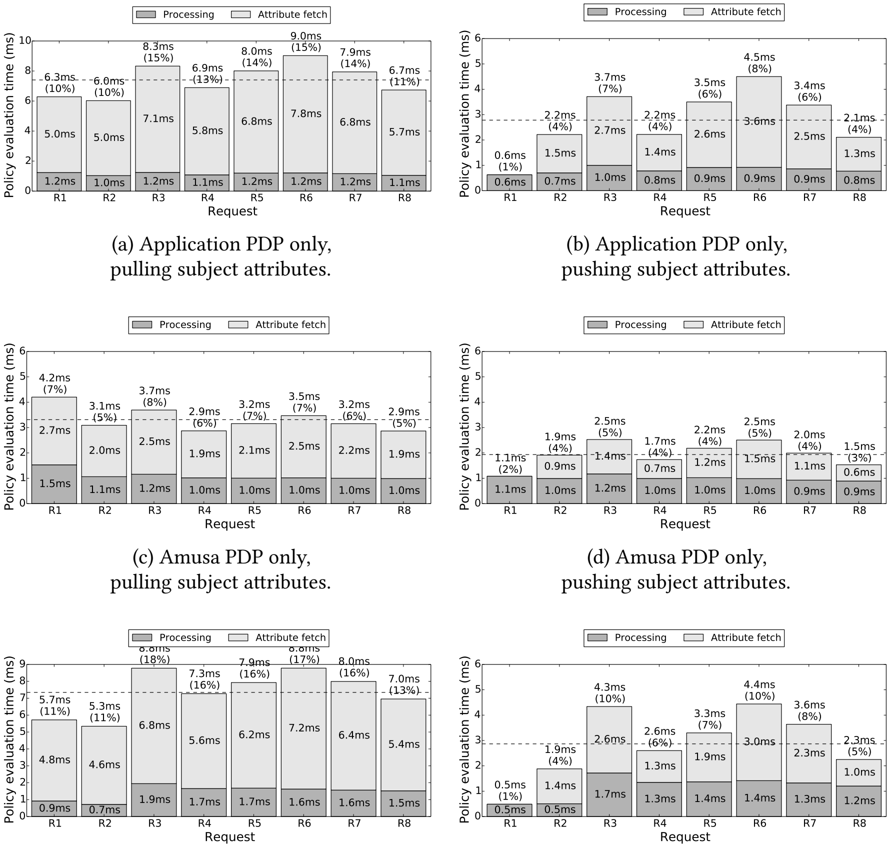
Firstly, the Application PDP that does not employ any pushed attributes (Figure 3.12a) represents the basic behavior. In this case, the largest portion of the policy evaluation time is spent on attribute fetch. This is a result of the large number of required database fetches, which each entail remote communication (an attribute fetch from the Application PDP required 1.2ms on average). This effect is the largest for R6, which requires 7 attribute fetches.

Secondly, the difference between the left column and the right column of Figure 3.12 shows the impact of attribute pushing: every attribute fetch that is avoided saves 1.2ms on average. This effect is most clear for R1, where all attributes can be pushed. R6 still requires 3 attributes to be fetched from the database.

Thirdly, the difference between the first row and the second row of Figure 3.12 shows the performance difference between the Amusa PDP and the Application PDP. The total overhead of attribute fetch is reduced by a linear factor of 2, because the Amusa PDP can fetch attributes from the local database. An attribute fetch from the Amusa PDP required 0.6ms on average. However, evaluating a policy at the Amusa PDP requires network communication, which introduces a fairly constant overhead of 1.25ms. This shows that the net impact of moving policy evaluation from the Application PDP to the Amusa PDP depends on the characteristics of the request and the policy, i.e. whether the number of database attribute fetches justifies the introduced network overhead. Figure 3.12 also shows that pushing attributes also has a positive impact on the Amusa PDP, but that this impact is lower because of the lower latency of an attribute fetch.

Fourthly, the combination of the Application PDP and the Amusa PDP provides the middle ground between both. This is illustrated in the third row of Figure 3.12. For R1 and R2, the combination behaves as intended. Indeed, the part of the policy on the Application PDP does not require communication with the Amusa PDP for reaching a decision, resulting in the performance of only using the Application PDP. However, for the other requests, the results are worse because of the attribute fetches from the Application PDP and the communication with the Amusa PDP. Again, this illustrates the case-specific trade-off between attribute fetch and communication overhead. The choice of splitting the provider and tenant policies was clearly not optimal. Our technique of policy federation (see Chapter 5) can be used to automatically optimize this decomposition.

**Summary.** The results of Figure 3.12 show that both performance tactics can have a positive effect on the policy evaluation time, but that the overall configuration of all tenants and the provider determines the final performance. However, considering



(a) Application PDP only, pulling subject attributes.

(b) Application PDP only, pushing subject attributes.

(c) Amusa PDP only, pulling subject attributes.

(d) Amusa PDP only, pushing subject attributes.

(e) Two-tiered PDP, pulling subject attributes.

(f) Two-tiered PDP, pushing subject attributes.

Figure 3.12: The total policy evaluation time from the point of view of the PEP, for every request of Table 3.1 and for each of the six combinations of the two performance tactics of Section 3.3.3. Each graph shows the portion of the evaluation time spent on network overhead, fetching attributes and processing the policy. Lower is better. The percentage on top of each bar represents the fraction of the complete time to process the application request spent on authorization. The dotted line represents the average over all requests.

the overall results, the average performance overhead of Amusa is low. The best combination of tactics in the test scenario is using only the Application PDP in the multi-tier deployment, combined with the pushing tactic. This leads to a mean performance overhead of 2.8ms for all requests. In a broader perspective, this is *4.9% of the server-side application request time* or *1.5% of the total client-side request time.* These tests can be considered worst-case given that the operations of our application were limited in complexity with respect to practical applications. Even in this context, our performance results can be considered low. Moreover, this overhead can still be lowered using other performance tactics, e.g. high-performant policy engines such as [152], which were not the focus of this work.

### 3.4.3 Integration effort

Finally, we evaluate how much effort it takes for the provider to integrate the Amusa middleware in its SaaS application. Because Amusa is designed to be a reusable middleware, this effort should be low.

As explained in Section 3.3.4, the integration of Amusa in a SaaS application entails three additions to the application: (1) instrumenting the application to employ Amusa for authentication, (2) instrumenting the application to employ Amusa for authorization and (3) extending Amusa to allow it to fetch resource attributes from the application database. Again, we focus on the latter points since we do not regard authentication as a contribution of Amusa and because authentication required only a limited and fixed amount of localized code in the application.

**Authorization.** The application should request an authorization decision from Amusa for every action supported by the application. Therefore, the amount of authorization code in the application can be expected to grow with the application. In the application prototype, authorization was required on 4 application-level methods: creating a document, rendering the view for creating a document, viewing a document and deleting a document. The four resulting access checks only required changes to the MVC controller and required 57 lines of code (6.3% of the complete application code). This limited and localized code illustrates a low instrumentation effort. Moreover, this prototype deliberately evaluated the worst case by employing the basic API of Listing 3.1. As such, the instrumentation effort can further be lowered using more extensive technology-specific simplifications.

**Attribute handlers.** Attribute handlers are required to enable Amusa to dynamically fetch resource attributes from the application database during policy evaluation. However, the policies of the eDocs case study did not require any attribute handlers, i.e. it was possible to push all resource attributes in the decision request from the application. Because these policies were deducted from a realistic case study, this illustrates that attribute handlers are only required for the more complex applications and policies. Moreover, because of the simple interface to which attribute handlers should comply (see Section 3.3.4), they can be expected to require only a low development effort.

## 3.5 Discussion

In this chapter, we presented the Amusa access control middleware for multi-tenant SaaS applications. Amusa enables both the provider and the tenants of a SaaS application to effectively specify their access rules. combines these rules securely and enforces them at run-time. In addition, Amusa requires low development effort of the developers of the SaaS application and introduces a low and configurable performance overhead based on two major performance tactics.

For designing Amusa we have deliberately built on the state-of-the-art technologies of policy-based access control with attribute-based tree-structured policies. As a result, the main contribution of Amusa is the SaaS-specific layer for flexible and secure multi-tenancy on top of these technologies. More specifically, this work applied these in a three-layered approach, which simplifies access control management through gradual refinement. To the best of our knowledge, this work is the most extensive study and application of these technologies described in literature and our experience leads us to believe that all three technologies are important enablers for future access control research.

Our experiences with Amusa have led to the identification of some interesting future work. For example, one important part of this future work is further improving performance. While the performance overhead of Amusa is low, it is still not negligible. As our results show, processing, attribute fetch and distribution all make up a large part of the total policy evaluation time and can serve future optimizations. While the former two are subject of recent research (e.g. respectively [152] and [113]), efficient distributed policy evaluation is still an open challenge, especially when taking into account attribute updates. This challenge particularly applies to SaaS applications that must support a large number of tenants and users and which execute on a large-scale distributed infrastructure. Because of this, we have focused on this problem in Chapter 6. This tactic can also be used to further improve the performance of Amusa.

In addition, our experiences with Amusa have also led to the identification of the current limitations of policy-based access control. Most importantly, while policy-based access control allows the access rules for a SaaS application to be specified independently of the application code and even to be modified at run-time without requiring the application to be stopped, our experience shows that it is currently still challenging and cumbersome to write a correct or complete policy for a certain application.

The most fundamental reason for this is that a policy editor requires information about the application that it wants to constrain, e.g., which types of resources that are present, which actions these support and which attributes these provide. However, in current technologies, this information can only be communicated by means of documentation. As a result, there is currently no way to verify whether a policy covers all actions on all types of resources of the application and a simple typo can lead to an incorrect attribute statement in a policy.

A second cause for the challenge of writing policies are the current policy languages. More precisely, Amusa builds on XACML, which is widely used in literature and is reported to be used in practice as well. However, our experience shows that it is hard for non-expert users to express a XACML policy because of the XML format and the non-trivial policy trees. As explained in Section 2.3.1, there are multiple tactics to address this challenge and we developed the more user-friendly but equally-expressive STAPL language that we have also integrated in Amusa. Towards the future however, more tool support is needed to enable non-expert users to express access rules.

We regard both of these challenges to be fundamental challenges for policy-based access control. Although these challenges are not the main focus of this thesis, the experiences of this work have led us to possible approaches to address them, which we further discuss in Chapter 7.

## 3.6 Related work

The Amusa middleware builds on a large body of work from the domains of multi-tenancy, access control and policy-based middleware.

Firstly, while SaaS is a relatively young paradigm, it has been subject of research for quite some time. For example, in 2007, Guo et al. [119] identified two high-level requirements: isolating tenants and allowing the SaaS application to be customized to the specific needs of each tenant. The latter is also identified by Bezemer et al. [47] and Sun et al. [195]. This work focuses on access control, which is both a means to provide (application-specific) tenant isolation and an important source of variability in SaaS. However, to the best of our knowledge, very little work has been performed to achieve these requirements. In the state of practice, we are not aware of solutions that provide expressive tenant-specific policy-based access control, while our industrial partners stressed the need for such technology. Tenant isolation is mostly implemented manually, and builds on strict data isolation in the data store. More specifically, a built-in tenant identifier is used in database queries. This approach is adopted for example in GAE [14]. However, this does not allow easy application-specific customization of the isolation policy. In the state of the art, Calero et al. [29] also focused on multi-tenant authorization for cloud applications. They opted for extending role-based access control (RBAC, [100]) specifically for multi-tenancy. This work has later been formalized by Tang et al. [196]. Amusa extends this approach by moving from RBAC to more expressive attribute-based policy trees and extending the architecture into a reusable middleware.

Secondly, Amusa was inspired by other access control systems described in literature. Multiple such systems have been described in the domain of grid computing, e.g., CAS, Cardea and PRIMA (for a good overview of this domain, we refer to [70]). Access control in this domain focuses on scalable access control management for a possibly large number of possibly large *virtual organizations*. Especially relevant for this work is CAS [173]. CAS allows resource owners to grant access to a community as a whole and lets the community itself manage fine-grained access control. This approach separates these two roles in access control management similarly to our separation between the provider that manages tenants as a whole and the tenants that manage their internal organization. In addition to this approach, the access control systems for grid computing also employ other techniques similar to the ones used by Amusa, such as the decoupling of enforcement and policy evaluation. Amusa combines these techniques with the recent technologies of ABAC and policy trees into a configurable access control middleware for the domain of SaaS. Apart from access control in grid computing, Fatema et al. [99] and Lazouski et al. [145] more recently also described access control systems relevant to Amusa. Both complement Amusa because they employ similar building blocks, but have a different focus, respectively privacy in multi-organizational systems and usage control in Infrastructure as a Service. Therefore, it would be interesting to see how these systems can be combined with Amusa.

Thirdly, Amusa builds on the experience with policies in the domain of middleware. For example, early work by Sloman [190] already applied policies for declaratively managing access control in distributed systems, which later led to the definition of the influential Ponder specification language for access control policies [79]. Next to access control, policies have been applied for a large variety of goals in the domain of middleware. Amongst others, Bacon et al. [36] employ policies for information flow control in multi-domain applications, Wun and Jacobson [211] for managing content-based publish/subscribe middleware and Kumar et al. [140] for describing self-management behavior. Moreover, policies also have a long history of being used for network management [121]. The common denominator of all this work is that policies are used to separate semantics from enforcement and describe the semantics declaratively. Amusa applied this principle to the domain of SaaS access control.

Finally, we want to highlight the work by Boehm et al. [51]. They also take a case study-based approach for designing an access control system in the domain of SOA. While their solution is rather limited, their requirements and resulting design principles strongly align to ours.

## 3.7 Conclusion

This chapter presented Amusa, an access control middleware for multi-tenant Software-as-a-Service applications. This research was conducted in close collaboration with two industry partners, an approach that resulted in a set of key requirements for access control in SaaS. Amusa in turn offers a management and enforcement architecture that supports these requirements. More precisely, Amusa enables both the provider and the tenants to effectively specify their access rules in terms of their own concepts using three-layered access control management based on attribute-based tree-structured policies. Amusa then combines the rules of all parties securely and enforces them at run-time. Moreover, the evaluation showed that Amusa also requires low development effort and introduces a low and configurable performance overhead. As a result, we believe that Amusa is a key building block for both managing and building multi-tenant SaaS applications.

Following up on Amusa, the next chapters discuss the remaining three contributions of this thesis. Because Amusa addresses the most fundamental challenge for SaaS access control, i.e., the challenge of enabling the provider and all tenants to express their access rules on the multi-tenant SaaS application, these contributions all fit within Amusa: federated authorization can integrate Amusa with the tenant-side authorization systems for scalable access control management, policy federation improves the performance of federated authorization and our final contribution enables the Amusa PDP to securely scale out in order to achieve large amounts of policy evaluations per second. As the first of these contributions, the next chapter shifts focus from the provider of a SaaS application to its tenants and presents the concept of federated authorization.

# Chapter 4

# Federated authorization

This chapter presents our second contribution: the concept of *federated authorization*. More precisely, this chapter focuses on the management overhead and trust issues that arise for the tenants when their access control policies are configured in the SaaS application and are evaluated by the SaaS provider. In this regard, federated authorization externalizes authorization from a SaaS application so that it can be performed at the premises of the tenants. Federated authorization thereby enables to centralize the access management of a tenant and at the same time enables that tenant to enforce a policy on a SaaS application without disclosing this policy nor the access control data that it requires. While there are still hurdles to be addressed for applying federated authorization in practice, our experience leads us to believe that federated authorization is a key building block for access control in future federated applications.

This chapter stems from both the goal of lowering the management overhead for tenants as well as from the goal of limiting the disclosure of sensitive tenant access control data and rules. As such, this chapter focuses on the challenges from outsourcing and the concern of low management overhead (see Section 1.2). This chapter is mainly motivated by the case studies of home patient monitoring and the collaborative care platform (see Section 1.4.1) and is based on our publications at On The Move 2013 [88] and at HealthInf 2015 [90].

## 4.1 Introduction

As explained in Chapter 1, SaaS applications should enable their tenants to control the access of their own end-users, e.g., their employees, to their data in the application, preferably by means of self-service. Most SaaS applications achieve this by allowing their tenants to configure users and their permissions using an application-specific dashboard. The Amusa middleware introduced in the previous chapter takes a similar approach. As a result of this approach, the access rules of the tenants are configured in the SaaS application itself and are evaluated by the SaaS provider. This leads to two problems:

**Problem 1: management overhead.** Firstly, large organizations such as hospitals typically employ on-premise access control systems for centrally and efficiently managing their users across their on-premise applications. However, while techniques for federated authentication enable remote applications to integrate with these systems in terms of user management, no similar techniques exist for authorization. As a result, SaaS applications currently only enable their tenants to configure their access rules in a dashboard of the SaaS application itself. In turn, the organization-wide access control management of these tenants is scattered and duplicated across the multiple SaaS applications that it uses, which makes it difficult for a tenant to obtain an overview of its complete access control management and leads to large administrative overhead. While this overhead may still be feasible for small organizations, it quickly becomes unfeasible for larger organizations such as hospitals, eventually leading to inconsistencies and security holes.

**Problem 2: necessary disclosure of sensitive tenant access control data.** Secondly, because the access rules of the tenants are evaluated by the SaaS provider, the tenants are forced to disclose their access rules and the access control data required to evaluate these to the SaaS provider. These rules however often require sensitive user data that are stored in the on-premise systems of the tenant, such as the details of a patient or a physician in the domain of e-health. Although the tenant trusts the provider with the data in the SaaS application itself, it does not necessarily want to trust the provider with sensitive data needed for access control, as also discussed by other authors, e.g., [210, 150, 32]. This is especially true in privacy-sensitive domains such as e-health. Moreover, regulatory requirements such as HIPAA [1] or the European DPD [71] even forbid the tenant to share this data.

These two issues present themselves both for state-of-practice SaaS applications as well as for state-of-the-art policy-based approaches such as the Amusa middleware. However, as explained in more detail in the previous chapter, state-of-practice SaaS applications commonly only support a limited access control model, such as a fixed set of roles to assign to users. As a result, the access rules of an organization are encoded in a limited access control model, which can be considered as less sensitive than the more elaborate form of these rules in an access control policy. Similarly, as the rules are simplified in this encoding, the access control data that has to be shared with these applications in order to evaluate them is more limited as well. As such, the problem of having to disclose sensitive access control data can be seen as relevant to all SaaS applications, but exacerbated when these applications make the shift towards policy-based access control. Therefore, this chapter assumes the point of view of state-of-the-art SaaS applications that employ policy-based access control. Notice that the problem of scalable management on the other hand is equally large for state-of-practice and state-of-the-art SaaS applications.

To address the two problems described above, this chapter investigates the technique of *federated authorization*. In analogy to federated authentication (see Section 2.4.3), federated authorization externalizes authorization from a remote application in order to perform it at the premises of another organization. For policy-based access control, this authorization comes down to policy evaluation. As such, federated authorization can be applied in the context of SaaS to evaluate the policies of a tenant at the premises of that tenant. When combined with federated authentication, this technique thereby enables centralizing the entire access control management of that tenant and at the same time keeps its sensitive access control policies and access control data confidential.

Apart from this work, other authors have discussed federated authorization as well (e.g., [28, 33, 150]). Moreover, domain-specific instances of federated authorization are already in use in practice (e.g., 3-D Secure for internet payments [4]). In addition, the recent web technology OAuth (see Section 2.4.3) can be seen as a basic form of federated authorization. Compared to these approaches, this work focuses on a generic attribute-based and policy-based access control middleware for federated authorization and the evaluation of the performance behavior of this middleware. In addition, we first position federated authorization in the context of SaaS and afterwards generalize and validate this technique outside the scope of SaaS, i.e., in the domain of collaborative applications involving more than two parties. This validation shows that federated authorization can serve as an enabler for addressing multiple challenges in this domain, leading us to believe that this technique is a necessary building block for future federated access control in general.

The remainder of this chapter is structured as follows. Section 4.2 demonstrates the need for federated authorization based on the case study of an e-health patient monitoring application provided to hospitals. Section 4.3 presents our generic attribute-based and policy-based access control middleware for federated authorization in the context of SaaS, thereby discussing the architecture of this middleware and the required extensions to the policy language XACML. Section 4.4 evaluates the performance behavior of this middleware based on a prototype.

Section 4.5 further discusses our experiences and Section 4.6 generalizes federated authorization outside the scope of SaaS by validating it in the case study of an e-health collaboration platform. Section 4.7 provides an outlook towards the future of federated authorization and Section 4.8 concludes this chapter.

## 4.2 Motivation and problem illustration

In this section, we motivate this work based on the case study of a patient monitoring application in the domain of e-health. We first discuss this case study, then list the resulting requirements for SaaS access control from the point of view of the tenants and finally show the need for federated authorization.

### 4.2.1 Case study: a patient monitoring service

While the previous chapter was mainly motivated by the case studies of eDocs and eWorkforce, this chapter is mainly motivated by a SaaS application in the domain of e-health, i.e., an application for monitoring patients of cardiovascular diseases at their homes, provided to hospitals as a service. This case study is based on a number of research projects [11, 12]. Similar to eDocs and eWorkforce, this case study is a representative state-of-the-art SaaS application that performs core business activities and is used by large organizations.

The patient monitoring service (illustrated in Figure 4.1) allows patients of cardiovascular diseases to be monitored continuously after leaving the hospital by wearing

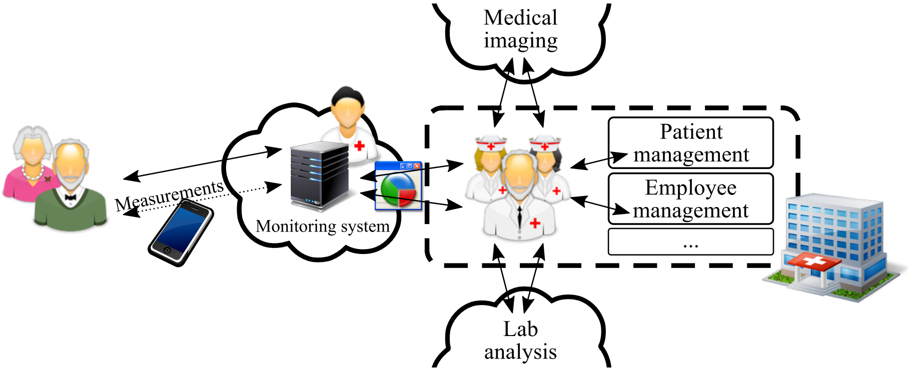


Figure 4.1: The case study that inspired this work: a system for monitoring patients at their homes, offered to hospitals as a service.

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 SaaS provider. More precisely, telemedicine operators employed by this provider continuously check upon the patients. For this, the application offers an overview of a 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 automatically. These physicians can then assess the situation by means of an overview of the patient's status similar to that of the telemedicine operators and can also check upon the status of a patient proactively. 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 application provides functionality such as patient questionnaires and shared notes on a patient overview.

In this case study, the hospitals are the tenants of the SaaS application and the patients, physicians, nurses and telemedicine operators are its end-users. Next to the monitoring application, the hospitals employ other on-premise applications, e.g., for patient management, and other SaaS applications, e.g., for medical imaging or lab analyses.

### 4.2.2 Resulting access control requirements

Because of the nature of the data in the patient monitoring application and the heavy regulation of the domain of e-health, security and in particular access control are of high importance to this application. As explained in Chapter 1, both the provider and the tenants should be able to constrain access to a SaaS application. This is also the case for this patient monitoring application. Firstly, the provider wants to constrain its tenants, e.g., to ensure that a hospital has enough billing credit to perform a certain action or is billed afterward. Secondly, the patient monitoring application is provided to multiple hospitals (i.e., multi-tenancy) and the provider should make sure that one hospital cannot access the monitoring data of another hospital (i.e., tenant isolation). Finally, e-health applications are subject to regulatory requirements (e.g., the European DPD [71]) and even if the data is hosted by the provider, the hospital is still accountable for it. Therefore, the hospitals have to be able to apply their own access rules on their data in the application as well.

While the previous chapter already focused on these requirements, this chapter zooms in on the third point, i.e., the access control of the tenants on the SaaS application. In addition to supporting a wide variety of access control concepts for tenants to express there access rules, this access control should adhere to two requirements from the point of view of the tenants: (i) it should provide scalable management, (ii) it should take into account sensitive access control data.

**Requirement 1: scalable management.** As a first requirement, the access control offered to tenants should provide scalable management for these tenants. More precisely, the main goal of the tenants is to limit the access of their own end-users to their own data in the SaaS application. This access control is therefore highly interrelated with the user management of these tenants: user accounts have to be created, access control policies have to be deployed and the access control data of the users such as their roles in the organization, their department, their shifts and their assigned patients have to be managed. However, organizations such as a hospital typically employ multiple on-premise and SaaS applications, they can have a medical staff of thousands and they treat even more patients. Manual user management on this scale would incur too much administrative overhead and would quickly lead to errors, inconsistencies and security holes. Therefore, hospitals typically have extensive centralized security systems at hand, e.g., organization-wide patient management systems. When employing SaaS applications such as the patient monitoring application, this degree of centralization should be upheld and security management should remain centralized.

**Requirement 2: sensitive access control data.** Secondly, the access control management facilities offered to tenants should enable the tenants to employ sensitive access control data in their policies. For example, a typical access rule employed by a hospital in this case study can be summarized as follows: "*a physician can only view or alter patient data if the patient who owns the data is in a life-threatening situation or if the physician is treating that patient or if the data is relevant to the specialization of the physician and the patient has given consent to the physician*". This small rule already requires user roles, treating relationships, patient consent, resource content and more. Clearly, some of this access control data is sensitive in nature, such as the list of patients being treated by a physician, the diseases of a patient or patient consent. While the hospital trusts the provider with the monitoring data in the SaaS application, it does not necessarily want to trust the provider with this sensitive access control data. Moreover, regulatory requirements such as HIPAA [1] or the European DPD [71] even forbid the hospital to share this data.

These two requirements lead to the need for federated authorization.

### 4.2.3 The need for federation authorization

The need for federated authorization arises from the requirements listed in the previous section.

Firstly, the need for federated authorization arises from the need for scalable management (Requirement 1). This need however is not new, it already arose when web applications matured and were adopted by large enterprises. To address this need, the technique of federated *authentication* [2, 6] was developed. As explained in Section 2.4.3, federated authentication enables a web or SaaS application to externalize authentication from the provider to the premises of the tenant. For SaaS, the advantages of this technique are threefold: (i) it allows the tenants to employ any preferred means of authentication, (ii) it gives the tenants full control over the access control data shared with the provider and (iii) it allows the tenants to integrate the user management of this SaaS application with their on-premise user management systems, thereby reusing these and achieving scalable centralized user management.

However, user management is only part of the complete access control management: after users have been defined, the tenants have to specify the access policies that constrain them. While federated authentication allows to centralize the user management and the access control data of these users at the tenant premises, the policies themselves still have to be configured using a dashboard in the SaaS application and are still located at and evaluated by the provider. This has two disadvantages: Firstly, the tenants still have to specify their policies for every individual application they employ, which increases both the initial management overhead when a new tenant adopts the application and the overhead of the authorization management afterwards. This fails to address the need for scalable management (Requirement 1) and will lead to incorrect or inconsistent policies. As a second disadvantage, the tenants are still forced to disclose their policies and the access control data required to evaluate them, which goes against the need to employ sensitive access control data in these policies (Requirement 2). As a result, this disadvantage either limits the expressiveness of the policies of the tenants or hinders the adoption of the SaaS application itself.

Related work exists for addressing each of these disadvantages. On the one hand, administrative scalability for authorization can be achieved using automated policy deployment. For example, Stihler et al. [193] propose a system in which a consumer of a web service automatically deploys its policies at the web service and efforts such as SPML [68] try to standardize the resulting configuration interfaces. However, while these techniques provide administrative scalability, all policies are still evaluated by the provider and they still involve disclosure of sensitive tenant access control data.

On the other hand, confidential policy evaluation can be achieved by encrypting the access control policies and access control data when sharing them with the provider.

This is an instance of a more generally known form of encryption called homomorphic encryption (e.g., see [112]). Homomorphic encryption allows computations to be performed on encrypted data so that the result equals the encrypted form of the result of the computations on the original data. In other words, this technique would enable computation on data without knowing this data. This would also apply to the problem of confidential policy evaluation by enabling the provider to evaluate encrypted policies based on locally-available encrypted tenant attributes. However, the performance overhead of generic homomorphic encryption is still too much for practical applications [112]. A possible solution is to design problem-specific instances of homomorphic encryption. For confidential policy evaluation, this approach has been explored by Asghar et al. [34]. In this case, the performance overhead is in the order of hundreds of milliseconds for evaluating a policy. The expressiveness of the supported policies is still limited however and this approach does not address the additional challenge of the administrative overhead.

Finally, XML gateways such as IBM Tivoli Access Manager [129] can be applied to enforce access control on remote services or applications. An XML gateway is placed at the perimeter of the organization and reviews every request to a remote service, similar to a firewall. This approach also allows enforcing tenant access control policies based on local access control data. However, the policies are limited to reasoning about messages sent to and from the application and as a result, these gateways are mainly used for transparently adding credentials or for encryption in practice. Moreover, SaaS applications are often designed to be used by mobile users, i.e., users that access the application from outside the premises of their organization, while XML gateways require every request to originate from the physical network of the tenant.

In conclusion, no approach exists that can provide both scalable access control management and confidential policy evaluation. To address this gap, we investigate the technique of federated authorization.

## 4.3 Federated authorization

In order to address the requirements discussed in the previous section, we investigate the technique of *federated authorization*. Similar to federated authentication, the goal of federated authorization is to externalize authorization from a remote application (see Figure 4.2). As such, when a user makes a request to the SaaS application, the application asks the access control system of the tenant to which that user belongs for an access decision. This system evaluates its policies locally for this request and returns its decision, which the application enforces afterwards. Notice that for a SaaS application, the provider also evaluates its policies about the tenant as a whole and

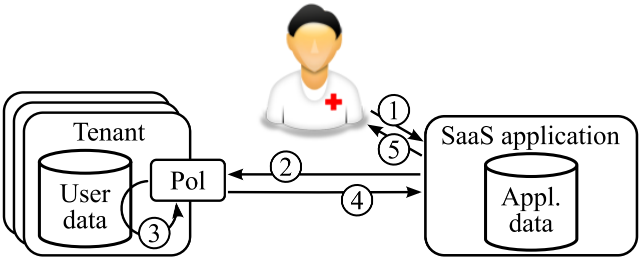


Figure 4.2: High-level overview of federated authorization applied to SaaS: When a user makes a request to a SaaS application (step 1), this application asks the access control system of the tenant to which this user belongs for an access control decision (step 2). This system evaluates its policies locally for this request (step 3) and returns its decision (step 4), which the application enforces afterwards, e.g. by returning the requested resource to the user (step 5).

combines this decision with the decision of the tenant so that the request is only permitted if both parties permit it as described in the previous chapter.

As a result, federated authorization allows the tenant to enforce access control on a SaaS application without having to disclose its policies nor the access control data required to evaluate them. In addition, federated authorization enables the tenant to integrate the policy management of a SaaS application with its on-premise management systems. When combined with federated authentication, federated authorization thus enables the complete centralization of the access control management of that tenant. Finally, while it was not our direct goal, federated authorization also enables a tenant to enforce any access rule expressed in any format and even employ any preferred supporting access control system, as long as it adheres to the interfaces agreed upon with the SaaS application. As such, federated authorization can also be regarded as a tactic to address the tenant variability challenge for access control discussed in the previous chapter.

In addition to this work, the technique of federated authorization has also been discussed by other authors, for example by Hulsebosch et al. in their study on federated access control in e-science [175]. In addition, simple domain-specific instances of federated authorization are already used in practice, e.g., for credit card payments where the bank of the customer is contacted to authorize a payment. Similarly, OAuth (see section 2.4.3) can be regarded as a basic form of federated authorization in which a policy is evaluated once, possibly by a third party such as a tenant, and the decision is cached afterwards.

Compared to these approaches, this work presents a generic investigation of federated authorization based on attribute-based access control policies. The resulting architecture therefore encompasses all of these approaches. To achieve this, the rest of this section describes the key features required for realizing federated authorization, a generic run-time middleware architecture and the required extensions to current policy languages. Afterwards, the next sections evaluate the performance behavior of federated authorization and validate the potential of this technique in federated applications outside the scope of SaaS.

## 4.3.1 Key features for supporting federated authorization

As explained in the introduction, we discuss federated authorization for SaaS applications that employ policy-based access control with attribute-based policies. In this context, federated authorization externalizes the evaluation of the tenant policies from the provider to the tenant. With respect to policy-based access control (see Section 2.3.2), federated authorization requires three additional features: (i) requesting an access control decision from the access control system of the tenant, (ii) handling local and remote attributes and (iii) handling local and remote obligations. Notice that these three features are required to support the most generic form of federated authorization, but may not be needed in every practical deployment. For each of these new features, we here determine what should be added to the XACML reference architecture (see Section 2.2.4) and to current policy languages.

### Key feature 1: Requesting tenant access control decisions

As a first key feature, the provider should be able to request an access decision from the access control system of the appropriate tenant for a certain request to the SaaS application. This feature impacts both the architecture and the policy language.

**Architecture.** Architecturally, this feature requires the tenant to provide a service to receive decision requests from the providers of the SaaS applications it employs. Such a request should identify the provider and should contain information about the subject, the resource, the action and the environment, similar to a request from a PEP to a PDP. Using this information, the access control system of the tenant determines and evaluates the applicable policies and returns its response. This response contains the decision itself (permit or deny) and possibly obligations, similar to a response from a PDP to a PEP. Notice that the provider also requires a way to identify a user and a way to know to which tenant a user belongs. For this, we assume that the provider has an authentication infrastructure in place that understands the concept of tenants.

**Policy language.** In terms of policy languages, the policy language should allow the provider to refer to the access control decision service of the tenant and specify how the result should be processed, similar to the result of an on-premise policy.

### Key feature 2: Handling local and remote attributes

As a second key feature, all required attributes should be made available to the respective policies. Because the provider reasons about its tenants as a whole (e.g., the hospital), the subjects of the policies of the provider are its tenants, the resources are the application data and the environment is the SaaS application. Thus, all attributes required by the provider policies are available locally. The policies of the tenants on the other hand reason about their end-users (e.g., the physicians and nurses of the hospital). Therefore, the subjects of the policies of the tenants are its end-users, the resources are the application data and the environment comprises both the SaaS application and the own access control systems. The data about the end-users of the tenant and the data in these systems are stored at the premises of the tenant, while the rest is hosted by the provider. Thus, the attributes required by the tenant policies are distributed over tenant and provider. As a result, evaluating the tenant policies requires to be able to employ both local and remote attributes. This impacts both the architecture and the policy language.

**Architecture.** Architecturally, this feature requires the attributes of the resources in the SaaS application and the attributes of the provider-side environment to be made available to the tenant for evaluating the tenant policies tenant-side. In addition, while attributes can be added to the initial request from the provider to the tenant, it is generally not possible to determine the required attributes for evaluating a policy for a certain request up-front. Therefore, the provider should provide a service to the tenant to dynamically fetch required attributes during policy evaluation.

**Policy language.** When evaluating the tenant policies, the policy evaluation engine should know where to find the required attributes. Therefore, the policy language should allow to define the location of each attribute referenced in the policies.

### Key feature 3: Handling local and remote obligations

As a third key feature, the tenant should be able to handle both tenant-side and provider-side obligations. As explained in Section 2.2.4, obligations express operations that should be executed in conjunction with enforcing the access decision. An example of a tenant-side obligation is logging and an example of a provider-side obligation is updating the access control history of a resource. Allowing the tenant to employ both local and remote obligations impacts both the architecture and the policy language.

**Architecture.** Architecturally, this feature requires the response from the tenant policy evaluation service to contain the obligations specified by the tenant which should be fulfilled by the provider. This was already mentioned before. The tenant response should not contain locally fulfilled obligations.

**Policy language.** In terms of policy languages, the policy language should allow the tenant to specify where obligations should be fulfilled: locally or remotely.

### 4.3.2 Generic middleware architecture

Based on the architectural requirements listed in the previous section, we now present a generic policy-based and attribute-based architecture for federated authorization. Our goal here is to provide a minimal architecture that supports these requirements. We therefore align to the XACML reference architecture for policy-based access control systems (see Section 2.3.2) as closely as possible. In this section, we first describe the decomposition of the architecture, then illustrate the resulting access control flow and finally describe the variation points left open by this generic architecture.

**Architecture decomposition.** Figure 4.3 shows the decomposition of the middleware architecture. First of all, the provider hosts the SaaS application and therefore also the PEP; no application components are located at the tenant side. Since both parties evaluate policies and process obligations, both have a PAP, a PDP, a Context Handler, one or more PIPs and an Obligation Service. The provider PIPs contain the attributes of the resources in the SaaS application ($A_R$), the attributes of the subjects of the policies of the provider ($A_{S,P}$) and the attributes of the provider-side environment ($A_{E,P}$). The tenant PIPs contain the attributes of the subjects of the policies of the tenant ($A_{S,T}$) and the attributes of the tenant-side environment ($A_{E,T}$). For handling decision requests, the tenant offers a Remote Policy Decision Point (RPDP) to the provider. For handling attribute requests, the provider offers an attribute service to the tenant. The provider PDP is extended with functionality to contact the RPDP and the tenant Context Handler is extended with functionality to contact the provider attribute service. To summarize, the resulting interface between provider and tenant consists of two services: the tenant policy evaluation service and the provider attribute service. Notice that the architecture still allows the tenant to use the tenant-side components for its on-premise applications as well, which is not shown in the figure.

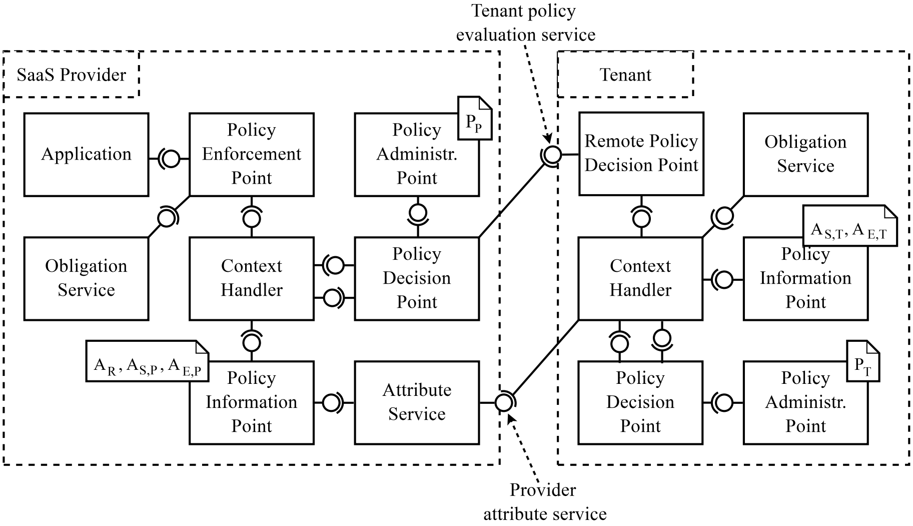


Figure 4.3: The generic architecture for federated authorization. $P_P$ and $P_T$ are the provider and tenant policy sets respectively and $A_R$, $A_{S,P}$, $A_{E,P}$, $A_{S,T}$ and $A_{E,T}$ are as defined in Section 4.3.2.

**Access control flow.** Figure 4.4 illustrates the access control flow that results from the generic architecture. The presented flow starts after the provider and tenant policies are loaded from their respective PAPs (step 0) and the end-user has been successfully authenticated. The remainder of the flow is as follows:

Step 1 When an end-user makes a request to the application, the PEP constructs an access control request and sends this to the local Context Handler.

Step 2 That Context Handler forwards this request to the local PDP.

Step 3 The provider PDP determines the applicable provider policies and evaluates these, thereby dynamically requesting attributes via the Context Handler. When the PDP encounters a remote policy reference, it asks the Context Handler to determine how and where to contact the appropriate tenant. The PDP then constructs a decision request and sends it to the tenant RPDP.

Step 4 From the tenant point of view, the RPDP acts similarly to a PEP and forwards the request of the provider to the tenant Context Handler.

Step 5 That Context Handler forwards this request to the tenant PDP.

Step 6 The tenant PDP determines the applicable tenant policies and evaluates these for this request, thereby dynamically requesting attributes from the local Context

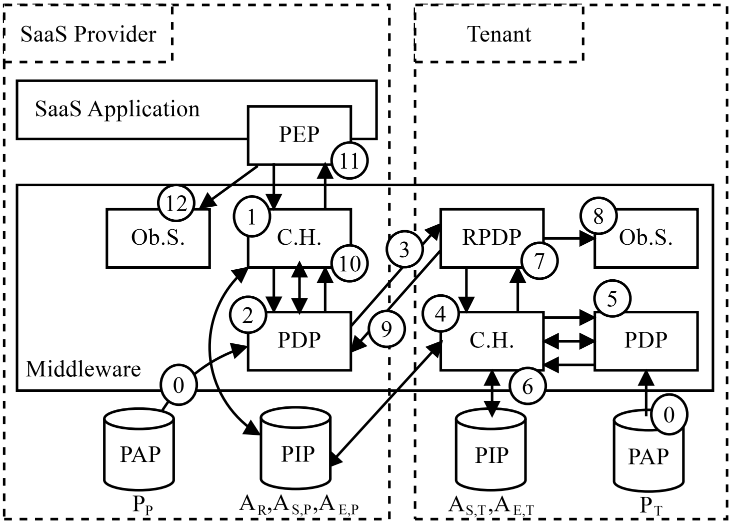


Figure 4.4: The access control flow resulting from the generic architecture of Figure 4.3. C.H. is Context Handler, Ob.S. is Obligation Service, $P_P$ and $P_T$ are the provider and tenant policy set respectively, PEP, PDP, PIP and PAP are as defined in Section 2.3.2 and $A_R$, $A_{S,P}$, $A_{S,T}$, $A_{E,P}$ and $A_{E,T}$ are as defined in Section 4.3.2. For readability reasons, the provider attribute service is not shown explicitly.

Handler. That Context Handler fetches tenant attributes locally and contacts the provider attribute service for provider attributes. The tenant PDP eventually returns its response (i.e., decision and obligations) to the local Context Handler.

Step 7 The Context Handler returns the response to the RPDP.

Step 8 The RPDP fulfills tenant obligations using the tenant Obligation Service and removes these from the response.

Step 9 The RPDP returns the resulting response to the provider PDP.

Step 10 The provider PDP combines the tenant decision with the result of the provider policies so that the request is only permitted if both parties permit it and returns the overall response to the provider Context Handler.

Step 11 The provider Context Handler returns the response to the PEP.

Step 12 The PEP fulfills the remaining obligations using the provider Obligation Service and enforces the decision.

**Variation points.** This architecture deliberately leaves open a number of variation points: the attribute fetching strategy, the channel protocols and the channel security properties.

*Attribute fetching strategy.* The generic architecture leaves open the attribute fetching strategy for tenant policy evaluation. In general it is not possible to determine the required attributes for evaluating a policy for a certain request up-front and it should therefore be possible to fetch attributes during a policy evaluation. However, there are multiple ways to do this, ranging from one-at-a-time to combined requests, using caching or not. Moreover, the provider could manually configure some attributes to be included in the initial policy evaluation request to the tenant so that these attributes should not be fetched by the tenant later on.

*Channel protocols.* The generic architecture leaves open which protocol to use for the attribute channel or the decision channel. For example, the SAML standard for federated authentication [2] defines two alternatives for web applications: in-band communication using HTTP redirects through the end-user browser and out-of-band communication using direct connections between identity provider and relying party, such as SOAP web services.

*Channel security.* The generic architecture leaves open the security properties of the attribute channel or the decision channel, such as transmission security or authentication.

### 4.3.3 Extensions to current policy languages

In addition to the supporting architecture, the three key features to support federated authorization identified in Section 4.3.1 all required policy language extensions. To illustrate the practical impact of these requirements, we have extended the XACML 2 policy language [165]. We opted for XACML because of its wide-spread use in both academia and industry, because of its active development and because STAPL (see Section 2.3.1) was not yet in development at the time of this work. Because STAPL and XACML share the same core model however, the required extensions would be very similar.

In this section we describe the extensions that we introduced, their specifications are provided in Appendix B. We start by briefly introducing XACML, for a more detailed explanation we refer to Section 2.3.1.

**The XACML policy language.** The XACML policy language is defined in the XACML standard in addition to the reference architecture for policy-based access control systems [165]. XACML employs attribute-based policy trees and expresses policies using XML. Three main elements are defined: `<PolicySet>`, `<Policy>` and `<Rule>`. A policy set can contain multiple policies and a policy can contain multiple rules. A rule specifies an effect (permit or deny), attribute-based conditions for this to hold and obligations to fulfill with it. A policy combines the results of its rules using a rule-combining algorithm (e.g., deny overrides) while a policy set combines the results of its children using a similar policy-combining algorithm. Each of these three elements also specifies to which requests it applies in terms of attributes.

**Referencing remote policies.** In order to reference remote policies, the `<RemotePolicyReference>` element is introduced. This element refers to a tenant as a whole, which behaves as a remote policy from the point of view of the provider. This approach is similar to the deductive policies introduced by Lischka et al. [150]. Evaluating a `<RemotePolicyReference>` returns a policy evaluation result similarly to the `<Policy>` element and the element can be part of a policy set. The `PolicyId` attribute specifies the id of the remote policy. Following the XACML design principles, it is left to the Context Handler to determine how and where to contact it. A remote policy reference can also contain a description, a target and obligations for local use.

**Handling local and remote attributes.** In order to differentiate between local and remote attributes, we follow the XACML design choice of having the Context Handler infer the location of an attribute based on its id instead of defining attribute properties declaratively. Thus, XACML is not extended for this requirement, but we do require the location of every attribute to be configured in the Context Handler.

**Handling local and remote obligations.** In order to differentiate between tenant and provider as obligation targets, the `<Obligation>` element is extended with the optional `FulfillWhere` property. This property specifies whether the obligation should be fulfilled locally (with the tenant) or remotely (with the provider), local fulfillment being the default. The new property is only to be used in tenant policies. It is the responsibility of the tenant to remove local obligations from its response so that the provider can interpret all obligations as before.

## 4.4 Performance evaluation

Following the description of the middleware supporting federated authorization, this section evaluates the concept of federated authorization in terms of performance based on a prototype of this middleware. More precisely, this performance evaluation investigates the impact of federation on the end-to-end time it takes for the provider to reach an access control decision.

### 4.4.1 Test setup

The tests compare three different cases of evaluating the tenant policies:

1. *Complete provider-side authorization*: both the policies of the tenant and its access control data are located at the provider side. As a result, the provider evaluates the tenant policies solely based on local access control data. This case can be expected to give the best performance results.
2. *Provider-side authorization with federated authentication*: the access control data of the tenant is located at the tenant side, but its policies are located at the provider side. As a result, the provider evaluates the tenant policies, fetches provider access control data locally and fetches tenant access control data from the tenant attribute service.
3. *Federated authorization*: both the policies of the tenant and its access control data are located at the tenant side. As a result, the provider requests an access control decision from the tenant RPDP, which fetches tenant attributes locally and fetches provider attributes from the provider attribute service.

Notice that of these three approaches, only federated authorization keeps sensitive access control data of the tenant confidential. Also notice that the policies of the provider are not evaluated in the tests for clarity because federated authorization is only used to externalize the evaluation of the policies of the tenant.

**Prototype.** The prototype of the middleware implements the architecture described in Section 4.3.2. As a result, the prototype contains four main components: (i) the provider PDP, (ii) the tenant PDP (iii) the provider attribute service and (iv) the tenant attribute service. For the PDPs, the prototype extends the SunXACML policy evaluation engine[1] with the new `<RemotePolicyReference>` element. Attributes are stored in SQLite databases[1]. Cross-organizational communication is realized out-of-band using SAML [2] and the SAML profile of XACML [154] over SOAP web-services[1] implemented on top of Apache Tomcat 7[1] using the Apache CXF services framework[1] and the OpenSAML Java library[1]. For similarity, both the provider and the tenant PDP are run on top of Tomcat. In order to focus on policy evaluation

[1] http://sourceforge.net/projects/sunxacml/, http://www.sqlite.org/, http://cxf.apache.org/, http://tomcat.apache.org/, https://wiki.shibboleth.net/confluence/display/OpenSAML/

time, the prototype also omits channel encryption or authentication. The prototype (6KLOC) is publicly available[2].

**Deployment.** In our set-up, the provider PDP, the tenant PDP, the provider attribute web service and the tenant attribute web service are all run on a separate machine, each with 4GiB RAM and two cores of 2.40GHz running Ubuntu 12.04. In order to focus on the impact of remote communication, local attribute databases are run on the same node as the services that use them. To simulate the distance between tenant and provider in a realistic SaaS setting, a constant single-way network delay of 5ms between tenant and provider is applied. In order to avoid influence of parallelism, the tests are run sequentially and PDP evaluation is done single-threaded.

**Employed policies.** The tests involve artificial policy evaluations that require 10, 20 and 30 attributes. Our experience in in our case studies shows that these amounts represent modest to large policies. Because multiple strategies for fetching attributes exist ranging from one-at-a-time to combined requests (see Section 4.3.2), the tests also compare the two extreme strategies: fetching all required attributes separately and fetching all attributes at the same time as one multi-valued attribute. All attributes are fetched just-in-time and no attribute caching is used in order to simulate the worst case.

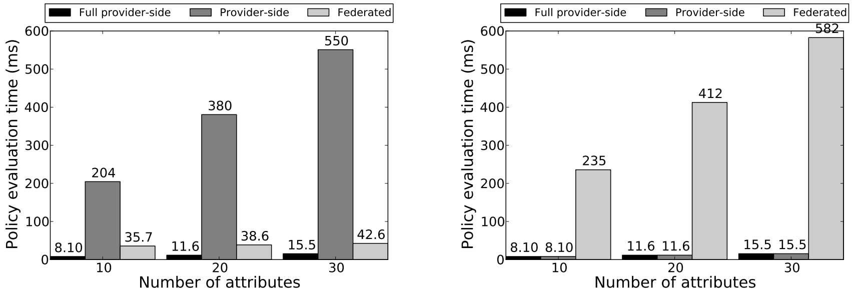
**Test protocol.** Each test starts with five warm-up requests and is repeated until the confidence interval lies within 1% of the sampled mean for a confidence level of 95%.

### 4.4.2 Results

Figure 4.5 shows the results for the case in which all attributes are fetched separately[3]. More precisely, Figure 4.5 shows the decision time in relation to the amount of single-valued attributes. If the policy only requires tenant attributes (Figure 4.5a), provider-side authorization with federated authentication performs significantly worse since each attribute is fetched separately when the provider PDP requires it and each query takes about 20ms as a result of the network latency and XML operations. If the policy only requires provider attributes (Figure 4.5b), federated authorization performs worse. Complete provider-side authorization and provider-side authorization with federated authentication have the same results in this case since all attributes are hosted by the provider.

[2] `http://people.cs.kuleuven.be/~maarten.decat/doa-trusted-cloud-2013/`

[3] The whole set of results of the tests is publicly available at[2].



(a) Only tenant attributes

(b) Only provider attributes

Figure 4.5: Decision time in terms of the amount of single-valued tenant and provider attributes.

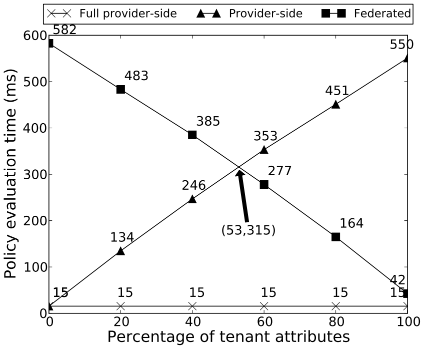
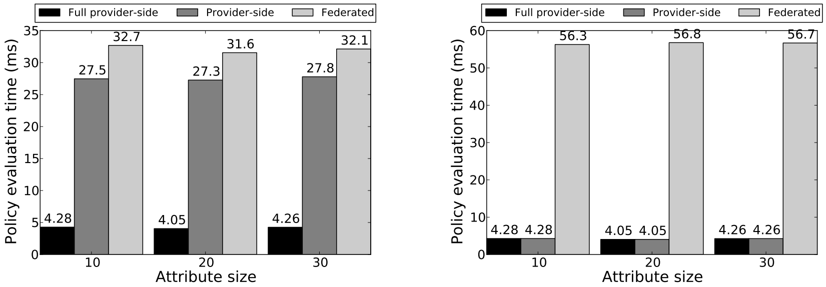


Figure 4.6: Decision time in terms of the percentage of tenant attributes in a total of 30 attributes.



(a) Large tenant attribute

(b) Large provider attribute

Figure 4.7: Decision time in terms of the amount of values in a single large provider or tenant attribute.

Figure 4.6 combines the tests with multiple tenant attributes and multiple provider attributes. This figure shows the decision time in relation to the percentage of tenant attributes in a total of 30 attributes. The figure firstly shows that complete provider-side authorization performs best in all cases. This is a natural consequence of the fact that complete provider-side authorization requires no remote queries. More interestingly however, Figure 4.6 also shows that federated authorization outperforms provider-side authorization with federated authentication when a little more than half of the attributes are tenant attributes. The asymmetry is a consequence of the extra request needed with federated authorization with respect to provider-side authorization with federated authentication.

Figure 4.7 shows the results for the case in which all required attributes are fetched at once as a single multi-valued attribute. More precisely, Figure 4.7 shows the decision time in relation to the size of the single attribute, i.e., the amount of values in this attribute, in this case 10, 20 or 30 values. If the policy only requires a tenant attribute (Figure 4.7a), complete provider-side authorization again performs best and both other cases perform similarly since they both involve a single attribute request between provider and tenant. The difference between provider-side and federated authorization is due to the complexity of the tenant-side operation: attribute fetch versus policy evaluation. If the policy only requires a provider attribute (Figure 4.7b), federated authorization performs significantly worse than the others, since only this case requires remote queries. In all cases, the decision time remains constant independent of the size of the required attribute. The absolute difference between Figure 4.7a and Figure 4.7b is caused by two requests instead of one. The absolute difference between many and large attributes in the complete provider-side case is caused by the fact that the amount of required XML translations grows linearly with the amount of attributes. Since these amounts are constant with respect to the attribute size, we did not combine them as in Figure 4.6.

**Summary.** From these results, we can conclude that federated authorization comes with a performance penalty compared to complete provider-side authorization. However, depending on the relative amount of tenant attributes in the tenant policies, federated authorization can achieve better performance than provider-side authorization with federated authentication. To illustrate with data from a realistic case, the example policy rules for the patient monitoring application require significantly more tenant attributes than provider attributes: the tenant hosts the subject roles, treating relationships, patient consent and patient diseases while the provider only hosts ownership relations and the application data itself. As a result, federated authorization has the ability to actually improve performance for this case.

## 4.5 Discussion

In the previous sections, we illustrated the need for federated authorization in SaaS applications, presented a generic architecture for this technique and evaluated its performance characteristics. In this section, we further discuss the trust, security and privacy implications of federated authorization and discuss potential performance improvements.

### 4.5.1 Trust implications

From the point of view of the provider, federated authorization has no impact on the required trust in the tenant: since tenants enforce access control on their own data in the application, they have no incentive to provide false or incorrect decisions.

However, from the point of view of the tenants, federated authorization does affect the required trust in the provider. Most importantly, federated authorization *lowers* the required trust in the provider since it removes the need to share sensitive access control data and policies. This was one of our explicit requirements for this technique. However, a potential threat here lies in the fact that the provider could still infer information about this data and policies, e.g., by re-engineering the policies, using the collection of access control requests and responses from the tenant collected over time. For this issue, we argue that the possibly inferred knowledge is limited since both the tenant policies and the access control data used for evaluating them are kept unknown. However, future work is required to answer this question more quantitatively, for example by using techniques such as logical abduction.

In addition to the threat of knowledge inference, two trust requirements still exist: the tenant still has to trust the provider (i) for full mediation, i.e., that it is contacted for every request to the SaaS application and (ii) for correct decision enforcement, i.e., that its decision is actually enforced correctly by the provider. Notice however that both issues were also present with provider-side authorization and that although federated authorization cannot remove these trust requirements, it does provide a basis for mitigating them: Full mediation is still not provable, but does become verifiable since the tenant is able to experimentally verify that it is contacted for every request to the SaaS application through sample-based testing. Similarly, correct decision enforcement becomes experimentally verifiable as well. Moreover, it can be made provable by extending the policy evaluation communication protocol with non-repudiation techniques [137]. However, these techniques are known to have a large architectural impact (e.g., on performance) and their incorporation would not be trivial.

### 4.5.2 Security implications

While federated authorization lowers the required trust in the provider, it does also introduce new security threats.

From the point of view of a network attacker, the main difference between provider-side and federated authorization is the externalization of the policy evaluation process. This results in a new communication channel between a tenant and the provider, which introduces a denial of service threat: since a tenant decision is required for every request to the SaaS application, blocking the channel or the services is a way to deny the use of the application for a specific tenant. This threat cannot be easily mitigated and as a result, there should be a fallback decision configured in case the tenant cannot be contacted. The channel should also be secured against the threats of information disclosure, tampering and spoofing and against attacks such as replay. For SOAP web services, WS-Security [144] provides the necessary security primitives.

### 4.5.3 Privacy implications

Apart from trust and security, federated authorization also has an impact on privacy.

As discussed up until now, federated authorization has two main benefits: it enables to centralize the authorization management of an organization and it benefits confidentiality by enabling an organization to enforce a policy on an application without having to disclose this policy nor the data that it requires. While we have not explicitly discussed it as such, privacy was one of the main drivers for this confidentiality within the case study of the patient monitoring system. In this case, the hospital primarily does not want to share the privacy-sensitive data required to evaluate a policy such as names or pathologies of patients. By extension, the hospital may even be forbidden to share it because of legislation such as the European Data Protection Directive [71]. As such, federated authorization can benefit privacy with a focus on the confidentiality of privacy-sensitive data.

Privacy however is more than confidentiality [91]. Other aspects are amongst others the disclosure of the identity of a person or the abilities to observe or link the actions of a person across multiple applications. In this regard, the effect of federated authorization can be both positive and negative. For example, in case the SaaS provider is not trusted, federated authorization also enables the use of pseudonyms, which benefits privacy by avoiding the release of the identity of a subject. In case a person does not trust the organization at which authorization is centralized however, federated authorization has a negative effect on privacy since it facilitates centralized control, linkability and observability. These issues are the result of designing this technique for untrusted SaaS providers and cannot easily be addressed.

### 4.5.4 Performance

As shown by our performance evaluation (see Section 4.4), federated authorization can be used as a performance tactic in a federated context by bringing policy evaluation to the data that it requires instead of the other way around. However, compared to complete provider-side authorization, federated authorization does come with a non-negligible performance penalty. Complete provider-side authorization however forces the tenant to disclose sensitive access control data to the provider. Therefore, we can conclude that federated authorization poses a clear trade-off between trust and performance.

Towards the future, the performance of federated authorization can be further improved. First of all, the performance evaluation shows that the overhead of federated authorization is highly dependent on the amount of requests between the provider and the tenant. Therefore, a good strategy for fetching attributes is essential to improve performance. This has also been discussed by Brucker et al. [58].

Secondly, the use of caching also has the ability to lower the required provider-tenant communication, for example by caching attributes or access decisions. However, caching also has an impact on security as it can lead to policy evaluation based on stale attributes or even stale cached access decisions.

Finally, a more fine-grained and dynamic policy deployment strategy also has the ability to improve performance, even while maintaining the trust advantages. This chapter has investigated two extremes: the tenant policies are either completely evaluated by the provider or completely by the tenant. Based on the location of the required access control data and its sensitivity, the tenant policies could be split and distributed for optimal performance while maintaining the confidentiality of the tenant access control data. Since access control is a security feature, the proven correctness of this operation is essential. In the next chapter, we present a technique to automatically split and deploy the tenant policies across tenant and provider for improved performance.

## 4.6 Validation of federated authorization in a wider context

In the previous sections, we discussed the technique of federated authorization in the context of SaaS. Similar to federated authentication however, this technique is applicable outside the scope of SaaS as well. Therefore, as a generalization of federated authorization and a validation of its potential, we applied this technique to a second case study, i.e., a collaboration platform in the domain of e-health. This case study can be seen as more complex than SaaS because more than two parties have to collaborate.

In this section, we first describe the case study, then highlight the requirements for access control and finally discuss the role of federated authorization.

### 4.6.1 Case study: a collaborative care platform

In order to validate the potential of federated authorization, we applied it to the case study of a collaboration platform in the domain of e-health. This case study is an example of the multiple collaboration platforms that are currently being created in the domain of e-health in order to facilitate collaboration between increasingly specialized care organizations. This case study is based on the O'CareCloudS research project [15].

The collaboration platform in this case study aims to facilitate information sharing between the multiple organizations that provide home care to a certain care receiver, such as a physiotherapist, a general practitioner, service flats and a hospital. The platform therefore digitizes the information that these organizations share, such as prescriptions of and notes about the care receiver.

A collaborative application such as this collaboration platform can be seen as an extension to the SaaS. More precisely, while the federation of a SaaS application only consists of multiple one-on-one collaborations between the provider and each tenant, the federation of a collaborative application consists of a larger amount of organizations that all have to work together.

Figure 4.8 illustrates the federation of organizations for the care platform. While this figure only illustrates a part of the complete federation, it already shows that a large amount of organizations is involved, such as general practitioner practices, elder homes, daily care organizations, physiotherapist practices, home nursing organizations, hospitals and catering services. Some of these are directly involved (e.g., the meal delivery service), others indirectly because of business relationships (e.g, the caterers) or because of the integration of the platform with the Electronic Health Record (EHR) for sharing patient data on a wider scale (e.g., other hospitals). Figure 4.8 also illustrates that the organizations in the federation are of very different nature, ranging from core-medical (e.g., general practitioners and hospitals) to supporting (e.g., caterers), and from large organizations (e.g., hospitals) to small organizations (e.g., a medical imaging practice) and even individuals (e.g., family, friends, general practitioners). Moreover, this federation is dynamic in the sense that new organizations can join the federation over time or others can leave.

The data shared in this federation via the collaboration platform can either be located

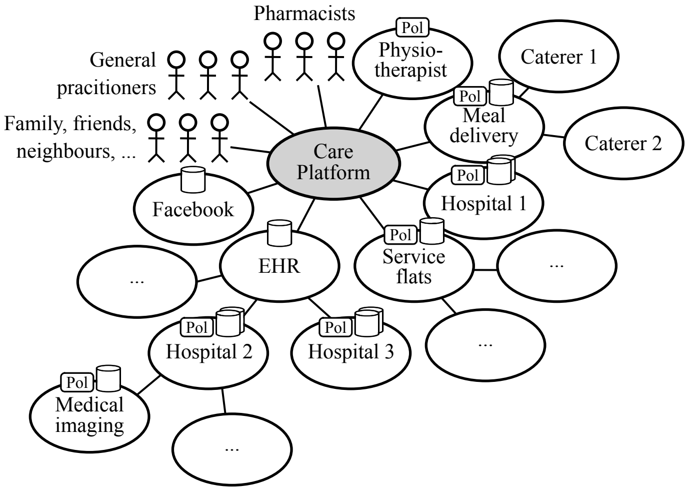


Figure 4.8: Part of the federation of organizations involved in the collaborative care platform. As illustrated, such a platform quickly leads to a federation of a large amount of organizations and individuals of different nature. Moreover, because every organization remains a separate domain of management, the access control data (indicated by the cans) and the policies that apply to the care platform are scattered across the federation.

in the platform itself, e.g., in case a home nurse submits a summary of his or her visit to the patient in a web front-end, or at the owning organization, e.g., in case a hospital shares the data about a patient managed by its patient management application. In any case, the other organization accesses this data through the care platform so that the platform acts as a central hub.

### 4.6.2 Access control requirements

While the collaboration platform aims to facilitate information sharing between the involved organizations, this information is sensitive, personal and medical. Therefore, the platform should also enable more *controlled* information sharing. To achieve this, the collaboration platform should perform access control.

In this case, there are three types of access control policies in this collaboration platform:

1. *Intra-organizational*: Firstly, the care organizations want to constrain their own employees. For example, the hospital imposes that each of its nurses can only

view data of care receivers explicitly assigned to him or her, which depends on internal task assignment. These policies are specific to each individual organization in the federation. As such, these policies are similar to the tenant policies in SaaS, with the difference that these policies should now also be enforced when an employee of the hospital accesses data of *another* organization in the federation through the platform.

2. *Inter-organizational*: Secondly, the care organizations want to control which other organizations are allowed access to their data. For example, the hospital trusts the meal delivery service, but not a private hospital that also uses the platform. Again, these policies are specific to each individual organization in the federation.

3. *Cross-organizational*: Thirdly, the platform itself imposes access control polices across all organizations. For example, these policies can impose that medical data can only be read by medical professionals that are not family members of the related patient, unless explicitly allowed by that patient. These policies are similar to the provider policies in SaaS, with the difference that these policies apply to all organizations using the collaboration platform.

Access control for this collaboration platform should support all three types of policies, for which federated authorization can play an important role.

### 4.6.3 The role of federated authorization

Federated authorization can play an important role in enforcing the policies mentioned in the previous section. More precisely, the platform operates as a hub through which all end-users access the data and the policies should therefore be enforced by the platform. As such, the platform behaves as a remote application from the point of view of the involved organizations, similar to a SaaS application from the point of view of a tenant. As a result of this similarity, the collaboration platform is faced with similar challenges for access control as SaaS.

These similar challenges are most apparent for the intra-organizational policies of each organization. In this case, federated authorization again enables these organizations to enforce their intra-organizational policies without sharing these or the required access control data with the other organizations or with the platform itself. In addition, federated authorization also enables scalable management of these policies from the point of view of the organizations, which again is mainly a requirement for large organizations such as a hospital. Finally, federated authorization enables these organizations to express any preferred policy in any preferred format and employ any preferred access control system.

A difference with SaaS however is that in addition to the intra-organizational policies, federated authorization also applies to the enforcement of the inter-organizational and cross-organizational policies. For these policies as well, the resource data is located in the platform, but the subject data is located in the home organization of those subjects. As a result, federated authorization can again be applied to evaluate (parts of) these policies at the premises of the organization of the subject, which can maintain confidentiality of this data or can be beneficial for performance if these policies require a lot of access control data located in the access control systems at this organization. Federated authorization can also be applied to the inter-organizational policies to keep the policies themselves confidential. This can be useful as it is likely that these policies are sensitive because they reason about other organizations in the federation.

However, as opposed to the policies of a tenant in SaaS or the intra-organizational policies in this collaboration platform, applying federated authorization to the inter-organizational and cross-organizational policies can also cause trust issues. The reason for this is that the inter-organizational and cross-organizational policies are not defined by the organization of the subject on which they are enforced. More precisely, the inter-organizational policies of an organization are defined by this organization, but enforced when a subject of *another* organization accesses its data; the cross-organizational policies are defined by the platform itself and are enforced when *any* subject access the platform. As a result, federated authorization for these policies leads to a situation in which a policy defined by a certain organization is evaluated by another organization. Therefore, the former organization should trust the latter one for correct policy evaluation. If there is no trust, federated authorization should be extended with additional techniques such as non-repudiation techniques or logical proofs. This issue was not present for the intra-organizational policies or for the tenant policies in SaaS because there, federated authorization is used to have an organization evaluate its own policies.

As a side remark, notice that the inter-organizational and cross-organizational policies also cause semantical challenges. More precisely, these policies are defined by a certain organization and reason about the subjects of another organization. As a result, the semantical challenge arises of how the policies of one organization can reason about the members of other organizations while for example the meaning of the role "nurse" can differ substantially in different organizational contexts. This challenge has previously been the subject of research in the context of role-based access control [108, 65], but is enlarged for the more expressive model of attribute-based access control [150, 175, 123]. In addition, when reasoning about other organizations as a whole, the cross-organizational policies require policy mechanisms that make abstraction of specific members of the federation such as higher-level federation roles such as "home care provider", "caterer" or "hospital" because realistic federations can be large and members can join and leave frequently. Both these semantical challenges are essential for access control in complex federations such as this collaborative care platform, but they are not specific for the technique of federated authorization.

In summary, federated authorization provides benefits on multiple levels. Semantically, it can facilitate enabling each organization to express policies in terms of its own organizational structure. Technically, federated authorization enables to evaluate a policy distributedly across organizational boundaries using any preferred system. In addition, federated authorization can be applied to improve performance of this distributed policy evaluation if the policy requires a lot of access control data located in a certain organization. In terms of trust, federated authorization enables the enforcement of policies without having to disclose them or the access control data they require. And finally, in terms of management, federated authorization enables the centralization of policy management. Although federated authorization does not address all of these challenges completely by itself, all of this does lead us to believe that federated authorization is a necessary building block for future federated access control in which data locality and inter-organizational trust relations dictate policy evaluation.

## 4.7 Outlook

In this chapter, we discussed the technique of federated authorization. We first discussed this technique in the context of SaaS and afterwards validated its applicability in the wider scope of inter-organizational collaborative applications. As we discussed, this technique has multiple potential benefits, such as lowering the required trust in the provider of a remote application and enabling scalable centralized access control management. This leads us to believe that federated authorization is a necessary building block for future federated access control.

In general, federated authorization fits into a technology domain referred to as Identity and Access Management or IAM. This domain mainly focuses on scalable and efficient access control management, a challenge that grows in importance as more and more information is digitized while companies grow in size and complexity.

The approach of externalizing authorization from an application (but not necessarily to another organization) can play an important role in the field of IAM, in the first place by enabling the centralization of the access management of an organization. By also representing the access rules as declarative policies, this approach additionally facilitates policy consistency, policy audits and policy reuse (as also discussed by Hulsebosch et al. [175]). In short, this approach enables a better overview of the overall access management of an organization.

Federated authorization instantiates this externalized authorization for the remote applications used by an organization. Towards the future, federated authorization even enables the creation of third-party authorization services that centralize the access management for an organization and relieve it from the management of the required infrastructure, an evolution called security as a service [187] or access control as a service [107]. The Kantara project called User-Managed Access (UMA, [23]) aims to achieve this for individual web users in order to improve their control over their web data based on OAuth.

However, we still see several challenges that have to be addressed in order to achieve wide-spread adoption of federated authorization in practice. Firstly, federated authorization requires standardization to easily employ it for multiple remote applications and to support dynamic and large-scale federations. More precisely, the large-scale adoption of federated authorization requires standardization of the interfaces and protocols between the involved parties. Initial steps in this direction are currently being made, e.g., by the OpenLiberty project called OpenAz [24].

Secondly, the performance of federated authorization has to be further improved. The reason for this is that authorization should be enforced on every action that a subject performs in an application. As such, if federated authorization is applied as described in the generic architecture, its overhead affects the end-user latency of every action in the application. In this regard, the OAuth protocol can be regarded as a simplified form of federated authorization that pragmatically opts for evaluating a coarse-grained policy once and caching the decision afterwards. This avoids the overhead for future requests of the same subject, but does not support fine-grained decisions for each individual action and increases the time to react to permission changes. In addition, our performance evaluation of the generic architecture shows that a large part of the policy evaluation time is spent on fetching attributes for fine-grained rules. Because of this, practical instances of federated authorization such as most OAuth instances opt for sharing a fixed set of attributes, which improves performance but restricts the expressivity of the involved policies. Towards the future, additional performance tactics are required in order to improve the capabilities of practical deployments of federated authorization.

Finally, the most fundamental challenge for successfully applying federated authorization in practice is that it builds on externalizing authorization from an application, which is still a challenge by itself in the first place. The reason for this is that compared to authentication, authorization is more tightly coupled with the application, both technically and semantically. Technically, authorization should be enforced on every action performed by a subject in the application. As a result, authorization interacts with multiple parts of the application code, which makes it non-trivial to modularize. Semantically, a policy author still has to be able to reason about the application on which the policy is enforced, e.g., the author has to know which resources are available, which attributes they provide and which actions they support. This makes it non-trivial to define a correct externalized access control policy, which lowers the benefits of externalized authorization and federated authorization. This challenge also explains why most instances of federated authorization in practice employ a limited or domain-specific access control model. We see this challenge as a major challenge for policy-based access control in general. Though this is not the main focus of this work, we further discuss our vision on addressing it in Chapter 7.

## 4.8 Conclusion

This chapter investigated the concept of federated authorization. Federated authorization externalizes authorization from a remote application such as a SaaS application. Federated authorization thereby, amongst others, enables to enforce a policy on an application without disclosing the policy or the access control data that it requires and enables to centralize the access management of an organization. In this chapter, we illustrated the need for this technique in the context of SaaS, described a generic attribute-based middleware architecture, evaluated the performance of this middleware based on a prototype, validated the potential of federated authorization outside the scope of SaaS and discussed the practical applicability of this technique. Although this chapter mainly focused on two e-health case studies, our interactions with industry partners have confirmed the need for federated authorization in other domains that rely on inter-organizational collaboration, such as electronic document processing [16] (the eDocs case study) and payment services [17]. As such, our experiences lead us to believe that federated authorization is a necessary building block for future federated access control.

However, while the concept of federated authorization is fairly intuitive, its realization still poses many challenges. As a result, federated authorization does not show a large adoption in practice yet. One of these challenges is the difficulty of externalizing authorization from an application and supporting fine-grained policies in the first place. As we regard this challenge as crucial to policy-based access control as a whole, we further discuss it in Chapter 7. A second challenge is the negative performance impact of federated authorization. With regard to this, we present the performance tactic of *policy federation* in the next chapter.

# Chapter 5

# Efficient federated evaluation of access control policies

This chapter presents our third contribution: the technique of *policy federation*. Following up on the previous chapter, policy federation optimizes the performance of federated authorization. Policy federation achieves this by automatically decomposing the tenant policies so that the resulting parts can be evaluated near the data they require as much as possible while keeping the sensitive access control data and policies of the tenant at its premises. While we primarily present this technique in the context of federated authorization, it can effectively be applied to any policy evaluation system in which the data required to evaluate a policy is spread across multiple locations, such as the multi-tier policy decision point of Amusa.

This chapter stems from both the goal of lowering the management overhead for tenants as well as from the goal of limiting the disclosure of sensitive tenant access control data and rules. In addition, this chapter focuses specifically on performance. As such, this chapter focuses on the challenges from outsourcing and the concern of low performance overhead (see Section 1.2). This chapter is mainly motivated by the case study of home patient monitoring and is based on our publications at MW4NG 2012 [84] and in the Journal of Internet Services and Applications [85].

## 5.1 Introduction

The previous chapter investigated the concept of federated authorization. Federated authorization externalizes policy evaluation from a SaaS application so that it can be performed at the premises of the tenant. This enables a tenant to centralize its access management and at the same time enables it to enforce a policy on a SaaS application without disclosing this policy nor the access control data that it requires. In addition, federated authorization can be beneficial for performance if the involved policies mostly require data located at the tenant because it brings policy evaluation to this data instead of the other way around.

In most cases however, the policies of the tenant also reason about the resources in the SaaS application and therefore also require data located at the SaaS provider. As a result, completely evaluating the policies of the tenant at the premises of this tenant leads to sub-optimal performance by having to fetch this data. Similarly, completely evaluating the policies of the tenant at the premises of the provider leads to sub-optimal performance by having to fetch the subject data, in addition to the disadvantage of the tenant having to disclose this data.

To further improve the performance of federated authorization, we introduce a technique called *policy federation*. In the process of policy federation, the policies of a tenant are decomposed and distributed over the tenant and the provider so that the resulting parts are evaluated near the data they require as much as possible while keeping sensitive tenant data and access rules local to the tenant premises.

This chapter provides a policy federation algorithm for attribute-based tree-structured policies similar to XACML, describes the required supporting middleware and evaluates the impact of policy federation on performance based on the policies of a hospital in the case study of the home patient monitoring application and a prototype of the middleware. As this chapter shows, policy federation effectively succeeds in keeping the sensitive tenant data confidential and at the same time improves policy evaluation time in most cases.

While we here describe and evaluate policy federation for the federated set-up between a tenant and a SaaS provider, this technique can effectively be applied to any policy evaluation system in which the data required to evaluate a policy is spread across multiple locations. As such, the work in this chapter could also be used to automatically and optimally deploy the Amusa policy tree across its multi-tier policy evaluation engine (see Chapter 3).

The rest of this chapter is structured as follows. Section 5.2 zooms in on the case study of the patient monitoring application and presents an extensive policy from this case study to be used in the rest of this chapter. Section 5.3 defines the attribute-based policy model and Section 5.4 the policy federation algorithm. Section 5.5 evaluates policy federation in terms of performance and thereby elaborates on the design of supporting middleware. Section 5.6 provides a discussion of policy federation. Section 5.7 covers related work and Section 5.8 concludes this chapter.

## 5.2 Case study analysis: home patient monitoring

This chapter builds upon the same case study as the previous chapter, i.e., an application for monitoring patients of cardiovascular diseases at their homes, provided to hospitals as a service. For a full description of this case study, we refer to Section 4.2.1; in this section, we only provide a summary of the case study and then zoom in on the policies of a hospital in this case study for use throughout this chapter.

### 5.2.1 Summary of the case study

Again we refer to Section 4.2.1 for a full description of the patient monitoring application. To summarize here, the application in this case study monitors patients of cardiovascular diseases after leaving the hospital. The application employs the measurements to provide a status overview of a patient to the physicians and nurses at his or her hospital. In addition, the designated physician of a patient is also notified in case of important evolutions. In addition, the application provides functionality such as patient questionnaires and shared notes on a patient overview. In this application, the hospitals are the tenants and they each manage multiple end-users, i.e., the patients, physicians and nurses. Next to the monitoring application, the hospitals also employ 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 patient monitoring application and of these security requirements, this chapter focuses on the sub-domain of access control.

### 5.2.2 Access control policies from the case study

The hospital's access control policies that apply to the monitoring application 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.

#### Structure of the hospital's policies

Similar to the previous chapters, this work builds upon attribute-based access control, which structures policies by making the distinction between the subject, the resource, the action and the environment. We apply the same structure in this discussion.

**Resources and actions.** The resources of the hospital's policies and the actions they support are determined by the structure of the data in the monitoring application. 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 resources are as follows: The raw measurements, the patient's status overview and the notifications are all created by the application 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 monitoring application as a whole.

**Subjects.** The subjects of the hospital's 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's policies provides the current time and date.

### Detailed policies

Following the general structure of the hospital's policies, this section illustrates one of these in detail by zooming in on the access rules 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 access rules in the case study. Other actions are constrained by similar rules.

We start from broad organization-wide access rules and end with specific rules 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 of these policies on-line[1].

**Organization-wide access rules.** The following organization-wide access rules of the hospital also apply to the monitoring application:

$R_1$. 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.

**Access rules about the monitoring application as a whole.** The following access rules of the hospital apply to the monitoring application as a whole:

$R_2$. Only physicians, nurses and patients can access the monitoring application.

$R_3$. 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 monitoring application.

$R_4$. Of the nurses, only nurses of the cardiology and the elder care department can access the monitoring application.

$R_5$. Nurses can only access the monitoring application during their shifts.

$R_6$. Nurses can only access the monitoring application from the hospital.

$R_7$. Of the nurses of the cardiology department, all nurses can access the monitoring application.

$R_8$. Of the nurses of the elder care department, only nurses who have explicitly been allowed to use the monitoring application can access the monitoring application.

**Access rules about viewing the status of a patient.** The following access rules of the hospital specifically apply to viewing the status of a patient:

$R_9$. 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, which is either triggered by the physician, triggered by a telemedicine operator or indicated by the monitoring data.

[1] http://people.cs.kuleuven.be/~maarten.decat/jisa2013/

$R_{10}$. 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 monitoring application.

$R_{11}$. Head physicians of the cardiology department can view the status of any patient in the monitoring application.

$R_{12}$. Standard physicians of the cardiology department can only view the status of any patient treated by themselves or by a physician in their team.

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

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

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

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

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

$R_{18}$. A patient can only access the monitoring application if (still) explicitly allowed by the hospital.

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

### Analysis

In terms of attribute-based access control, these 19 access rules 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 a resource, the current date etc. We provide an extensive overview of the required attributes of these policies on-line[1], but in summary, 19 of these attributes 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 a resource) 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 attributes required to reach a decision for a single request ranges from 4 to 13 with a mean of 7.65. In summary, this case study illustrates that the policies of a tenant for a SaaS application require attributes from both the tenant and the provider. This leads to sub-optimal performance, which is the focus of this work.

### 5.2.3 Problem statement and solution

As discussed in the previous chapter, the hospital's access control policies would be evaluated by the provider in traditional SaaS applications. This has multiple disadvantages and most importantly for this chapter, it forces the hospital to disclose all required attributes for evaluating its policies to the provider. These attributes include sensitive attributes that the hospital does not want to disclose for reasons of limited trust or even cannot share by law.

A solution to this problem is to have the hospital evaluate its policies itself, at its own premises. To achieve this, we investigated the technique of federated authorization in the previous chapter. Because 19 of the 30 different attributes required by the hospital policies are hosted by the hospital itself, this approach can also improve performance as it effectively brings policy evaluation to the data that it requires instead of the other way around. However, the policies of the tenant also reason about the resources in the SaaS application and therefore require data located at the SaaS provider. As a result, completely evaluating the policies of the tenant at the premises of this tenant leads to sub-optimal performance by having to fetch this data.

In this chapter we introduce the technique of *policy federation*, which aims to further improve the performance of federated authorization. More precisely, the process of policy federation decomposes and distributes the hospital's policies over the SaaS provider and the hospital based on the location and sensitivity of the attributes and parts of the policies. The goal of this process is to evaluate every part of the policies near the data it requires as much as possible while still keeping sensitive tenant policies or attributes confidential. For example, if the hospital evaluates whether a user has treated the owner of the status overview in the last two months ($R_{10}$), this data remains confidential. Similarly, if the hospital evaluates whether a user is a general practitioner ($R_3$), this data does not have to be fetched by the provider and if the provider evaluates whether the monitoring data indicates an emergency ($R_9$), this data does not have to be fetched by the tenant.

The complete solution presented in this chapter 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.

## 5.3 Policy model

This section defines a simple policy model that enables us to reason about policy federation. For similar reasons as the previous chapters, this chapter builds upon attribute-based access control and we employ policy trees similar to XACML (see Section 2.3.1). More precisely, the policy model employed in this chapter encompasses the core features of XACML, although we do not take obligations into account in this chapter. This minimal subset therefore resembles the features of the STAPL policy language and 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 [74]. With respect to these, the model presented in this work focuses on the aspects related to policy federation, i.e., the general structure of a policy tree and how a policy tree is evaluated.

### 5.3.1 Structure of a policy tree

The policy model used in this work represents policies using the concept of a policy tree. Similar to STAPL, the leafs of the policy tree are called *Rules*, the intermediary nodes are called *Policies*. To avoid confusion with the general terms "rules" and "policies", we use capitals to denote the elements of a policy tree.

**Rules.** Rules state in which conditions a certain request is permitted and in which it is denied. They therefore consist of an effect and a condition. The effect of a Rule is either Permit or Deny, respectively permitting or denying the request. The condition determines whether the effect holds or not: if the condition evaluates to `true`, the result of the Rule is its effect; if the condition evaluates to `false`, the result of the Rule is NotApplicable. Thus, the result of evaluating a Rule is either Permit, Deny or NotApplicable.

Because this work builds upon ABAC the conditions are expressions on the attributes of the subject (*s*), the resource (*r*), 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. The top-level function of a condition should result into a boolean, but expressions can internally also contain other types of functions, such as numeric additions or string concatenation.

Using the notation $Rule$=<*Effect, Condition*>, access rule $R_2$ as defined in Section 5.2.2 can be represented as follows:

$R_2$ = *<Deny, “physician” not in s.roles & “nurse” not in s.roles & “patients” not in s.roles >*

**Policies.** Policies combine the results of several children, either Rules or other Policies. They therefore consist of an ordered list of children, a combination algorithm and a target. The combination algorithm combines the results of the children into the result of the Policy. In this work, we limit ourselves to three combination algorithms that are present in XACML and suffice to express the policies from the case study: PermitOverrides, DenyOverrides and FirstApplicable. Appendix C explains the behavior of these algorithms in detail. The target is an attribute-based expression that determines whether the Policy and its children apply to the request or not. As such, if the target evaluates to `true`, the children of the Policy are evaluated and the result of the Policy is determined by the combination of the results of these children; if the target evaluates to `false`, the children are not evaluated and the result of the Policy is NotApplicable.

Notice that policy evaluation in general requires a single result, i.e., the access control decision. Since every set of policies can be combined to a single combined policy tree using the combination algorithms, we assume the policy tree to have a single root, which applies to all requests. For simplicity, we also assume the top element of a policy tree to be a Policy, i.e., not a Rule.

Using the notation $Policy$ = *<Target, CombinationAlgorithm, Children>*, the example rules of Section 5.2.2 can be combined into a single policy tree as follows (illustrated in Figure 5.1):

$P_0$ = *<true, FirstApplicable, [*$R_1$*,* $R_2$*, <“physician” in s.roles, DenyOverrides, [*$R_3$*,* $R_9$*, ...,* $R_{14}$*]>, <“nurse” in s.roles, DenyOverrides, [*$R_4$*, ...]>, ...>*

**Sensitive elements.** In the model, two elements of a policy tree can be declared sensitive: (i) the attributes used in a Rule or Policy and (ii) the Rules or Policies themselves. For Policies, sensitivity applies to the whole policy tree below it. In practice, these sensitivity labels can either 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 tree, using a separate meta-policy provides the advantage of central management. Elements of a policy tree 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[1].

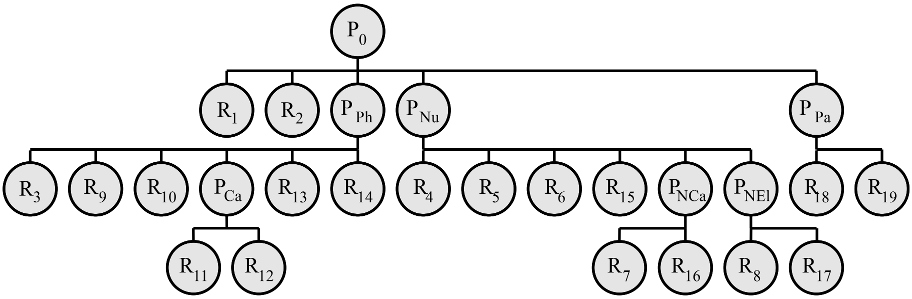


Figure 5.1: Representation of the example policies of Section 5.2.2 as a policy tree using our policy model. The intermediate policies bundle the rules for a certain type of subjects, e.g., $P_{Ph}$ contains the rules that apply to physicians, $P_{Ca}$ those that apply to physicians of the cardiology department and $P_{NEl}$ those that apply to nurses of the elder care department.

### 5.3.2 Evaluation of a policy tree

How a policy tree is evaluated also affects policy federation. We here define two aspects: (i) the order in which the elements of the tree are evaluated and (ii) how attributes are fetched.

**Evaluation order.** A Rule is evaluated by simply evaluating its condition. A Policy on the other hand 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 children are evaluated in the given order and their results are combined into the result of the Policy. As a consequence of evaluating the children in order, the policy tree is evaluated depth-first. For now, we assume that all children and expressions are evaluated sequentially.

**Fetching attributes.** During the evaluation of a policy tree, the required attributes are fetched from their respective Policy Information Points (see the reference architecture for policy-based access control systems discussed in Section 2.3.2). Because the required attributes for evaluating a part of a policy tree depend on the values of former attributes, it is generally impossible to determine the set of required attributes up-front and we assume that an attribute is only fetched when it is required. To enable this, the identifiers of the subject and the resource 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 tree for a single request. This caching avoids unnecessary attribute fetches and is required 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.

## 5.4 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.

### 5.4.1 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 [87]. The latency of a remote attribute fetch between tenant and provider is of 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 required between the tenant and the provider to reach an access decision.

An important design decision is the granularity of the policy distribution. In theory, even the individual clauses in the expression of a target or a condition could be distributed. However, we deliberately limit the granularity to complete Rules or Policies in the policy tree. As such, the decomposed policy tree remains compatible with existing policy systems and the existing 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 Rules and Policies into an equivalent set of smaller Rules and Policies that can then be separately deployed. Afterward, the algorithm tries to recombine multiple remote policy references into a single reference 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 tree $P$ to be federated, annotated with sensitivity labels in the tree and (ii) the list of attributes, each having a location and

**Algorithm 1** Overview of the policy federation algorithm. The methods `normalize()`, `decompose()` and `combine()` are defined in Algorithms 2, 3 and 4. AbstractPolicy is a type that represents an abstract element in the policy tree, i.e., a Rule or a Policy.

**Inputs:** $P$: a policy tree, 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 sub-tree at the root of the new policy tree that is deployed provider-side and can reference tenant-side policies, $S_P$: the set of referenced sub-trees to be deployed provider-side, $S_T$: the set of referenced sub-trees 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 AbstractPolicy p in $S_T$: $S_T$.replace(p, combine(p))
for AbstractPolicy p in $S_P$: $S_P$.replace(p, combine(p))
```

sensitivity label. The location of an attribute is either tenant-side or provider-side; the sensitivity label of an attribute, Policy or Rule is a boolean that determines whether the attribute, Policy or Rule can be shared with the provider or not. The algorithm provides three outputs: (i) $root$: the sub-tree at the root of the new policy tree which is deployed provider-side can reference tenant-side policies, (ii) $S_P$: the set of referenced sub-trees to be deployed provider-side and (iii) $S_T$: the set of referenced sub-trees to be deployed tenant-side. As we will see, several policy transformations are applied to the policy tree throughout the algorithm. Figure 5.2 lists these transformations and Appendix C proves their correctness by means of truth tables.

As shown in Algorithm 1, the algorithm consists of three major steps: normalization, decomposition and combination. In the next sections, we go into detail about each of these steps.

### 5.4.2 Step 1: Normalization

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

$$< T_1|T_2, CA, [P_1...P_n] > \Leftrightarrow < true, FirstApplicable, [< T_1, CA, [P_1...P_n] >, < T_2, CA, [P_1...P_n] >] > \tag{T1}$$

$$< Permit, C_1|C_2 > \Leftrightarrow < true, PermitOverrides, [< Permit, C_1 >, < Permit, C_2 >] > \tag{T2}$$

$$< Deny, C_1|C_2 > \Leftrightarrow < true, DenyOverrides, [< Deny, C_1 >, < Deny, C_2 >] > \tag{T3}$$

$$< T, PermitOverrides, [P_1, P_2, P_3] > \Leftrightarrow < T, PermitOverrides, [< true, PermitOverrides, [P_1, P_2] >, P_3] > \tag{T4}$$

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

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

$$< T, PermitOverrides, [P_1, P_2] > \Leftrightarrow < T, PermitOverrides, [P_2, P_1] > \tag{T7}$$

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

Figure 5.2: Policy transformations used in the policy federation algorithm. T1, T2 and T3 allow Policies and Rules to be split in an equivalent set of smaller elements and vice versa; T4, T5 and T6 allow Policies with more than two children to be expanded into binary trees or vice versa; T7 and T8 show the commutativity of PermitOverrides and DenyOverrides. Appendix C proves the correctness of these transformations by means of their truth tables. While these transformations are here shown for 2 or 3 elements, each can be generalized to $N$ elements.

step of the federation algorithm iteratively applies transformations T1, T2 and T3 of Figure 5.2 to the top of the given policy tree until no more children can be transformed, as shown in Algorithm 2.

Notice that transformations T1 to T3 only utilize `or` statements. The reason for this is that we want to remain compatible to XACML and only employ FirstApplicable, PermitOverrides and DenyOverrides, while converting an `and` statement would require other combination algorithms. For example, the equivalent of T1 for an `and` statement would require the combination algorithm BothApplicable.

**Algorithm 2** Definition of the `normalize()` method. AbstractPolicy is a type that represents an abstract element in the policy tree, i.e., a Rule or a Policy.

```
def normalize(AbstractPolicy p):
    AbstractPolicy p' = p.applyTransformations( [T1, T2, T3] )
    if p' != p:
        // a transformation was applied
        return normalize(p')
    else:
        if p is Rule:
            return p
        else: // Policy
            for Child child in p.children:
                p.children.replace(child, normalize(child))
            return p
```

**Results from the case study.** When applying the federation algorithm to the policy tree from the case study (see Figure 5.1), the algorithm will only apply T2. The reason for this is that the targets of the Policies in the tree only consist of exactly 1 element, e.g., *"physician" in s.roles* or *s.department == "elder care"* and rules are formulated as positive rules, i.e., permitting rules instead of denying ones. As an example, $R_{12}$ is split into two parts using T2:

*<Permit, o.owner_id in s.treated | o.owner in s.treated_by_team>*

$\Updownarrow_{T2}$

*<true, PermitOverrides, [<Permit, o.owner_id in s.treated>, <Permit, o.owner_id in s.treated_by_team>]>*

Similarly, $R_9$ is split into three times three parts because of the disjunctive normal form of two conjunctives with each three elements, $R_{10}$ is split in four parts and $R_{13}$ in two parts. In total, the policy tree of Figure 5.1 is transformed in the policy tree of Figure 5.3.

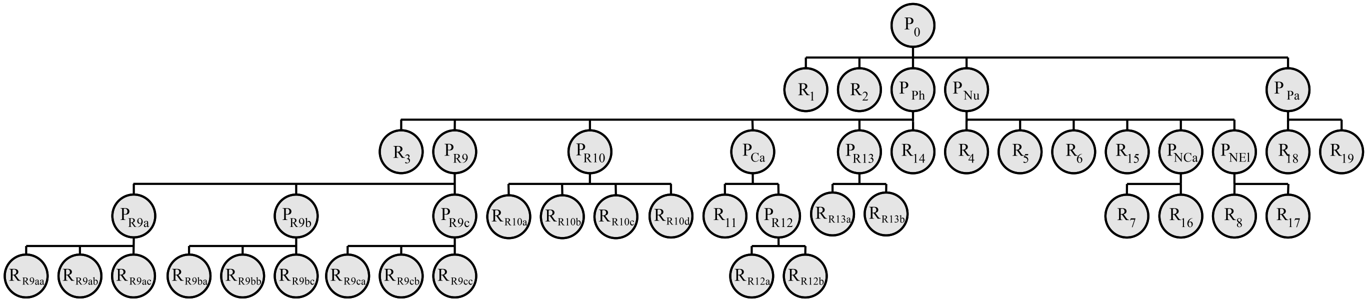


Figure 5.3: The result of normalizing $P_0$ illustrated in Figure 5.1.

**Algorithm 3** Definition of the `decompose()` method. $C_{i,P}$, $C_{i,T}$ and $C_{PR}$ are as defined in Section 5.4, $S_T$ and $S_P$ are as defined in Algorithm 1. AbstractPolicy is a type that represents an abstract element in the policy tree, i.e., a Rule or a Policy.

```
def decompose(AbstractPolicy p, Side parentSide):
    if p is Policy:
        for AbstractPolicy child in p.children:
            p.children.replace(child, decompose(child))
    ($C_{i,P}$, $C_{i,T}$) = 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
```

### 5.4.3 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 child of a Policy on the same side as the Policy is larger than the cost of evaluating it on the other side plus the cost of making a remote policy evaluation request, the child 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 of the policy tree, which should always be deployed provider-side. For a Policy or Rule 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.

**Cost functions for Rules.** For Rules in the policy tree, the cost functions are as follows:

$$C_{Rule,P} = N_{A,P} * C_L + N_{A,T} * C_R \tag{CF1}$$

$$C_{Rule,T} = N_{A,T} * C_L + N_{A,P} * C_R \tag{CF2}$$

The cost functions determine the cost of the provider ($C_{Rule,P}$) and the tenant ($C_{Rule,T}$) evaluating a certain Rule 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 less than $C_R$.

An important detail is the handling of cached attributes (see Section 5.3.2). 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 elements 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 conditions of previously evaluated Rules and targets of previously evaluated Policies. For these, we take into account super-Policies, previous Rules and Policies on the same level and previous Rules and Policies on the same level as super-Policies. In case a Policy has a target that matches all requests, the required attributes of the first child are taken into account as well. 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 a Rule or Policy, while some attributes may not be needed every time, e.g., the attributes required by the second clause in a conjunction when the first clause returns `false`.

**Cost functions for Policies.** For the Policies in a policy tree, the cost functions are as follows:

$$C_{Policy,P} = N_{A,P} * C_L + N_{A,T} * C_R + \sum K_{i,P} \tag{CF3}$$

$$C_{Policy,T} = N_{A,T} * C_L + N_{A,P} * C_R + \sum K_{i,T} \tag{CF4}$$

$N_{A,P}$, $N_{A,T}$, $C_L$ and $C_R$ are defined similarly as for Rules. Policies however only directly require attributes because of their targets and again, cached attributes are not taken into account. $K_{i,P}$ and $K_{i,T}$ represent the cost of evaluating the $i$'th child $Child_i$ of Policy $P$ in case $P$ is evaluated provider-side or tenant-side respectively. In case $Child_i$ is evaluated on the other side than $P$, 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 $Child_i$ when evaluating $P$ provider-side, thereby actually deciding on the optimal evaluation location of $Child_i$:

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

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

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

In summary, $C_{i,P}$ and $C_{i,T}$ are defined as CF1 and CF2 for Rules; for Policies, $C_{i,P}$ and $C_{i,T}$ are defined recursively as CF3 or CF4.

**Results from the case study.** The elements of $P_0$ all require more tenant attributes than provider attributes, except for the sub-tree resulting from normalizing $R_9$. As a result, most of the policy tree will be deployed tenant-side, starting from the root and only the sub-tree resulting from normalizing $R_9$ is still deployed provider-side. Because the root Policy $P_0$ is deployed tenant-side, a provider-side policy reference is inserted as the new root. In total, the policy tree of Figure 5.3 is transformed in the policy tree of Figure 5.4.

### 5.4.4 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

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

```
def combine(AbstractPolicy p):
    if p is Rule:
        return p
    else:
        AbstractPolicy[ ][ ] groups = p.getCombinableChildren()
        for AbstractPolicy[ ] group in groups:
            Policy cp = new Policy(p.target, p.combinationAlgorithm, 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.children.replace(group, new RemotePolicyReference(cp)
        return p
```

provider (see Algorithm 4). More precisely, the algorithm combines multiple Policies and/or Rules referenced in a single Policy into a larger equivalent Policy and combines their remote policy references into a reference to that new Policy. For this, the algorithm employs transformations T4, T5 and T6 as defined in Figure 5.2. In case of FirstApplicable, only consecutive remote policy references in the children of a Policy 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 T7 and T8.

**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. As such, the final policy tree is shown in Figure 5.4.

### 5.4.5 Discussion: policy equivalence

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

**Definition: Policy equivalence.** Two policy trees $P_1$ 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 resource, the action and the environment: $Ctx = (A_S, A_O, A_A, A_E)$. The request $R$ is a subset of the context: $R \subset Ctx$.

The policy federation algorithm maintains policy equivalence because (1) only step 1 and step 3 transform the policy tree and every applied transformation (see Figure 5.2) maintains policy equivalence and (2) both the original policy tree and the federated policy tree 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.

## 5.5 Performance evaluation

In this section, we evaluate policy federation in terms of performance.

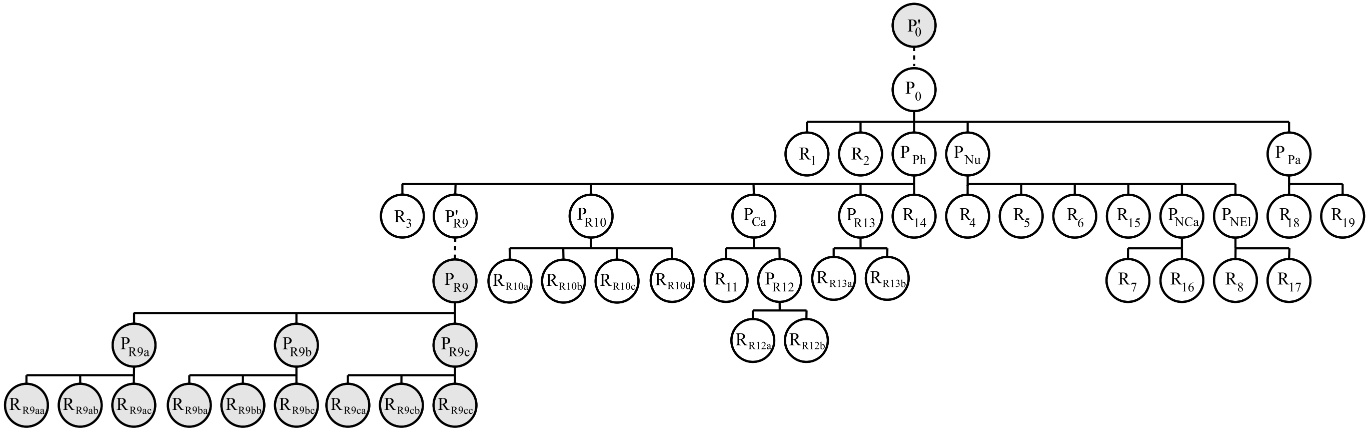


Figure 5.4: The result of decomposing the normalized version of $P_0$ illustrated in Figure 5.3. Grey elements are deployed provider-side, white elements are deployed tenant-side. Elements with a prime symbol are references to a remote policy.

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 11ms; for policies of one order of magnitude larger[2], the algorithm still takes less than 2 seconds. Because these durations fit the asynchronous usage of the federation algorithm, we do not provide more details about this and focus on the impact of policy federation on policy evaluation time.

### 5.5.1 Middleware 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 [7]. The source code is publicly available on-line[1].

Figure 5.5 shows the architecture of the supporting middleware in terms of the XACML reference architecture for policy-based access control systems (see Section 2.3.2). As illustrated, this architecture is an extension of the architecture for federated authorization of the previous chapter (see Section 4.3.2). In the architecture of Figure 5.5, both the provider and the tenant will evaluate polices and therefore both have a Policy Administration Point (PAP), a Policy Decision Point (PDP), a Context Handler and one or more Policy Information Points (PIPs). The provider hosts the SaaS application and therefore also the PEP. The provider hosts the attributes concerning the resources in the application ($A_R$) and the provider part of the environment ($A_{E,P}$); 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). As a result, the interface between the provider and a tenant now consists of four services, compared to two for federated authorization as discussed in Chapter 4.

In the prototype, 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 5.5 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.

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

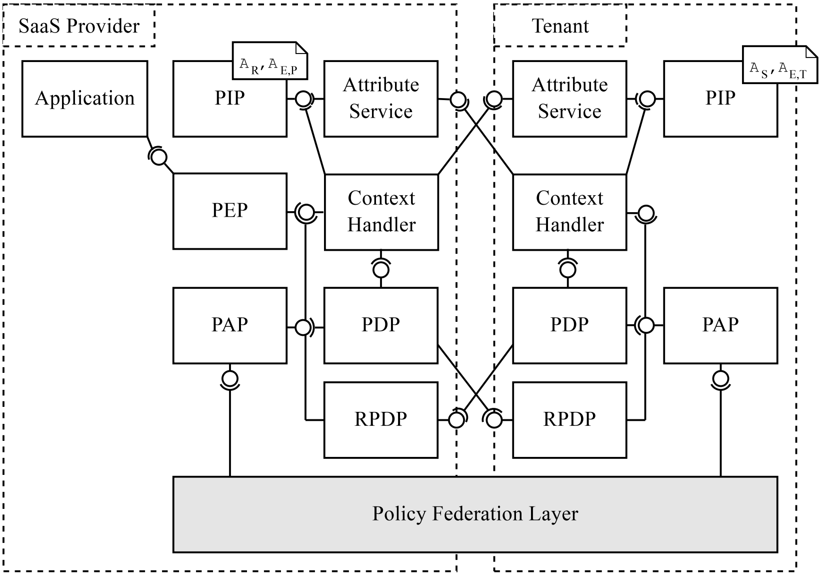


Figure 5.5: Architecture of the supporting middleware for policy federation in terms of the XACML reference architecture (see Section 2.3.2). The Policy Federation Layer is the focus of this work.

### 5.5.2 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 Section 5.2 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.

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

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 10ms to simulate the distance between these parties. In order to avoid overhead of parallelism, the tests are run sequentially and PDP evaluation is performed 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%.

### 5.5.3 Results

Figure 5.6 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 less remote requests

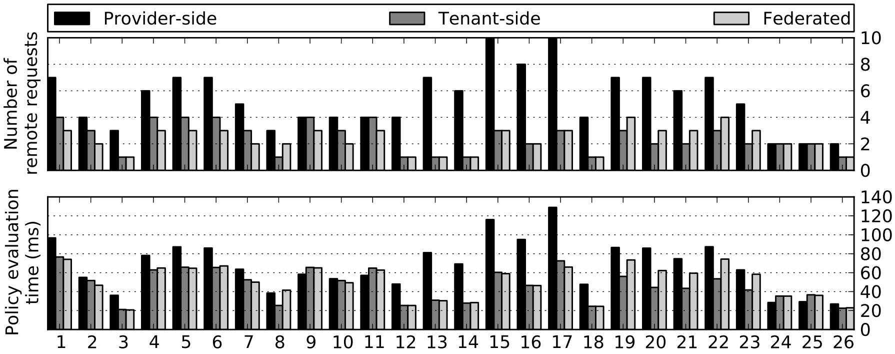


Figure 5.6: Results of the performance tests. The upper chart shows the number of remote requests needed for evaluating the policies for a certain request (lower is better), the lower chart shows the resulting policy evaluation time in milliseconds (lower is better). For each access request, we show 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.

than tenant-side evaluation. The same number is achieved if $R_9$, i.e., the sub-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., for 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 requests and longer evaluation times than tenant-side evaluation. This is caused by the fact that $R_9$ 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. This issue can further be optimized in the future.

## 5.6 Discussion

In the previous sections, we presented the technique of policy federation, which aims to decompose and deploy access control policies over multiple parties for improved performance and confidentiality. In this section, we discuss the results of this work and in which ways it can be refined and extended.

**Confidentiality.** Similar to federated authorization, policy federation keeps the sensitive tenant attributes and policies confidential by not sharing them with the SaaS provider. As discussed in the previous chapter however, a potential threat to this approach is the possible inference of policies or attributes by the provider based on the complete set of access requests and decisions. For this, 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 confidentiality model employed in the federation algorithm can be refined. The algorithm now assumes that an attribute or element in the policy tree is labeled sensitive or non-sensitive. In a more extensive case, a sensitivity meta-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.

**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 5.5.3. Second, the algorithm achieves sub-optimal results because it assumes that remote policy evaluations take equally long as remote attribute fetches, while a policy evaluation in general takes more time than an attribute fetch (amongst others because a policy evaluation can require multiple database accesses and an attribute fetch always requires one). While policy evaluation engines are expected to provide improved performance towards the future (e.g., [152]), the cost functions in the algorithm can be refined to take into account this difference. As a further extension, the performance properties of the infrastructures of the provider and the tenant 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.

**Obligations and attribute updates.** Another part of future work is to incorporate obligations, i.e., operations which should be performed in conjunction with enforcing the access control decision (see Section 2.2.4 for more information). For example, obligations can be used to specify that the user should agree to a license agreement or that the policy evaluation engine should write out a log, send an e-mail to an administrator or update an attribute value. 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.

An interesting subset of obligations are attribute updates. Attribute updates can be used to model history-based policies [110], 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 (i) attribute updates require concurrency control in case of distributed policy evaluation and (ii) history-based policies are known to have a large impact on performance [110]. Both are therefore interesting tracks for future research.

**Generalization to N>2 parties.** A final possible extension of this work is a generalization to more than two parties. This chapter 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 service provided to multiple hospitals which collaboratively provide care to the same patient or the collaboration platform discussed in the previous chapter. If this set-up reduces to each organization applying its specific policies to a shared application, the algorithm can separately be applied to the policies of each organization without change. For set-ups that do not show this pattern, e.g., 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 chapter to apply to this situation as well.

## 5.7 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 5.3. For an overview of this large body of work, we refer to [97].

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. [40] 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. In addition, Ardagna et al. [32] focus on controlled disclosure of sensitive access control policies to clients in the domain of web services. They therefore also discuss policy decomposition and transformation rules. However, while this work has definitely been an inspiration to ours, 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. [149] 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 which limits its 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. [34] employ attribute and policy encryption, extending the work of di Vimercati et al., e.g., [93]. 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 [58]. Policy federation can be complemented with the work of several other authors, e.g., Wei et al. [203], who focus on decision caching and Gheorghe et al. [113], who focus on infrastructure reconfiguration for optimal attribute retrieval and cross-request attribute caching.

## 5.8 Conclusion

This chapter introduced the technique of *policy federation*. Policy federation aims to improve the performance of federated authorization by decomposing and distributing the tenant-specific policies across tenant and provider so that each part of the policies is evaluated near the data it requires as much as possible while sensitive tenant data and policies are kept confidential. In this regard, we defined a simple yet generic attribute-based policy model, described a detailed algorithm for policy federation and elaborated on the design of supporting middleware. As our evaluation shows, policy federation succeeds in keeping the sensitive tenant data confidential and has the ability to improve policy evaluation time as well. While we presented policy federation in the context of federated authorization, this technique can effectively be applied to optimize the performance of any policy evaluation system in which the data required to evaluate a policy is spread, e.g., the collaboration platform presented in the previous chapter or the multi-tier policy evaluation engine of Amusa presented in Chapter 3. As such, this work has contributed to a growing collection of performance techniques for policy-based and attribute-based access control.

The next chapter returns focus from the tenants of a SaaS application to its provider. More precisely, the next chapter focuses on the challenge of applying policy-based access control to large-scale applications such as SaaS applications and supporting the large number of requests per second that these applications face.

# Chapter 6

# Concurrent evaluation of access control policies

This chapter introduces our fourth contribution: a scalable concurrency control scheme specifically for evaluating access control policies. The goal of this chapter is to enable policy-based access control for large-scale distributed applications such as most SaaS applications. When applying access control to such applications, the access control policies must be evaluated concurrently as well. However, for certain classes of policies such as history-based policies, concurrency can be exploited to achieve incorrect access decisions. Moreover, general techniques for concurrency control in databases are not able to scale to the size of SaaS applications and at the same time provide the full consistency required for security. Therefore, this chapter presents a concurrency control scheme specifically for access control. This scheme leverages the domain-specific structure of a policy evaluation to scale to a large number of machines while incurring only a limited and bounded latency overhead.

This chapter stems from the goal of enabling policy evaluation to securely scale out with low performance overhead. As such, this chapter focuses on the challenge of the large scale of a SaaS application and the concern of low performance overhead (see Section 1.2). This chapter is based on our publications at MW4NG 2013 [87] and at ACSAC 2015 [86].

## 6.1 Introduction

Most SaaS applications aim for a large amount of tenants. To be able to cope with the resulting amount of requests to the application, these SaaS applications are deployed on a distributed infrastructure. This way, multiple requests can be handled in parallel by different machines and the number of machines can be scaled up for achieving larger throughputs while handling each request with low latency.

Because access control should be enforced on most, if not any, request to an application but should not impede the correct use of this application, the access control system should be able to provide the same throughput, scalability and low latency. For policy-based access control, this means that the access control policies must also be evaluated concurrently and distributedly.

For certain policies however, concurrent evaluation can lead to incorrect access decisions if not performed properly. For example, take the rule "*if the user has had access to documents of Bank A, he or she is not allowed to access documents of Bank B*". This rule is taken from the eDocs case study (see Chapter 3) and is an example of the well-known and common class of *history-based policies* [56, 95, 110, 125, 153, 171, 169]. For these policies, one evaluation of the policy influences future evaluations, which leads to race conditions. As a result, if a user has not had access to documents of Bank A or Bank B, this user could exploit concurrency to have the policy permit two parallel requests, thus violating the policy. This problem affects the practical application of history-based policies in large-scale applications such as SaaS applications.

A possible solution to this problem would be to model a policy evaluation as a transaction on the database that contains the data utilized in the policy. In this case, policy evaluation requires *strong consistency* because it is a security measure and therefore does not allow a single incorrect decision. However, databases inherently cannot scale to the size of current applications and at the same time provide such strong consistency. Moreover, policy evaluation often requires data from multiple sources in practice. This would lead to distributed transactions, which do not perform well. As a result of all this, what we need is an efficient and scalable domain-specific scheme for concurrency control at the level of policy evaluation itself.

In this chapter, we present and evaluate such concurrency control scheme. More precisely, we first model history-based policies using attributes, policy trees and obligations and then present a concurrency control scheme for evaluating such policies. By leveraging the specific structure of a policy evaluation, this domain-specific scheme (i) is able to avoid incorrect access control decisions due to concurrency, (ii) is able to scale to a large number of multi-core machines and (iii) incurs only a limited and asymptotically bounded latency overhead.

The rest of this chapter is organized as follows. Section 6.2 elaborates on the need for concurrency and concurrency control in policy evaluation. Section 6.3 presents our scheme for concurrency control, both for a centralized and a distributed deployment. Section 6.4 verifies that this scheme is able to scale to a large number of machines by means of a prototype. Section 6.5 discusses related work and our approach. Section 6.6 concludes this chapter.

## 6.2 Problem elaboration

This section starts by discussing the need for concurrent and distributed policy evaluation and illustrates the need for concurrency control.

### 6.2.1 The need for concurrency and distribution

This chapter focuses on concurrent and distributed evaluation of access control policies. Our experiences in our case studies and related work (e.g., [31, 63, 130]) illustrate that these techniques are crucial to applying policy-based access control to realistic applications for several reasons. For example, distribution is inherent to some of these applications because multiple distributed parties have to collaborate (e.g., [149]). The most common reason for concurrency and distribution however is throughput: current applications such as SaaS applications are aimed for a large number of users, which results into large amounts of requests per second. In order to cope with this

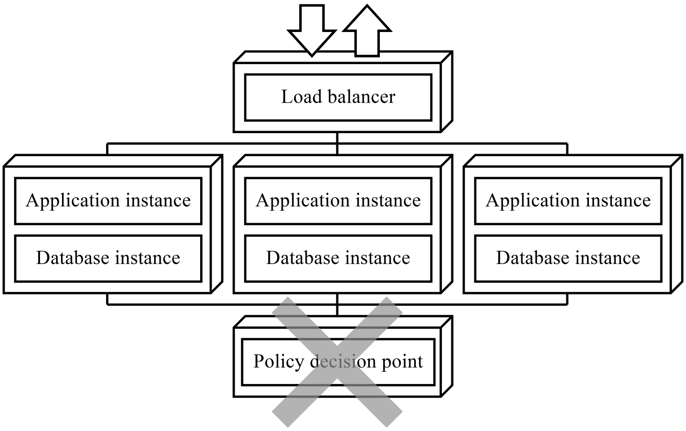


Figure 6.1: In order to support large throughputs, current applications are designed for and deployed on a scalable distributed infrastructure. In these cases, a central policy decision point is not able to scale with the application.

requirement, these applications are deployed on a distributed infrastructure so that multiple requests can be handled in parallel by different machines. In addition, these applications employ tactics in order to scale out horizontally without introducing large latency overheads, e.g., load balancing over multiple machines (see Figure 6.1).

Applying policy-based access control to these applications leads to equally large amounts of policy evaluations per second because access control should be enforced on every action performed in the application. Without concurrency or distribution, the only possibility is a central policy evaluation point that sequentially evaluates the policy for multiple requests and therefore is not able to scale with the application. In other words, realistic applications, and SaaS applications in particular, require to be able to evaluate a policy for multiple requests concurrently (i.e., in parallel) and distributedly (i.e., on multiple machines).

### 6.2.2 The need for concurrency control

While other authors have also focused on concurrent and distributed policy evaluation (e.g., [31, 63, 130]), one crucial issue has been overlooked for now: the need for concurrency control.

Concurrent or distributed policy evaluation is trivial in case evaluating the policies only requires to read data, i.e., the policies do not update data. This is the case for policies of models such as lattice-based access control and role-based access control (see Section 2.2.3). As a result, such policies are already being applied in distributed applications without any issues. There is however an important class of policies that does update data as a result of evaluating the policy: *history-based policies* (sometimes also called *stateful policies* [125]). A well-known instance of such policies also present in our case studies are dynamic separation-of-duty policies or Chinese wall policies [56], for example:

$P_1$*: If a user has had access to documents of Bank A, he or she is not allowed to access documents of Bank B.*

Another common class of history-based policies are policies that put upper limits on the amounts of resources consumed by subjects, for example:

$P_2$*: A user cannot view more than 10 movies per month.*

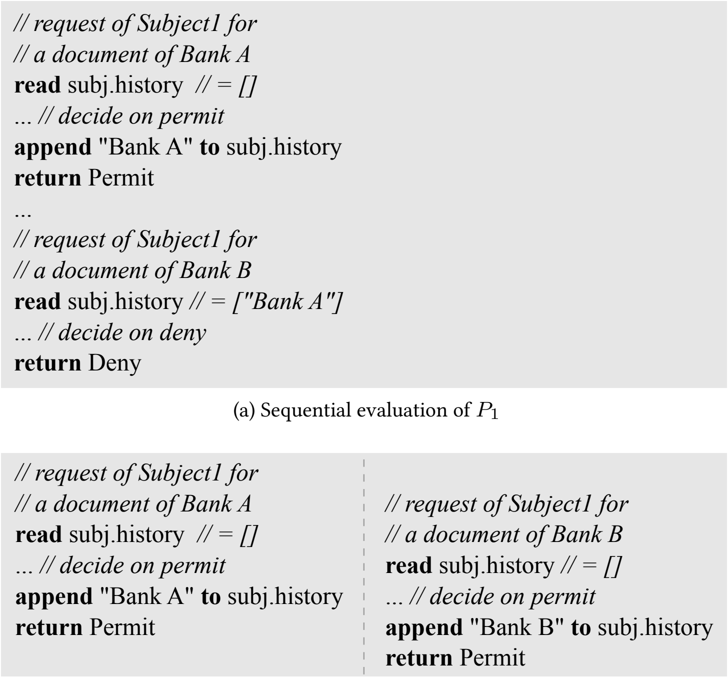
Or similar policies from the point of view of the resources:

*$P_3$: An article cannot be shared more than 5 times.*

Other authors mention similar examples (e.g., [56, 62, 95, 110, 111, 130, 153, 169, 171]).

These policies are called history-based policies because one access decision depends on the previous decisions, i.e., the *history.* For example in $P_1$, permitting the subject to access a document of Bank A influences the policy evaluations for the same subject and documents of Bank B. In this case, the sequential evaluation of the policy for a document of Bank A and a document of Bank B for the same subject would lead to a permit and a deny (as illustrated in Figure 6.2a).

However, because these policies both read and update the history, *race conditions* can occur when evaluating such policies concurrently. For example, Figure 6.2b illustrates the concurrent version of Figure 6.2a. In this case, both policy evaluations initially see an empty history of the subject, leading to the incorrect result of permitting both



```
// request of Subject1 for
// a document of Bank A
read subj.history  // = []
... // decide on permit
append "Bank A" to subj.history
return Permit
...
// request of Subject1 for
// a document of Bank B
read subj.history // = ["Bank A"]
... // decide on deny
return Deny
```

(a) Sequential evaluation of $P_1$

```
// request of Subject1 for          |
// a document of Bank A             | // request of Subject1 for
read subj.history  // = []          | // a document of Bank B
... // decide on permit             | read subj.history // = []
append "Bank A" to subj.history     | ... // decide on permit
return Permit                       | append "Bank B" to subj.history
                                    | return Permit
```

(b) Concurrent evaluation of $P_1$

Figure 6.2: Sequential versus concurrent evaluation of history-based policy $P_1$ of Section 6.2.2. As illustrated, concurrently evaluating a history-based policy can lead to incorrect access decisions if not performed properly.

requests. Similarly, a concurrent evaluation of $P_2$ and $P_3$ could respectively permit a user to view more than 10 movies in one month and permit a document to be shared more than 5 times.

These examples are instances of a well-known problem in distributed systems called a *read-write conflict* and they illustrate that race conditions in policy evaluation can lead to security holes that can be actively exploited by an attacker. This problem affects any distributed or concurrent enforcement of history-based policies, both in practice and in literature. In order to prevent these race conditions from leading to incorrect decisions, we require a form of *concurrency control* that ensures that concurrent policy evaluations provide the same decisions as sequential evaluations, a property called *serial equivalency*.

### 6.2.3 The need for concurrency control at the level of policy evaluation

In order to address the race conditions when evaluating history-based policies, a form of concurrency control is required. A straightforward approach to achieve this would be to model these evaluations as transactions on an underlying database. However, there are two reasons why this approach does not work.

Firstly, our experience shows that the data required for evaluating a policy often originates from multiple sources, such as a directory service of user data and a database of application data. Concurrency control in this situation would require distributed transactions over multiple technologies, which does not perform well because of the network and marshaling overhead of the negotiations between all involved attribute sources.

Secondly, even if all access control data is available in a single database, policy evaluation is a security technique that does not tolerate a single incorrect decision and therefore requires *strong consistency*. Strong consistency means that all reads and updates are seen by all database instances in the same sequence. However, because strong consistency requires to negotiate every write with every replica, databases (or other types of data storage) are inherently not able to provide both strong consistency and scalability. For example, the relational database MySQL[1] does provide strong consistency, but limits read-write transactions (which are required to model the evaluation of a history-based policy) to a single master server. More recent “NoSQL” databases such as Cassandra[1] or MongoDB[1] on the other hand were designed for scalability, but therefore limit transactions to compare-and-set operations on a single element in the database (i.e., a table row or a document). A history-based policy however can update data of both a subject and a resource (e.g., a combination of $P_1$

[1] https://www.mysql.com/, http://cassandra.apache.org/, https://www.mongodb.org/

and $P_3$) and a single subject can access multiple resources, which makes it impossible to segregate the data of a single policy evaluation in separate database elements. As a result, these databases also do not provide the necessary concurrency control for policy evaluation.

In conclusion, we argue that we require an efficient and scalable scheme for concurrency control at the level of policy evaluation itself.

### 6.2.4 Requirements for concurrency control

As a result of the previous discussion, the goal of this work is to design a scheme for concurrency control for policy evaluation. This scheme (i) should prevent incorrect access decisions because of race conditions, (ii) should provide low latency overhead and (iii) should be able to scale out to large numbers of machines in terms of throughput. As we will see, we achieve these properties by leveraging the domain-specific structure of a policy evaluation.

Notice that we focus on concurrency and distribution between multiple policy evaluations for throughput and deliberately leave concurrency within a single policy evaluation (e.g., as described in [149, 194]) out of scope because this does not add extra complexity for concurrency control.

## 6.3 Concurrency control

The goal of this work is to design a scalable and efficient concurrency control scheme for the evaluation of history-based access control policies. In this section, we first describe how we model history-based policies in current attribute-based tree-structured policy languages such as XACML and STAPL, then describe possible approaches to concurrency control and finally describe our resulting scheme for concurrency control.

### 6.3.1 Modeling history-based policies in current policy languages

In this work, we represent policies in the model of XACML and STAPL. As explained in Section 2.3.1, this model combines three interesting techniques: attribute-based access control (ABAC), policy trees and obligations. Firstly, ABAC is able to express a wide variety of well-known access control concepts and effectively generalizes the previous models of identity-based access control, lattice-based access control and role-based access control [127]. Secondly, policy trees enable structuring multiple

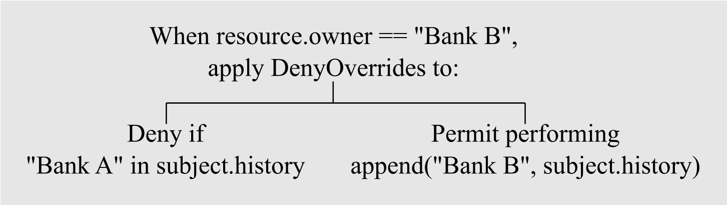

When resource.owner == "Bank B",
apply DenyOverrides to:
Deny if
"Bank A" in subject.history
Permit performing
append("Bank B", subject.history)

Figure 6.3: Representation of $P_1$ of Section 6.2.2 as an attribute-based policy tree with obligations, similar to XACML.

rules into one well-defined policy and reason about possible conflicts between these rules. Thirdly, obligations define operations that should be performed when enforcing an access decision, such as “log this event” or “notify a security administrator that access is denied to a confidential document”.

History-based policies can be represented in XACML and STAPL by modeling the history as an attribute that is updated as a side-effect of policy evaluation, an approach also advocated by Park and Sandhu [171]. For example, in $P_1$, that attribute would be `subject.history`, which contains the list of companies of which the subject has accessed documents. The update of this attribute can then be represented in XACML as an obligation[2], an approach also taken by Chadwick [62]. Figure 6.3 illustrates the resulting representation of $P_1$ as an attribute-based policy tree with obligations. Interestingly, because obligations are always defined to only hold for a certain decision, they are only enforced after the final decision is known, i.e., after the policy has been evaluated entirely [5]. As a result, all attribute updates are performed after the last attribute read (see Figure 6.4), a property that we build upon later on.

As a side-remark, notice that the access control system performs the history updates in our approach, but that one could argue that the history should be updated by the application itself or that the history updates should be performed implicitly, for example using logs or provenance data [169]. In order to be able to perform concurrency control on the level of policy evaluation however, we argue that the attribute updates are best performed by the access control system itself. The discussion in Section 6.5 goes deeper into this topic.

[2] While obligations can also be used to express other operations than attribute updates, e.g., writing a log, we only take into account attribute updates as obligations for the rest of this chapter for simplicity. This choice does not affect the validity of this work as attribute updates are the only type of obligation that require concurrency control.

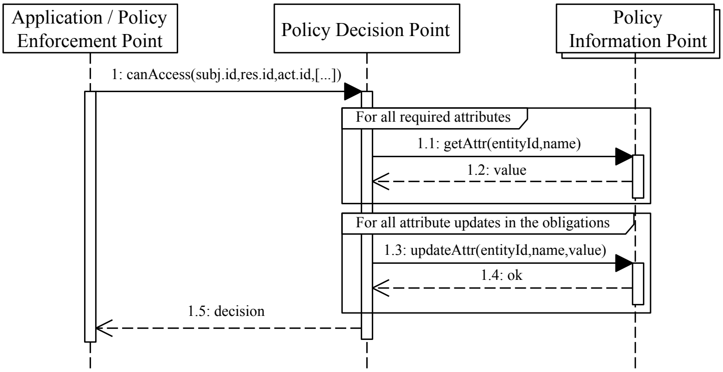

Application / Policy Enforcement Point
Policy Decision Point
Policy Information Point
1: canAccess(subj.id,res.id,act.id,[...])
For all required attributes
1.1: getAttr(entityId,name)
1.2: value
For all attribute updates in the obligations
1.3: updateAttr(entityId,name,value)
1.4: ok
1.5: decision

Figure 6.4: A black box representation of a policy evaluation by a policy decision point. As illustrated, all attribute updates are performed after the last attribute read [5], a property that we build upon in our approach.

## 6.3.2 Tactics for concurrency control

As explained, our goal is to design concurrency control that prevents incorrect concurrent access decisions by enforcing serial equivalence. In attribute-based policies such incorrect decisions result from an attribute update that is executed during another policy evaluation that has already read an older value of this attribute. Thus, our goal can be refined to detecting such conflicting pairs of attribute updates and reads in concurrent policy evaluations and preventing these from returning an incorrect decision to the application. Moreover, examples of history-based policies in literature (e.g., [56, 62, 95, 110, 111, 130, 153, 169, 171]) and our own case studies (see the previous chapters) only exhibit attribute updates for subjects and resources, i.e., not for the action and the environment. These subject and resource attributes are always assigned to a single subject or resource. As a result, our goal can further be refined to detecting and addressing pairs of conflicting attribute updates and reads for the same attribute of the same subject or resource. For example in Figure 6.2b, the *append* of the left evaluation and the *read* of the right evaluation make up such a pair.

**Three possible tactics.** From a high-level point of view, there are three tactics for concurrency control that can provide serial equivalence: locking, timestamp ordering and optimistic concurrency control (see for example [72]). With locking, every evaluation would try to lock an attribute when reading or updating it and block until the lock is granted. These locks are only granted if there are no conflicting locks yet (e.g., a read lock when trying to obtain a write lock). With timestamp ordering, every evaluation would be given a timestamp and every attribute read and update would be checked immediately to make sure that the attribute has only been read (updated) by earlier transaction upon an update (read). If not, the evaluation is restarted immediately. Finally, with optimistic concurrency control, conflicts with ongoing evaluations are only checked upon commit at the end of the evaluation, allowing the evaluations to read and update attributes as normally. If conflicts are detected, any already executed updates are rolled back and the evaluation is restarted together with any evaluations that have seen these updates.

**Selected tactic.** Of these three tactics, optimistic concurrency fits policy evaluation the best. In this approach, concurrency conflicts are only checked upon commit at the end of the evaluation. This allows the evaluations to read and update attributes as normally. If conflicts are detected, any already executed updates have to be rolled back and the evaluation has to be restarted together with any evaluations that have seen these updates, which leads to overhead. For policy evaluations however, the attribute updates are always executed after the last attribute read as illustrated in Figure 6.4. As a result, this process naturally allows to check for conflicts after all reads and before actually executing the updates (i.e., right before the *append*s in Figure 6.2b), which avoids having to roll back any update. In addition, we expect that situations such as a user accessing documents of both Bank A and Bank B in parallel rarely occur in non-malignant behavior. As such, the overhead of restarting evaluations is negligibly small in the whole space of non-malignant requests (and we do not care about the performance for malignant requests). Finally, optimistic concurrency control also has the advantage that it can be executed on a separate layer of concurrency control without having to rely on any database feature.

**Resulting high-level design.** As a result of our choice for optimistic concurrency control, we effectively split up the Policy Decision Point of Figure 6.1 into a system with three types of components: the worker, the attribute database and the coordinator. Firstly, one or more *workers* are responsible for evaluating the policy for different requests. To do so, these workers can all fetch attributes from one or more *attribute databases*, i.e., Policy Information Points, which store the attributes across multiple policy evaluations. Finally, the *coordinator* is responsible for performing concurrency control on the policy evaluations performed by the workers and is therefore placed in between the application and the workers. Notice that the interface to the Policy Decision Point remains the same from the point of view of the application or the Policy Enforcement Point.

In the next sections, we describe two instances of the coordinator: a centralized coordinator that enforces our concurrency control scheme on a single machine, and a distributed coordinator that is able to scale out itself.

### 6.3.3 Centralized coordinator

In this section, we describe our basic scheme for concurrency control and how it can be implemented in a centralized coordinator that manages multiple workers.

**Basic approach.** Our basic approach is that the coordinator receives every policy evaluation request before it is passed to a worker and every decision before it is passed back to the application. The coordinator then keeps track of which attributes were updated during each policy evaluation. More specifically, because a policy only reads and updates attributes of a single subject and resource, the coordinator maintains the list of attributes that were updated for the subject and resource of every ongoing policy evaluation. To achieve this, the workers return the list of attributes that were read in a policy evaluation and the application has to specify at least the id of the subject and the resource.

**Protocol.** Figure 6.5 illustrates the resulting protocol for concurrency control. When the application requires an authorization decision, it sends a policy evaluation request to the coordinator (step 1). The coordinator then assigns a unique identifier to this evaluation (step 2) and passes the request along to a worker[3] (step 3). The worker then evaluates the policy for this request (step 4) and returns its decision together with the list of attributes that should be updated and the list of attributes that were read (step 5). The coordinator then checks whether this evaluation has read any attribute that was updated during this evaluation (step 6). If not, the coordinator persists the attribute updates (step 7), adds these attributes to the lists of updated attributes of policy evaluations for the same subject or resource (step 8) and returns the decision to the application (step 9). In case the coordinator detects a conflict (e.g.,

[3] Our scheme deliberately leaves open the way in which the coordinator decides to which worker a request should be assigned, any technique for load balancing can be used.

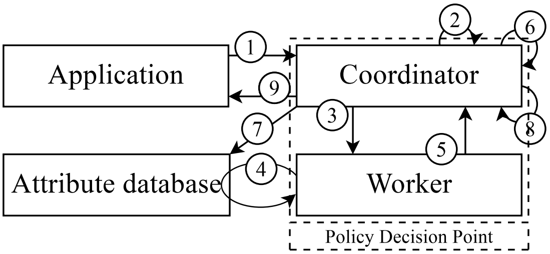


Figure 6.5: The protocol for concurrency control using the centralized coordinator, in case there is no conflict.

when trying to commit the right evaluation in Figure 6.2b), the coordinator restarts the policy evaluation by reassigning the original request to a worker after clearing its administration for this evaluation. Notice that in the reference architecture for policy-based access control systems (see Section 2.3.2) the Policy Enforcement Point enforces obligations. In our approach however, the coordinator, which is a part of the Policy Decision Point, executes the attribute updates in order to be able to reason about both attribute reads and updates for concurrency control.

**Possible deployments.** Figure 6.6 illustrates two possible deployments using the centralized coordinator. As illustrated, the centralized coordinator can manage multiple workers, but will have an upper limit to throughput as it cannot scale out itself. In order to address this limitation, the next section extends the centralized coordinator into a scalable distributed coordinator. Note that we deliberately do not show the attribute database in Figure 6.6 because our system does not rely on a specific deployment of this database, as will be explained in Section 6.3.5.

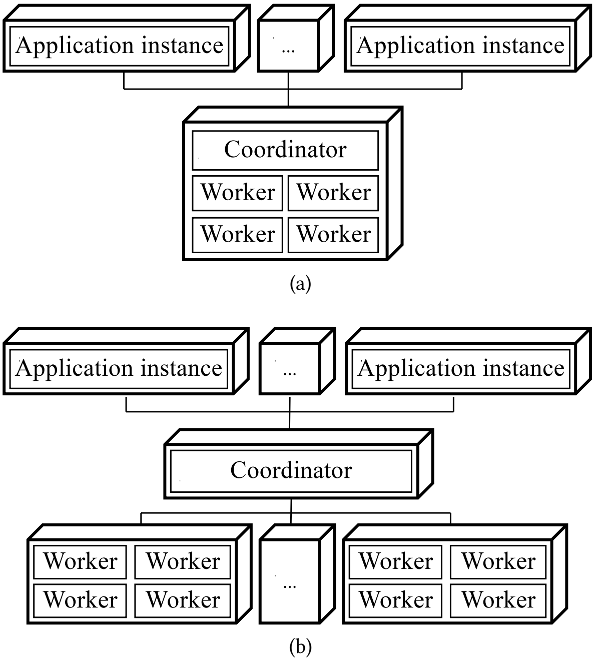


Figure 6.6: Two possible deployments using a centralized coordinator.

### 6.3.4 Distributed coordinator

In order to avoid that the coordinator becomes a bottleneck to the scalability of the system, it should be scalable as well.

**Approach.** In order to scale out the coordinator, we build on the observation that attribute updates only occur for subjects and resources, and that every policy evaluation has exactly 1 subject and 1 resource. As a result, the list of relevant attributes that were updated during an evaluation essentially consists of two parts: the updated attributes that belong to the subject of the evaluation and those that belong to its resource. Our approach to designing a scalable coordinator essentially distributes these two parts over two coordinators, one responsible for the subject of the evaluation and one for the resource. These two *distributed coordinators* collaborate for correct concurrency control.

The key to scaling out is that every request is handled by two distributed coordinators, but that these are part of an arbitrarily large pool of coordinators. More precisely, each subject and resource is assigned to exactly one responsible coordinator and this coordinator manages this subject/resource for all policy evaluations related to it. To achieve this, all application instances and distributed coordinators agree on the ordered list of coordinators in the system and the responsible coordinators are determined based on a hash of the id of the subject and resource in question. This hash should ensure that the subjects and resources are distributed uniformly across the pool of coordinators.

**Protocol.** Figure 6.7 illustrates the resulting collaboration protocol between the two distributed coordinators responsible for a single request. First, when the application requires an authorization decision, it determines the coordinator that is responsible for the subject in question based on the id of the subject and the list of coordinators (step 1). The application then sends a policy evaluation request to this coordinator (step 2). This coordinator assigns a globally unique id to this evaluation, sets up the administration for the subject, adds any tentatively updated attributes to the request (see step 9), determines the coordinator responsible for the resource in question (all step 3) and forwards the authorization request to that coordinator (step 4). That second coordinator sets up the administration for the resource (step 5) and assigns the request to a worker (step 6) which evaluates the policy for this request (step 7). After the evaluation, the worker sends the result to the coordinator responsible for the subject (step 8). This coordinator checks whether the evaluation employed subject attributes that were updated in the mean while. If not, it tentatively executes the updates of subject attributes (step 9). This means that future evaluations for this subject will see these updates, but that the coordinator will restart evaluations that

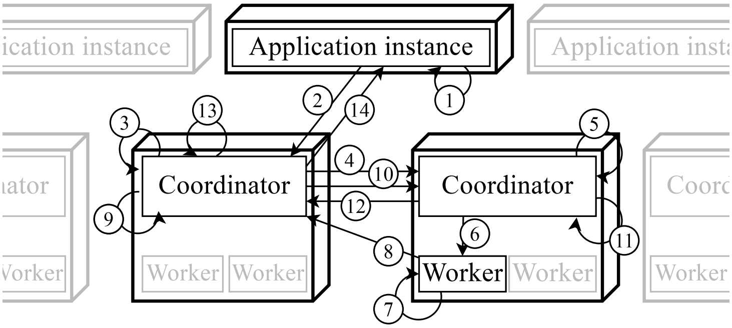


Figure 6.7: The protocol used for distributing concurrency control over two coordinators, in case there is no conflict.

have read these attributes if this evaluation could not commit entirely because of conflicts for resource attributes (see steps 11 and 12). The coordinator then asks the coordinator responsible for the resource whether there are conflicts for resource attributes (step 10). That coordinator checks its administration and if there are no conflicts, executes the updates of resource attributes, clears its administration for this evaluation (both step 11) and acknowledges success to the first coordinator (step 12). This coordinator then executes the tentative attribute updates, clears its administration for this evaluation (both step 13), and passes the decision along to the application (step 14).

In case conflicts were found, the protocol ends differently. Firstly, in case the coordinator responsible for the subject detects a conflicting update of a subject attribute, it does not tentatively execute the attribute updates, but immediately restarts the evaluation by clearing its administration for this evaluation and resending the initial request to the coordinator responsible for the resource. That coordinator notices that this request was restarted by checking its list of ongoing evaluations, also clears its administration for this evaluation and assigns the request to a worker again. Secondly, in case there were no conflicts for subject attributes but the coordinator responsible for the resource detects a conflicting update of a resource attribute, that coordinator does not execute the updates of the resource attributes, but notifies the subject coordinator that the evaluation cannot commit. This coordinator then restarts the evaluation as described before and will also restart evaluations that employed the tentatively updated subject attributes (this is checked before checking for conflicting subject updates).

Note that this protocol does not suffer from race conditions itself because (i) the same coordinator is contacted for the same subject or resource, (ii) the same coordinators are responsible for managing an evaluation both before and after and (iii) these coordinators are contacted in the same order. Additionally, the messages between these coordinators should be guaranteed to arrive in the order as they were sent, which is a common property of current messaging technologies.

**Possible deployments.** Figure 6.8 illustrates three possible deployments using the distributed coordinator. Most interestingly, Figure 6.8c illustrates that the coordinator can be deployed on the same machines as the application instances. This effectively allows the access control system to scale out together with the application itself.

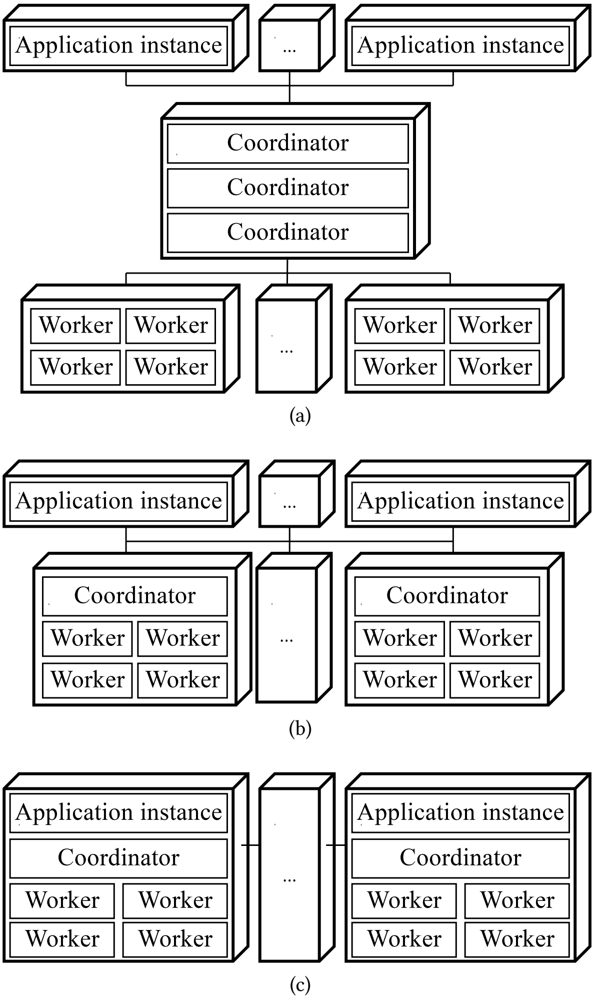


Figure 6.8: Three possible deployments using a distributed coordinator.

### 6.3.5 Scaling out the attribute database

As explained in the previous sections, our design employs the attribute database to store the attributes across multiple policy evaluations. More precisely, *all* workers read attributes from this database and *all* coordinators store updated attributes in this database. As a result, this database should be able to scale with the system and should therefore be distributed and/or replicated as well. However, for correctness, our design as explained for now requires this database to provide a consistent view of the attributes to all workers, which hinders its scalability as explained in Section 6.2.3.

To address this, each coordinator caches recently updated attributes and adds these to policy evaluation requests of the appropriate subject or resource (in step 2 of Figure 6.5 and steps 3 and 5 of Figure 6.7). This tactic is possible because the attributes of each subject and resource are cached by exactly one coordinator as a result of their distributed responsibilities. By then having each coordinator cache updated attributes longer than the inconsistency window of the attribute database, this tactic enables a fully consistent view for all workers, even while employing a scalable attribute database with eventual consistency and limited support for transactions such as Cassandra or MongoDB. As an additional result, our system is independent of the exact deployment of the attribute database (as long as it provides the necessary throughput for the policy evaluations), which is the reason why we do not show it in Figures 6.6, 6.7 and 6.8.

## 6.4 Evaluation

The previous section presented our concurrency control scheme for evaluation of history-based policies and explained how this scheme prevents race conditions from leading to incorrect decisions. Our goal was to achieve this in a scalable manner with low latency overhead. In this section, we validate this performance behavior using a prototype implementation. More specifically, we evaluate (i) the latency overhead of concurrency control, (ii) the impact of conflicts on the system, and (iii) the scalability of our scheme in three different deployments. First, we describe the prototype and test set-up.

### 6.4.1 Prototype and test set-up

To evaluate the performance of our concurrency control scheme, we developed an extensive prototype of it that integrates with the Amusa middleware (see Chapter 3). The prototype builds upon the STAPL language for tree-structured attribute-based policies. As explained in Section 2.3.1, STAPL is defined as a DSL in Scala and takes on the core model of XACML [5], but adds a simpler syntax and an efficient evaluation engine. The prototype is also written in Scala and employs the Akka actor framework for concurrency and distributed communication[4]. For our tests, we deployed the prototype on virtual machines on an internal cloud platform, each ranging from 2 to 8 CPUs and having sufficient RAM.

The policy employed in the tests is a realistic policy (i.e., not an artificial one) that resulted from the e-health case study of the patient monitoring application (see Chapter 4). The policy consists of 14 logical rules structured in a policy tree of depth 4. It employs 27 different attributes such as the shifts of nurses, the patients of a physician and the owner of a resource. The values of the attributes for the different subjects and resources are stored in MySQL databases and are fetched during policy evaluation. Because the resulting policy evaluation time varies heavily from request to request, we employ 29 different access requests that together cover the whole tree and we report averages over the complete set. The average evaluation time for these 29 different requests without any of our added functionality is $2.27ms$. All of our tests start with a warm-up phase that is disregarded in the results and the tests are repeated until the results are stable.

The code of the prototype, the employed policy and the test results are available on-line[5].

### 6.4.2 Latency overhead

As a first test, we measure the latency overhead of the coordination for concurrency control on policy evaluations without concurrency conflicts. More specifically, we measure the time between sending an access request and receiving the response while the workers perform an artificial policy evaluation of $0ms$. For the centralized coordinator, we deployed a client, a worker and a centralized coordinator on a single machine. For the distributed coordinator, we deployed the client and a varying number of distributed coordinators on individual machines. Each coordinator is deployed with 1 worker and a local replica of the attribute database.

For the centralized coordinator, the average overhead on every request is $0.98ms$. This time is mainly due to the asynchronous queuing nature of the prototype because of the use of Akka, which introduces a latency overhead in favor of scalability and throughput.

For the distributed coordinator, Figure 6.9 illustrates the overhead with respect to the amount of distributed coordinators. As we can see, the overhead of even 1 coordinator is notably more than for the centralized coordinator, which is because the client is now

[4] http://akka.io/

[5] http://people.cs.kuleuven.be/~maarten.decat/acsac2015/

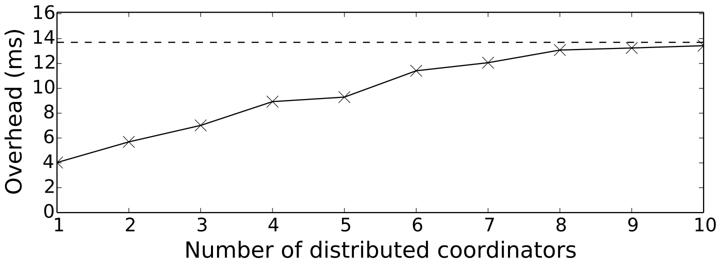


Figure 6.9: The overhead of the distributed coordinator with respect to the size of the pool of distributed coordinators.

located on a different machine. Moreover, this overhead grows for a larger number of distributed coordinators. However, the overhead asymptotically approaches an upper limit, which is the result of our strategy to uniformly distribute the management responsibilities over the available coordinators (see Section 6.3.4). As a result, the chance that the same coordinator is responsible for both the subject as well as the resource of a single evaluation becomes $1/nbCoordinators$. In other words, for an increasing number of coordinators, the overhead asymptotically grows from the situation where all requests are handled by a single coordinator to the situation where all requests are handled by two collaborating coordinators, which requires more remote communication. In this case, the latency overhead approaches $14ms$, which is limited and can amongst others be further improved by employing a more efficient network or more efficient marshaling.

**Conclusion.** This test demonstrates that our approach imposes a latency overhead on every request, but that this overhead is limited and asymptotically bounded with respect to the number of distributed coordinators.

### 6.4.3 The impact of conflicts

The previous test did not involve concurrency conflicts. Therefore, we now measure the impact of concurrency conflicts on the behavior of the system. More specifically, we measure the throughput and latency of requests that are sent one after the other while a centralized coordinator applies an artificial chance of conflicts ranging from 0% to 100%. The coordinator is deployed on a single machine with 4 CPUs together with 4 workers.

For the latency, the results show that an increasing chance of conflicts leads to an increasingly large group of requests that are restarted one or more times and thereby

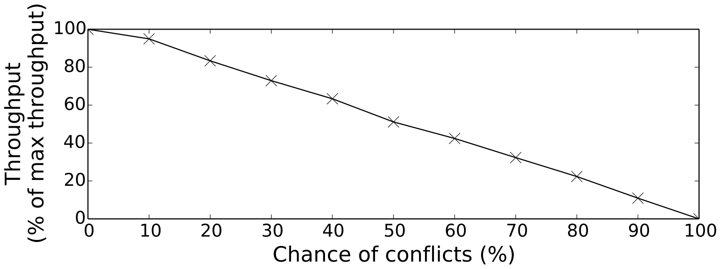


Figure 6.10: The throughput of a centralized coordinator compared to the chance for concurrency conflicts, measured on a single machine with 4 workers and 4 CPUs.

suffer increased latency. The result on the throughput of the system is depicted in Figure 6.10. As shown, the throughput is only minimally affected by the low numbers of conflicts that we expect in normal behavior. For an increasing chance of conflicts, the throughput systematically lowers (while the system is still working at its maximal capacity). The throughput eventually reaches 0 requests/second when every request keeps being restarted.

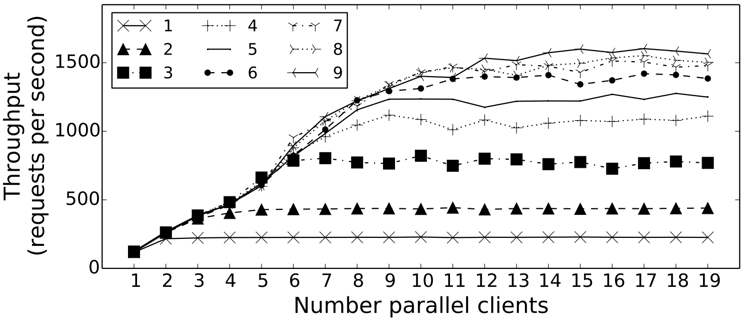
**Conclusion.** This test demonstrates that the number of conflicts lowers the effective throughput of the system. This behavior is independent of the deployment of the system. The impact is low when there is a low chance of conflicts, which we expect in normal behavior. However, our approach can be further optimized for specific situations with larger numbers of conflicts, such as controlled access to a resource shared by a large number of users.

### 6.4.4 Scalability

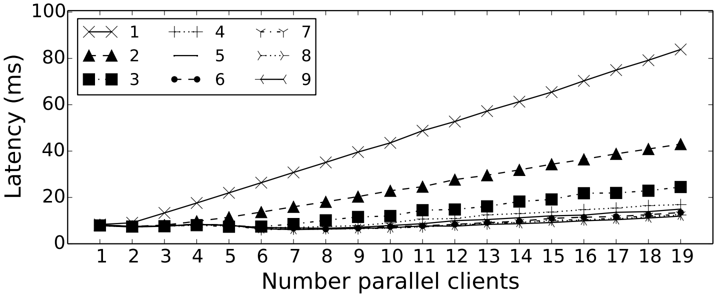
The previous tests zoomed in on the latency overhead of our concurrency control scheme and the impact of conflicts on its behavior. In the following tests, we evaluate the scalability of our scheme using three different deployments that illustrate that it can scale to large sizes of multi-core machines and a large number of machines. For clarity, we discuss the maximal capacity of the system by performing these tests without concurrency conflicts.

#### Centralized coordinator, co-located with workers

In a first deployment, we evaluate the behavior of a centralized coordinator and a varying number of workers deployed on a single machine (as was illustrated in



(a) Throughput



(b) Latency

Figure 6.11: The latency and throughput of a centralized coordinator and a varying number of workers deployed on a single machine of 4 CPUs, with respect to a growing number of parallel clients.

Figure 6.6a). More specifically, we measure the throughput and latency of a growing number of parallel clients that each repeatedly send one random request after the other.

Figure 6.11a illustrates the resulting throughput for a machine with 4 CPUs. As shown, the throughput of the system initially grows linearly with the growing load, but eventually reaches a maximum where every worker is fully employed. This maximum grows with the number of workers, but has a certain upper limit itself. At that point the maximal throughput does not increase any more by adding another worker because the hardware is fully employed. This upper limit depends on the number of CPUs of the machine and lies around 4 workers for 2 CPUs, 7 workers for 4 CPUs, and 9 for 8 CPUs, supporting approximately 800, 1500 and 2000 requests per second. For larger loads, the throughput of the system remains constant and does not decrease, which is the result of the fact that the prototype queues requests for a fixed number of processing threads instead of spawning new threads for new requests.

Figure 6.11b illustrates the resulting latency. As shown, the latencies initially lie around $6.5ms$, but start to grow linearly with the growing number of clients once the system is at its maximal throughput. This is the result of our test set-up with a fixed number of clients that only send a new request once the previous is answered, so that the queues in the system always contain a fixed number of requests that depends on the number of clients.

**Conclusion.** These results demonstrate that our system is capable of using a multi-core machine to its full capacity, but also that a single machine has a fixed maximum throughput. Once this upper limit is reached, the system should scale out in order to avoid latency increase. Notice that the upper limit shown in this test would also be the upper limit in case we would have employed database transactions on a fully consistent relational database instead of our domain-specific scheme for concurrency control as described in Section 6.2.3. Our goal is to scale beyond this limit, which these databases cannot.

### Centralized coordinator, multiple worker machines

In order to surpass the limitations of the previous deployment, we can employ multiple machines. Therefore, in a second deployment, we evaluate the behavior of a centralized coordinator that manages a set of 1 up to 10 worker machines, each hosting 4 workers (as was illustrated in Figure 6.6b). Each worker machine has 2 CPUs and also hosts a local replica of the attribute database. Because the throughput and latency behave similarly to Figure 6.11, we here focus on the maximal throughput of the system.

Figure 6.12 illustrates the resulting maximal throughput for an increasing number of worker machines and a coordinator hosted on a machine of 2, 4 and 8 CPUs. As shown, the maximal throughput initially grows for an increasing number of worker machines, but eventually reaches an upper limit. In this test, the upper limit is approximately 2800 requests/second for a coordinator with 2 CPUs and 4600 requests/second for 4 CPUs. For 8 CPUs, our tests are not conclusive whether an upper limit has already been reached for 10 worker machines.

**Conclusion.** This test demonstrates that our system can be scaled over the limits of a single machine supporting thousands of requests per second, but that a central coordinator still has an upper limit to scalability. In order to address this, the coordination should be distributed as well, which is the reason why we designed the distributed coordinator in Section 6.3.4. We evaluate the behavior of this distributed coordinator next.

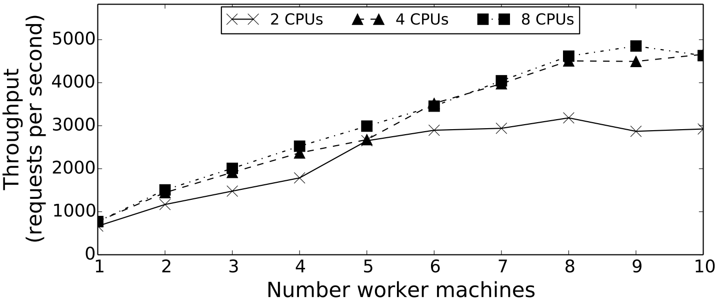


Figure 6.12: The maximal throughput of a centralized coordinator deployed on a machine with varying number of CPUs and serving a growing number of worker machines of each 2 CPUs and each hosting 4 workers.

### Distributed coordinator

In a third and final deployment, we evaluate the behavior of a group of distributed coordinators deployed on individual machines (as was illustrated in Figure 6.8c). Each machine has 2 CPUs and hosts 4 workers and a local replica of the attribute database.

Figure 6.13 illustrates the resulting maximal throughput for an increasing number of distributed coordinators. As shown, this throughput grows with the increasing number of coordinators. The increase of throughput is less than the maximal capacity of a single machine, e.g., from 898 requests/second for 1 coordinator to 1232 requests/second for 2 coordinators. This can be explained by the same reasoning as the increasing but bounded latency in Section 6.4.2 and the fact that once there are multiple coordinators, a part of their capacity is used for the intermediate messages. However, the resulting throughput is in the same order of magnitude as the previous deployment and most importantly, does not show an upper limit for at least 10 machines and 40 workers. This illustrates the horizontal scalability of our scheme.

**Overall conclusion.** To conclude, this whole evaluation verifies that our scheme for concurrency control is able to scale out to a large number of multi-core machines while incurring an asymptotically bounded latency overhead. This result was achieved by leveraging the domain-specific structure of a policy evaluation. The three different deployments in our evaluation support increasingly large throughputs and scalability, but are also increasingly complex. However, because the interface to the application is the same for all deployments, our system allows to migrate from the least complex to the most complex deployment when the application grows. As such, we believe that this evaluation demonstrates that our scheme can become an important enabler

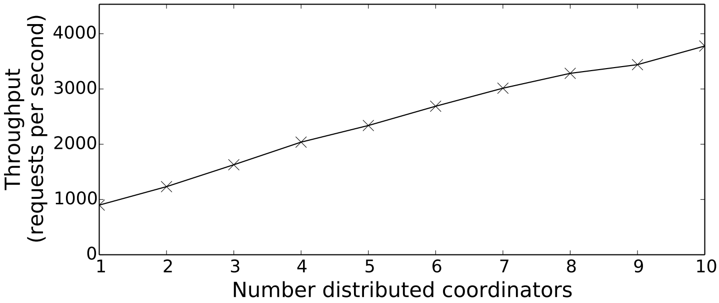


Figure 6.13: The maximal throughput of a growing number of distributed coordinators, each located on a machine of 2 CPUs and each managing 4 workers.

for enforcing history-based policies (and access control policies in general) in realistic and large-scale applications.

## 6.5 Discussion

In this chapter, we presented a scheme for concurrency control specifically for the evaluation of access control policies and demonstrated that it can scale out while incurring only a limited and bounded latency overhead. In this section we want to highlight the work of several other authors that have influenced this work in addition to the work already mentioned in the previous sections. Afterwards, we discuss other possible applications of concurrency and distribution complementary to this work, the need to take into account failure as future work and the vague border between application logic and access control logic.

**Related work.** This work builds on previous work on distributed policy evaluation of several other authors. For example, Chadwick [62] was one of the first to describe a distributed policy decision point with coordination using attribute updates specified in obligations, similar to this work. Alzahrani et al. [31] extend this into a decentralized network of policy decision point peers. Both however neglect the concurrency issues addressed by this work. Other authors do take these issues into account. For example, Dhankhar et al. [92] address these with locking, but report on policy evaluation times of seconds. This fortifies our choice not to opt for this approach. More recently, Kelbert and Pretschner [130] discuss a decentralized system for policy evaluation and build on the underlying database for concurrency control. For scalability, they employ Cassandra[1], which however does not support transactions over multiple attributes as required by policy evaluation (see Section 6.2.3). Finally, Janicke et al. [125] propose a formal model for the concurrency issues described in this chapter, and Gay et al. [111] discuss *service automata*, which can be regarded as policy decision points that can communicate with each other for coordinated distributed policy evaluation. This work fits into both theoretical frameworks.

**Other applications of concurrency and distribution.** This work focused on applying concurrency and distribution for increasing the throughput of policy evaluation. However, these techniques can also be employed for other reasons, such as evaluating a single policy concurrently for lowering the policy evaluation time or because the required access control data is distributed itself (e.g., [149]). These applications are complementary to this work as they require different forms of concurrency control, such as achieving a consistent view on the attributes across parallel branches of the same policy evaluation in the presence of out-of-band attribute updates or continuously changing attributes (e.g., time). As such, these applications pose interesting future directions to complement this work.

**Taking into account failure.** This chapter focused on addressing security issues that stem from concurrency in policy evaluation in the context of distributed systems. In addition to concurrency, distribution also introduces the need to take into account *failure* of parts of the system, especially if a system should consist of a large number of machines. Therefore, in order to achieve feasible scalable policy evaluation, taking into account failure is an important part of future work.

Other authors have also focused on the impact of failure on access control. For example, both Crampton et al. [74] and Tsankov et al. [199] formally discuss how to correctly handle failures such as an unreachable attribute database on the level of access control policies, but both still regard the policy decision point as a single and central component.

For our distributed policy decision point, we have to take into account failure of its individual components, i.e., of a worker, the centralized coordinator, a distributed coordinator or of the attribute database. In general, the centralized coordinator is a single point of failure, which can be addressed by using the distributed coordinator. Failure of a distributed coordinator and of a worker can be addressed by monitoring these components and restarting ongoing evaluations, as their state is limited to these ongoing evaluations. Finally, the attribute database hosts all the access control state and should therefore be able to survive failure without loss of data using persistence and replication. Because our separate layer of concurrency control allows to work with eventually-consistent databases (see Section 6.3.5), we can employ existing scalable solutions for this (e.g., Cassandra[1]).

One aspect of the distributed coordinator that should be handled in more detail is the redistribution of the subjects and resources for which each coordinator is responsible. More specifically, a failure of a single distributed coordinator should not affect the responsibilities of all coordinators as this would lead to large numbers of incorrect evaluations that have to be restarted. To solve this, we envision shifting from our current hash function to an existing solution such as the pseudo-random data distribution function of Ceph [206]. This tactic can then also be used for dynamically adding new coordinators to the pool at run-time.

Finally, notice that there is an interesting interaction between failure and history-based policies: the failure of a coordinator after having persisted an attribute update but before returning the decision to the application can deny a subject from rightful access to a resource, for example by incrementing the number of watched movies without actually having permitted the subject. This issue can be addressed by caching the results of policy evaluations so that the original decision will be given and the attribute is not updated twice.

**Application logic vs access control logic.** Finally, as an interesting side result of this work, we briefly discuss the line between application logic and access control logic. More specifically, during this work we often debated whether the access control system should perform the history updates or the application itself. This question aligns to a broader discussion about the border between application logic and access control logic.

In earlier access control models such as lattice-based and role-based access control (see Section 2.2.3), this border between application logic and access control logic was clear: access control was expressed in access control-specific concepts, such as security classifications or roles. ABAC however allows to express domain-specific and application-specific concepts, such as the number of times a user has watched a movie this month, and recent approaches such as relationship-based access control [78] further blur this border.

History-based policies and concurrency control are an interesting argument in this discussion. On the one hand, one can debate that the history is updated by the application itself, especially in case of domain-specific concepts such as the number of movies seen by a user in $P_2$, as these are often required by the application itself for reasons other than access control, such as movie recommendations for users. Another approach is to update the history information implicitly, for example using logs or provenance data as a source of history data (as recently suggested by Nguyen et al. [169]). However, in both approaches, the transactions for access control would span both access control code and application/logging code, which are possibly located on different machines (leading to distributed transactions that also span multiple abstraction layers) and which are supposed to be decoupled as much as possible in the approach of policy-based access control.

This illustrates that concurrency control is an aspect of access control that forces us to make a concrete distinction between application logic and access control logic for correctness and feasibility. In this work, we clearly argue that the access history (or at least the part of it relevant to access control) is best managed by the access control system itself and explicitly represented in the access control policies. If, however, the application also requires this data, it can be replicated in the access control system or the separate layer for concurrency control can temporarily cache updated attributes as described in Section 6.3.5, which demonstrates the wide applicability of our system.

## 6.6 Conclusion

This chapter presented a concurrency control scheme specifically for the evaluation of access control policies. This work is motivated by the need to enforce access control policies on large-scale distributed applications such as SaaS applications, in which the policies have to be evaluated concurrently and distributedly as well. However, for certain classes of policies such as history-based policies, concurrency can be exploited to gain elevated access. By leveraging the specific structure of a policy evaluation, our concurrency control scheme effectively prevents such incorrect access decisions and is able to scale to a large number of machines while incurring only limited and asymptotically bounded latency overhead. This result could not have been achieved by employing more general techniques for concurrency control. As such, this chapter has taken an important step to applying policy-based access control to realistic applications in practice.

# Chapter 7

# Conclusion

This chapter concludes this thesis. We first revisit the contributions of this thesis in light of the challenges for SaaS access control discussed in Chapter 1. Following from this, we then discuss a number of valuable directions for future research on policy-based access control in general. Finally, we wrap up with concluding thoughts.

## 7.1 Contributions

This thesis focused on access control for Software-as-a-Service (SaaS) applications. Because of the use of application-level multi-tenancy, SaaS must separate different tenants in application code. As a result, application-level access control is a crucial part of SaaS security.

However, SaaS access control is also challenging. For example, in addition to separating tenants, SaaS access control should also enable each individual tenant to constrain its own end-users in terms of the structure of its own organization. SaaS access control should also incur low management overhead for these tenants and should take into account the limited trust of these tenants in the SaaS provider. Finally, SaaS access control should be able to securely scale to the number of end-users of a SaaS application while incurring only limited latency overhead.

In this context, the contributions of this thesis are four-fold:

**Contribution 1.** Firstly, we presented and evaluated the Amusa access control middleware for multi-tenant SaaS applications. Amusa enables the provider and the tenants to express and enforce their specific access rules by building on policy-based access control with attribute-based policies. In addition, Amusa encapsulates this functionality in reusable middleware that requires low engineering overhead and incurs low performance overhead.

**Contribution 2.** Secondly, we described, evaluated and validated the concept of federated authorization. Federated authorization externalizes policy evaluation from a SaaS application. It thereby enables to centralize the overall access control management of a tenant and enables the tenants to enforce access rules on a SaaS application without having to disclose these rules nor the data they require. While there are still challenges to be addressed, the benefits of federated authorization lead us to believe that it is a key enabler for access control in future federated applications.

**Contribution 3.** Thirdly, we presented and evaluated the technique of policy federation. Policy federation automatically decomposes and deploys the policies of a tenant for a SaaS application so that the parts are evaluated close to the data they require as much as possible while sensitive access rules or access control data are kept local to the premises of the tenant. Policy federation was originally designed to optimize the performance of federated authorization, but more generally applies to set-ups where a policy is collaboratively evaluated across multiple policy decisions points.

**Contribution 4.** Finally, we presented and evaluated a scalable scheme for concurrency control for policy evaluation. This scheme avoids that concurrency leads to incorrect decisions when concurrently evaluating policies such as history-based policies. In addition, by building on the domain-specific structure of a policy evaluation, this scheme itself is able to scale to a large number of machines while incurring only a limited and bounded latency overhead.

## Validation

An important aspect of each of our contributions is its practical applicability. In order to achieve this, we validated the work of this thesis in multiple ways. For example, we validated the potential of federated authorization for collaborative e-health applications in an e-health research project and presented the results at a domain-specific e-health conference [90].

Most importantly, this work has been validated by starting from four realistic case studies: a SaaS application for electronic document processing, a SaaS application for automated workforce management, a SaaS application for monitoring patients of cardiovascular diseases at their homes and a platform for collaborative home care. Each of these case studies originated from collaborations with industry partners in research projects (see Section 1.4.1).

Based on these case studies, we identified the requirements and challenges for SaaS access control. The contributions of this thesis adhere to these requirements and our prototypes correspond to these case studies. In addition, our contributions were discussed, validated and refined in collaboration with the involved industry partners. The results of this process are presented in this thesis.

### Evaluation

Another important aspect of our work is its impact on performance. More precisely, access control should incur low latency overhead on the legitimate use of the (SaaS) application that it constrains. In addition, access control should be able to scale to equally large amounts of requests per second as this application.

In order to achieve this, our contributions of policy federation and the concurrency control scheme are specifically aimed at performance. In addition, all of our contributions are evaluated in terms of performance. For this, we systematically developed prototypes and statistically tested these. The code of these prototypes and the full set of measurements is available on-line (see Appendix D).

In addition to performance, we also employed our prototypes to build practical experience with the employed technologies and to evaluate the required engineering effort for incorporating the Amusa middleware in a SaaS application.

## 7.2 Revisiting the challenges for SaaS access control

The previous section summarized the contributions of this thesis. As with any research, the work in this thesis is not complete yet. Each of the previous chapters already discussed future work specifically for each of our contributions. In this section, we discuss the limitations of this work for SaaS access control in general (see Chapter 2) by revisiting the challenges listed in the introduction of this thesis (see Section 1.2). For clarity, we employ the same structure as Section 1.2 in this discussion.

### Functional challenges

Firstly, Section 1.2 highlighted that SaaS access control should separate the tenants in a SaaS application as well as enable the provider to constrain its tenants and enable these tenants to constrain their end-users. The Amusa middleware supports this functionality, but purely from a functional perspective, so do state-of-practice SaaS applications. As we will discuss next, our contributions and limitations mainly lie in the non-functional challenges.

### Non-functional challenges from multi-tenancy

On top of the functional challenges for SaaS access control, Section 1.2 highlighted several non-functional challenges. A first set of such challenges originates from the usage of application-level multi-tenancy in SaaS applications. Firstly, multi-tenancy requires to be able to modify access rules for an application without having to recompile or restart this application. Secondly, every tenant wants to be able to enforce access rules specific to its own domain and organization, which leads to a variability challenge for access control.

In this work, we addressed the former challenge by opting for policy-based access control. This technology enables both the access control data as well as the access rules themselves to vary per tenant and to modify these without having to recompile or restart the application. In addition, we opted for attribute-based policies, which enables the tenants to express a wide variety of access rules. As such, this work effectively demonstrates that the challenges from multi-tenancy can be addressed based on these technologies.

However, there are still limitations to our approach in order to apply it in practice. For example, this work showed that policy-based access control comes with a non-negligible performance overhead. In our case, this performance overhead mainly stemmed from having to fetch attributes during policy evaluation. In response, we developed and discussed techniques that can lower this impact. If needed, this overhead can even be avoided completely by only employing attributes that are already known when the access decision is needed or that can be cached. This approach would provide the benefits of policy-based access control without its performance overhead at the cost of limiting the rules that can be expressed. Further research is required to investigate the extent of these limitations in practice.

Next to the performance impact however, there are also more fundamental limitations to our approach. For example, while attribute-based access control enables tenants to express a wide variety of access control concepts including domain-specific concepts, they still do not suffice to express some fundamental access rules (this is further discussed in Section 7.3.3). Moreover, Amusa currently only supports to enforce policies on individual requests for individual resources. As a result, Amusa for now fails to enforce policies on, for example, database queries that return multiple results. Additionally, in terms of low engineering overhead, Chapter 3 concluded that the effort to incorporate Amusa in an application is moderate to low. However, here we assumed that the attributes of subjects and resources are readily available somewhere in the infrastructure of the organization. This in turn requires a structured access control culture in the organization, which may pose a hurdle for the adoption of this technology in large and complex organizations with legacy infrastructure. All of these challenges require future research. However, these are not limitations of our approach per se, but rather limitations of the current state of policy-based access control in general. Based on these limitations, we discuss future directions for research on policy-based access control in Section 7.3.

### Non-functional challenges from the large scale of SaaS applications

A second set of non-functional challenges for SaaS access control originates from the large scale of SaaS applications. In this regard, Section 1.2 discussed that the large number of tenants requires SaaS access control to provide self-management to these tenants, but that this self-management should still be guaranteed secure. Additionally, on a more technical level, Section 1.2 discussed that the large scale of SaaS applications challenges performance. More precisely, SaaS access control should support a high throughput of access decisions while each decision is reached with low latency. To achieve this, SaaS access control should amongst others also function correctly in the context of a scalable distributed infrastructure.

With respect to the challenge of secure self-management, the Amusa middleware demonstrated that the technology of policy-based access control with tree-structured policies can enable tenants to autonomously specify their own access rules while the policy trees guarantee the security properties of the end result. With respect to the challenge of scalable access control, this thesis focused on scaling out policy evaluation because of our focus on policy-based access control. More precisely, we defined a concurrency control scheme to address the possible concurrency issues when evaluating access control policies.

Again, there are limitations to our approach in order to apply it in practice. Firstly, with respect to self-management, our approach is limited by the fact that it is currently still challenging and cumbersome to write a correct policy for a certain application. Two reasons for this are that there is no overall view on which resources, actions and attributes an application supports and that there is only limited tool support for policy editing. Additionally, while the current policy languages still do not support some basic business rules, these languages are already hard to use for non-expert users. Again, these limitations are not limitations of our specific approach, but of policy-based access control in general.

Secondly, with respect to scalable access control, our approach is limited by the focus on concurrency control. While this focus does address a fundamental challenge in scaling out policy evaluation, this is not the only challenge to address. For example, next to concurrency, distribution also introduces the need to take into account failure of parts of the system, which has not been incorporated into our approach yet. This is discussed in more detail in Section 6.5. Another limitation of our approach is that while we achieve a result that could not have been obtained by generic database technologies, our approach only applies to the evaluation of policies that involve a single subject and a single resource. Although every access control policy that we have encountered in our case studies as well as in literature adheres to this, future research is required to investigate how this approach can be generalized or at least transferred to other types of policies if and when such policies come up.

### Non-functional challenges from outsourcing

A final set of non-functional challenges for SaaS access control originates from the fact that SaaS is a form of outsourcing. In this regard, Section 1.2 explained that tenants do not necessarily trust the provider of a SaaS application completely.

In this context, this thesis focused on enabling tenants to enforce an access control policy of which they do not want to disclose the data required to evaluate it or the policy itself. This situation presents itself mostly in privacy-sensitive domains such as e-health.

While our approach of addressing this challenge by externalizing policy evaluation from the SaaS application has certain limitations itself (see Chapter 4), there also exist other trust requirements that we did not address. For example, the tenant still has to trust the provider that its policies are correctly enforced by the SaaS provider to every request to this application. In addition, the tenant still has to trust the SaaS provider that its data has not been read or tampered with by an attacker or by the provider itself. These challenges are currently in investigation by other researchers, e.g., by encrypting the data of the tenant before transmitting it to the provider in order to guarantee integrity or confidentiality. Future research is required to see how their work can be integrated with ours.

### Additional concerns: low performance, management and engineering overhead

In addition to the functional and non-functional requirements discussed above, we also took into account the additional concerns that the access control techniques designed in this thesis should impose limited latency overhead, management overhead and engineering overhead. As such, we designed our contributions with these concerns in mind and evaluated these systematically afterwards.

In addition, Amusa was designed specifically to provide efficient incremental access control management based on attributes. A limitation here is that it has not been proven yet that attributes actually facilitate efficient access control management, tough this belief seems to be growing [123]. Moreover, the technique of federated authorization was designed to lower the management overhead for an organization employing multiple SaaS applications by means of centralization. Apart from the future challenges for the adoption of this technique in practice discussed in Chapter 4, the limitation of this work is that we have not empirically studied whether centralization can effectively lower the management overhead, although again this does align with common belief.

### Overall conclusion

The focus of this thesis was to design access control techniques that address the specific challenges for access control in SaaS. Our approach was to build upon the existing state-of-the-art technologies of policy-based access control with attribute-based tree-structured policies and investigate whether and how these techniques can be applied to address these challenges.

In this regard, our contributions and experience lead us to believe that these technologies should indeed be able to address the SaaS-specific challenges for access control. However, our contributions still have their limitations of which the most fundamental ones are caused by the limitations of these technologies, mainly of policy-based access control. As such, we believe that the major challenges for policy-based access control also lead to the major remaining challenges for SaaS access control. As a result, we see these challenges as the most important directions for follow-up research to this thesis and we discuss them in the next section.

## 7.3 Future directions for policy-based access control

The previous section discussed the limitations of this work in the context of SaaS access control by revisiting the challenges identified in Chapter 1. We believe that some of these limitations are not specific to our approach, but are fundamental challenges to policy-based access control as a whole. These challenges have to be addressed in order to successfully apply this technology in practice, also outside the scope of SaaS.

In this section, we discuss the most important of these challenges as tracks for future research: investigating the semantical interface between policies and an application, applying policies to database queries and providing supporting tools. In addition, we discuss the link between audit and authorization as an interesting opportunity for future research. All of these tracks of future research fit into a larger view on policy-based access control, which we discuss in the end.

### 7.3.1 Investigating the semantical interface between policies and applications

A first major and fundamental open challenge for policy-based access control lies in the interface between policies and the application on which they are enforced. More precisely, policy-based access control aims to externalize authorization, i.e., policy evaluation, from an application so that the access rules can be changed without having to change the application code. The current state of policy-based access control achieves this. However, an access control policy should still reason about the application that it constrains. For example, in terms of attribute-based access control, the editor of a policy should know which types of resources an application provides, which actions these support and which attributes these contain. This knowledge is required to specify a *correct* policy, e.g., one that only employs existing attributes, and to specify a *complete* policy, i.e., one that covers all types of resources and all actions supported by an application. We call this information the *semantical interface* between the policy and the application.

Currently, technologies such as XACML require this information to be communicated through documentation. As such, there is currently no way to verify whether a policy covers all actions on all types of resources of the application and a simple typo can lead to an incorrect attribute statement in a policy that will only be detected when evaluating this policy at run-time. This makes it still challenging and cumbersome to write a correct policy for a certain application. As such, the challenge for policy-based access control is to develop techniques that facilitate correct and simple policy specification but also maintain the decoupling of policies from application code.

A possible solution to this problem is to introduce a new software artifact, e.g., a model of the structure of the resources present in an application, their supported actions and their provided attributes. Amongst others, this model can then be used in a policy editor to check the correct types and names of the employed resource attributes and to check whether the policy provides a decision for every action on every type of resource. Ideally, this model does not introduce yet another software artifact to be maintained by the developers of an application, but should be derived from existing artifacts such as architectural diagrams, interface definitions or annotations in code. A first step towards such a model has already been explored by Verhanneman et al. [201] based on aspect-oriented programming, an approach that we can build upon.

In extension, this approach can also be applied to an organization that employs one or more applications. More precisely, any organization has a certain structure of its members, e.g., a certain hospital can operate general practitioners, specialists, nurses and supporting staff, these can be organized in teams, departments and floors, each nurse has his or her own shift hours etc. This structure determines the

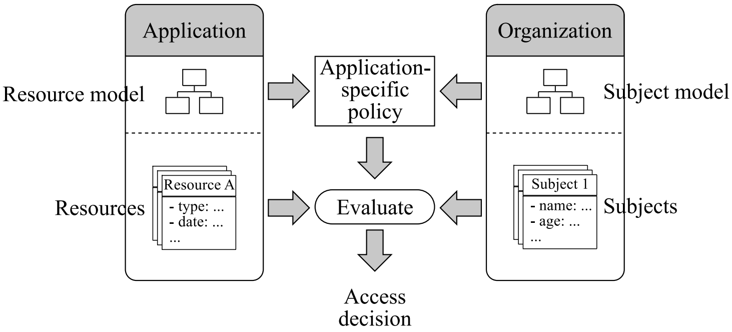


Figure 7.1: Our vision on simplifying the specification of a policy for a certain application and organization: for achieving correctness and completeness checking, this policy should be based on a model of the resources and actions in the application and a model of the subjects in the organization.

structure of the subjects of an organization and their attributes. As such, the policy of a certain organization for a certain application in essence maps this structure of subjects to the structure of resources provided by the application in order to determine who is permitted to perform which actions on which resources in which conditions (this vision is illustrated in Figure 7.1). Because this structure of the subjects of an organization remains constant across the applications that the organization employs, it is an interesting idea to make this structure concrete in a software artifact similar to the resource model of an application described before. This could further ease the burden of writing a correct access control policy for a certain application.

Our recent work on policy reuse [89] provides a first step towards expressing these resource and subject models. However, we have also learned that the interface between the application and the organization that employs it is more fine-grained than discussed before. One reason for this is that an application commonly also makes assumptions on the structure of its users, e.g., that each user has an e-mail address for the e-Docs application or that there are patients and supervising physicians in the patient monitoring application. As such, more research is required to address this fundamental challenge.

### 7.3.2 Applying policies to database queries

A second open challenge for policy-based access control is applying policies to non-request-response paradigms. More specifically, policies are now mainly enforced on individual requests to an application that each represent a single action on a single resource. Examples of these actions are “send a document”, “view the status of a patient" or "update the progress of a task". In this case, the authorization question is "Is this subject permitted to perform this action on this resource?". As such, the policy is evaluated for this specific request and a decision is returned.

However, some types of applications and some operations in certain applications do not align to this paradigm. An important instance of such operations is searching in a database and listing the results. For example, in the case of e-Docs, a common operation is that a user views a collection of documents such as the documents in his or her inbox, the documents of a customer for which he or she is responsible or the documents that belong to a project in which he or she participates. As such, listing these documents entails a search query in a database of potentially millions of documents and the result should only contain documents that match the query *and* that the subject is permitted to access. In the approach of policy-based access control, the latter is specified in policies, which means that policies should somehow be enforced on database queries.

A straight-forward approach to enforce access control policies on a database query is to perform the query without taking into account the policy and afterwards evaluate the policy for every result of the query to filter out denied results. It is clear however that this brute-force approach will not provide suitable performance because of the overhead of fetching too many entries from the database and then having to evaluate the policy for every such entry. As such, more efficient techniques for enforcing access control policies on database queries are required.

Because many common applications employ operations such as the examples above, we believe that this challenge is a second challenge that is fundamental to the successful adoption of policy-based access control. Other authors have recently also focused on this challenge. For example, Rizvi et al. [178] investigated the use of database views for enforcing access control in a transparent way and Jahid et al. [124] compile XACML policies into access control lists that are natively supported by some databases. Another interesting tactic is to automatically translate the access control policies into database queries and merge these with the application query so that only permitted entries are returned. The advantages of this tactic are that no specialized functionality is required from the underlying database. An open question for this tactic however is whether an access control policy can fully be translated into a database query and which information is required about the database in order to automate this process. We believe that this poses an interesting and important track for future research.

### 7.3.3 Supporting tools and technologies

In addition to the challenges discussed before, we also identified the need for tools and technologies that support policy-based access control. While we regard the previous challenges as more fundamental to the approach of policy-based access control, we do believe that these tools and technologies are essential to its adoption in practice and also pose interesting research challenges.

### Expressive policy languages

A first required technology is a policy language that actually enables us to express the access rules that we want to express. More specifically, we require a policy language that is more expressive than the current attribute-based languages such as XACML. As explained in Chapter 2, this thesis employs attribute-based access control because attributes enable us to express a wide variety of access control concepts. However, while attributes support far more access control concepts than previous models, it still does not support some basic rules from our case studies such as "*a physician can only access data of patients whom he treated in the last six months*". In terms of attributes, this rule would require an attribute such as `patients_treated_in_last_6_months`, which actually encodes an access rule in an attribute. As a result, changing this rule to the last 7 months requires an attribute to be changed, which violates the separation between access rules and access data.

As explained in Section 2.2.3, new access control models are currently being developed, amongst others in response to this problem. One recent and advanced model that we especially believe in is the model of *Entity-based access control* (*EBAC*), which expresses access rules in terms of the entities in an application. EBAC thereby unifies attribute-based access control and the recent notion of relationship-based access control [61, 104, 66]. We have recently extended STAPL to support tree-structured entity-based policies [52], but more research is required to validate the potential of this model and design a complete and easy-to-use supporting policy language.

### User-friendly policy languages

In addition to a more expressive policy language, our work also identified the need for a more user-friendly policy language. More specifically, the experience of employing XACML in our own prototypes and explaining this technology to industry partners in research projects shows that this language has a steep learning curve and is overall hard to use.

A first cause for this is the use of XML, which is not easy to read or write for humans and leads to large and verbose policies. As explained in Section 2.3.1, possible solutions to this problem are the use of graphical user interfaces to generate XACML policies, the use of a more user-friendly policy language to generate XACML policies and the design of a more user-friendly policy language that can be directly interpreted by a policy decision point. The STAPL language [89] follows the latter approach, but future research is required to show whether this is the most suitable solution.

A second and more fundamental cause for the steep learning curve of XACML are the rule expressions and policy trees themselves. While these concepts are fairly easy to apprehend for access control experts with programming experience, our experience shows that correctly expressing a large policy with multiple detailed rules is still a challenge for most non-expert users. One possible tactic for addressing this challenge is to enable the user to specify a policy in a user-friendly format and translate these rules to a more technical policy language such as XACML or STAPL. Examples of such formats are UML (e.g, as discussed by Lodderstedt et al. [155]) and natural language (e.g., as discussed by Shi and Chadwick [188]). Another tactic is to enable experts to define complex rules and policies as reusable higher-level patterns that can easily be reused by non-expert users in their policies. We recently investigated this tactic as an expansion to STAPL [89] and believe that this approach can gravely simplify policy specification. However, addressing this problem holistically requires a multi-disciplinary approach involving amongst others security experts, programming language experts and usability experts.

### Policy analysis

In addition to an expressive and user-friendly language for specifying policies, another important supporting technology is the automated analysis of policies. More precisely, when specifying policies, questions come up such as "Does this policy cover all actions in the application?", "Does this rule conflict with any of the other rules?", "Which resources can this subject access?" and "Which subjects can access this resource?". Similarly, when updating a policy, the question arises of what the impact is of the policy change, i.e., "Which additional actions are now permitted to which subjects and which actions are now denied?". Discussions with industry partners in our research projects have pointed out that being able to answer these questions is crucial for the trust in policy-based access control in practice.

These questions have already been identified in the early work on policy-based access control and management, e.g., by Sloman [190]. As a result, other authors have already focused on the challenge of answering these questions in the past. For example, Fisler et al. [103] encode XACML policies as multi-terminal binary decision diagrams in order to formally verify properties of the policies and perform change-impact analysis. Kolovski et al. [135] take a similar approach based on description logic. Additional research is needed to investigate how these solutions integrate with the work in this thesis, e.g., the layered policy management of Amusa, and whether these solutions scale to the amount of subjects and resources in the SaaS applications of our case studies.

### 7.3.4 The link between authorization and audit

Finally, we want to highlight to potential of combining authorization and audit. As opposed to the previous sections, this is not a fundamental challenge for policy-based access control, but rather an opportunity to further improve its use.

This thesis focuses on authorization, but authorization is only a part of access control (see Section 2.1). More specifically, the process of access control is generally divided into authentication, authorization and audit. Authentication makes sure that the subject is who he, she or it claims to be, authorization verifies that an authenticated subject is permitted to perform the requested action before the action is performed and audit afterwards checks whether a subject performed an action that should have been denied, for example based on logs.

As such, authorization and audit make up two interesting complementary technologies. On the one hand, authorization is able to prevent an action if it is not permitted, but the complexity of the rules that can be enforced is limited because of the tight performance requirements. On the other hand, audit can employ a far larger set of access control data and is far less constrained by timing requirements. Audit can therefore enforce more complex access rules and even search for patterns across multiple requests, but it is not able to prevent an action from being performed and therefore requires it to be possible to roll back the action.

As a result of this complementarity, we believe that the combination of authorization and audit can be very powerful. For example, such a system could be used to enforce complex access control policies a posteriori while they pose too much performance overhead for authorization and gradually shift enforcement to a priori authorization as technology improves. Similarly, audit data could be used to verify the intended effect of an authorization rule. Although this approach still requires a leap in technology, e.g., a common policy language for both authorization and audit and semantically linking the authorization and audit data, we believe that this idea is promising and is worth working towards in the future.

### 7.3.5 The complete picture: a view on policy-based access control

The research directions discussed in the previous sections all fit within a broader view on applying policy-based access control in practice.

In this view (illustrated in Figure 7.2), the access rules that an organization wants to enforce on the applications that it employs are specified in declarative policies, which is the core of policy-based access control. Ideally, the organization-wide rules are specified by security managers of the organization as high-level business rules, in terms of the structure of the organization (which is represented as a subject model) and

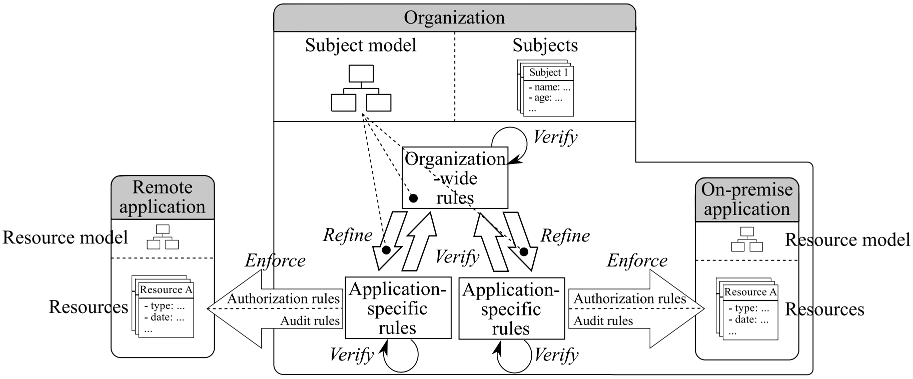


Figure 7.2: Our view on applying policy-based access control in practice.

by means of a user-friendly policy language. An example of such an organization-wide rule is "*medical professionals are not permitted to access medical data of patients that have withdrawn consent for them*".

In turn, these organization-wide rules are then refined into more detailed application-specific access rules that hold for each individual application. This refinement can be automated by using the resource model provided by each application. For example, the medical professionals involved in the patient monitoring system of our case study are physicians and nurses and the medical data in this application are the monitoring data and status overview. In addition, the resource model and subject model are used to automatically verify the correctness and completeness of resulting rules. Depending on their complexity and size, the application-specific access rules are then enforced by means of either authorization or audit. Afterwards, the audit data is also used to automatically analyze whether the access rules enforce the behavior intended by the security managers. This approach could further be extended to derive security policies of even lower levels in the infrastructure, such as firewall configurations.

In summary, this view on policy-based access control employs policies as software artifacts that enable automation for more efficient, more scalable and more correct access control management. This view aligns with visions articulated by other authors as early as 1994 [190], but as the previous sections explained, there are still challenges to be addressed before it can be made reality.

## 7.4 Concluding thoughts

This thesis focused on the problem of access control for SaaS applications. These applications are characterized by their architectural choice for application-level multi-tenancy, their scale and the fact that they are a form of outsourcing from the point of view of the tenants. As such, this thesis presented and evaluated techniques for access control that enable the provider and the tenants to enforce their specific access rules on the common SaaS application, that lower the management overhead for a tenant, that incur low performance overhead, that take into account the limited trust of a tenant in the provider and that are able to scale out together with the SaaS application.

This work fits into the larger field of access control and access control management, a field often coined as Identity and Access Management (IAM). This field is currently witnessing a large interest in industry for several reasons. Amongst others, as an increasing amount of business information and processes is digitized, being able to express and enforce who is permitted to access what is a crucial part of security. At the same time, organizations grow both in size and complexity, consisting of diverse types of employees that are structured in departments, teams and projects that span multiple regions and sectors. This makes it more challenging to express who can access what, even within the scope of only a single organization. Moreover, businesses are increasingly becoming specialized, as for example shown by the O'CareCloudS project [15] for the e-health sector. As a result, business relationships also grow in complexity. Specifically for IT, the current evolution towards cloud computing and SaaS further stresses the need for access control because business data is increasingly often stored and processed outside of the physical premises of an organization and "traditional" security tools such as firewalls cannot protect this data anymore.

In order to cope with these trends, new access control techniques are being developed at an increasingly fast pace. Because of the advent of professional web applications for example, we have seen the evolution towards federated authentication with initiatives such as OpenID [6] and SAML [2]. Afterwards, OAuth [122] took the first step towards federated authorization and is now paving the way for more complex protocols such as OpenID Connect [180] and UMA [23]. Also outside the scope of web applications, organizations are transitioning toward centralized user management and some are pioneering by taking the step towards attribute-based management of their members and data [123]. Externalized authorization is more and more regarded as the next step towards efficient access control management, which eventually can lead to third-party services for centralized authorization management.

The results presented in this thesis fit within these evolutions with a focus on efficient access control for SaaS. In addition, the basic technologies employed in this thesis, i.e., attribute-based access control and policy-based access control, are potential answers to the challenge of efficient access control management in an increasingly complex environment. However, these technologies are just now starting to be applied in practice and it will still take a long time before we can fully apply the results of access control research in practice for the reasons outlined above. In this regard, this thesis has developed and evaluated techniques that again take us one step closer towards successfully applying these technologies and achieving efficient large-scale access control management in practice.

# Appendix A

# Example of an access control policy

Chapter 2 mentioned that XACML and STAPL allow to express a wide spectrum of access control concepts, including domain-specific concepts. This appendix demonstrates how some of the most prominent of these concepts can be expressed using an example policy from the eDocs case study (see Section 1.4.1).

The example policy is shown in Listing A.1. This policy employs the syntax of STAPL. As shown, STAPL supports policy trees built from `Rules` and `Policies`. The former specify attribute-based conditions using the keyword `iff`, the latter specify an attribute-based target using the keyword `when` and a combination algorithm using the keyword `apply`. Both can also specify obligations using the keyword `performing`.

Using these primitives, the following access control concepts can be expressed:

**Identity-based policies.** The most basic access control concept is permitting or denying specific subjects to access specific resources. XACML and STAPL support this by using the identifiers of the subject and the resource. For example, lines 5 and 6 of Listing A.1 contain an identity-based rule that specifically permits subject “user123” to view resource “docABC”.

**Permission-based policies.** Secondly, XACML and STAPL support to reason about permissions assigned to subjects by using a multi-valued attribute. For example, lines 25 and 26 of Listing A.1 deny an account manager to view a document if he or she does not have the permission “read_document”.

```
Policy := when (resource.type == ‘document’)
          apply FirstApplicable to (
  // For viewing documents
  Policy := when (action.id == ‘view’) apply FirstApplicable to (
    Rule := permit iff (subject.id == ‘user123”
            and resource.id == ‘docABC’)
    // For helpdesk operators
    Policy := when (‘helpdesk_operator’ in subject.roles)
              apply DenyOverrides to (
      Rule := deny iff (not (environment.now >= 08:00
              and environment.now <= 17:00)),
      Rule := deny iff (not (subject.location == ‘Brussels’)),
      // Dynamic separation of duty
      Policy := when (resource.owner == ‘Bank A’)
                apply PermitOverrides to (
        Rule := permit iff (not (‘Bank B’ in subject.history))
                performing (append(‘Bank A’, subject.history)),
        Rule := deny
      ), // the dual policy for Bank B here
      Rule := permit
    ),
    // For account managers
    Policy := when (‘account_manager’ in subject.roles)
              apply DenyOverrides to (
      Rule := deny iff (not (
              ‘read_document’ in subject.permissions)),
      Rule := deny iff (not (
              resource.owner_org in subject.assigned_customers)),
      Rule := permit
    ) performing (
      log(subject.id + ‘ was denied ’ + resource.id) on Deny
    ),
    // For recipients
    Policy := when (‘recipient’ in subject.roles)
              apply DenyOverrides to (
      Rule := deny iff (not (subject.id == resource.owner)),
      Rule := permit
    )
    ... // rules for other types of subjects
  ),
  // For sending documents
  Policy := when (action.id == ‘send’)
            apply PermitOverrides to (
    // Sending quota
    Rule := permit iff (subject.nb_sent_this_month < 1000)
            performing (increment(subject.nb_sent_this_month)),
    Rule := deny
  ),
  ... // rules for other actions
)
```

Listing A.1: An example of a STAPL policy that combines multiple access control concepts in the context of the eDocs system for electronic document processing.

**Role-based policies.** Thirdly, XACML and STAPL support to reason about roles, again using a multi-valued attribute. For example, lines 8, 23 and 34 of Listing A.1 each check the roles of the subject in the target of a policy. Hierarchical roles can then be emulated by flattening the hierarchy into `subject.roles`. Similarly, checking whether the subject has a certain permission in one of his or her roles can be emulated by providing the list of all permissions of the subject from his or her roles as `subject.permissions`, which can be used similarly to the permission-based policy above. While STAPL does provide more extensive primitives to reason about hierarchical roles [89], these are not employed in this thesis in order to remain compatible with XACML.

**Group-based policies.** XACML and STAPL also support to reason about groups, which is similar to roles. Because of this similarity, Listing A.1 does not show a specific example of groups.

**Ownership-based policies.** XACML and STAPL also support to reason about ownership by using the identifier of the subject and the owner of a resource. For example, line 36 of Listing A.1 denies a recipient to view any other document than his or her own documents.

**Domain-specific ownership-based policies.** XACML and STAPL also support domain-specific forms of ownership using similar attributes as illustrated before. For example, lines 27 and 28 of Listing A.1 deny an account manager to access documents of customers other than the ones that are explicitly assigned to him or her. In this case, the concept of documents is specific to the application and the concept of assigned customers is specific to the organization.

**Time-based policies.** XACML and STAPL also support to reason about the date and time by using an environment attribute. For example, lines 10 and 11 of Listing A.1 deny a helpdesk operator to view documents outside of shift hours.

**Location-based policies.** XACML and STAPL also support to reason about the location of a subject. For example, line 12 of Listing A.1 denies a helpdesk operator to view a document from outside the head office in Brussels.

**Obligations.** XACML and STAPL also support obligations, which can be used to execute an operation in conjunction with enforcing the access decision. For example, lines 30 and 31 of Listing A.1 specify to write out a log in case an account manager was denied access to a document.

**History-based policies.** XACML and STAPL also support history-based policies by using obligations to update an attribute that contains the history. For example, lines 45 to 46 of Listing A.1 specify that each user can only send 1000 documents each month. In this case an obligation specifies to increment the attribute `subject.nb_sent_this_month` on Permit. Chapter 6 goes deeper into these history-based policies.

**Dynamic separation of duty.** As a special case of history-based policies, XACML and STAPL also support dynamic separation of duty policies or Chinese wall policies [56]. For example, lines 14 to 19 of Listing A.1 deny a helpdesk operator to access documents of Bank B once he or she has had access to documents of Bank A and vice versa. In this case, the `subject.history` attribute contains the organizations of which the helpdesk operator has viewed documents. Again, this attribute is updated by means of an obligation.

**Combinations of the above.** Finally, the power of XACML and STAPL lies in the fact that any of the concepts demonstrated before can be combined into a single policy by using policy trees. Listing A.1 illustrates this nicely. In full, the complete policy for the eDocs case study combines over 40 different rules and employs over 30 different attributes. Similarly, the policy for the patient monitoring system combines over 20 different rules. The resulting policies employ all of the concepts illustrated above, which illustrates the need for complex and fine-grained access policies as discussed in the challenges for SaaS access control (see Section 1.2).

# Appendix B

# Extensions to XACML for federated authorization

This appendix lists and illustrates the extensions to XACML 2.0 performed for supporting federated authorization as discussed in Section 4.3.3.

## B.1 Remote Policy Reference

The XML schema for the `<RemotePolicyReference>` element is given in Listing B.1, an example of its usage is given in Listing B.2.

```
<xs:element name=”RemotePolicyReference”
    type=”RemotePolicyReferenceType”/>
<xs:complexType name=”RemotePolicyReferenceType”>
    <xs:element ref=”xacml:Description” minOccurs=”0”/>
    <xs:element ref=”xacml:Target” minOccurs=”0”/>
    <xs:element ref=”xacml:Obligations” minOccurs=”0”/>
    <xs:attribute name=”PolicyId” type=”xs:string”
        use=”required”/>
</xs:complexType>
```

Listing B.1: XML schema of the new `<RemotePolicyReference>` element.

```
<RemotePolicyReference PolicyId=”tenantA”>
 <Description> The policy for tenant A. </Description>
 <Target>
  <Resources>
   <Resource>
    <ResourceMatch MatchId=”string-equal”>
     <AttributeValue DataType=”string”>
       A
     </AttributeValue>
     <ResourceAttributeDesignator
      AttributeId=”resource:owning-tenant”
      DataType=”string”/>
    </ResourceMatch>
   </Resource>
  </Resources>
 </Target>
</RemotePolicyReference>
```

Listing B.2: Example usage of the `<RemotePolicyReference>` element. Namespaces are not shown in full for readability reasons.

## B.2 Obligation targets

The XML schema for the extended `<Obligation>` element is given in Listing B.3, an example of its usage is given in Listing B.4.

```
<xs:element name=”Obligation” type=”xacml:ObligationType”/>
<xs:complexType name=”ObligationType”>
    ... <!-- Id, attribute assignments and effect -->
    <xs:attribute name=”FulfillWhere” type=”LocationType”
        use=”optional” default=”local”/>
</xs:complexType>
<xs:simpleType name=”LocationType”>
    <xs:restriction base=”xs:string”>
        <xs:enumeration value=”local”/>
        <xs:enumeration value=”remote”/>
    </xs:restriction>
</xs:simpleType>
```

Listing B.3: XML schema of the extended `<Obligation>` element. For the specification of the original element we refer to Section 5.45 of [165].

```
<Obligation ObligationId=”log access” FulfillOn=”Permit”
    FulfillWhere=”local”>
    <AttributeAssignment AttributeId=”log:access”
        DataType=”string”>
        <SubjectAttributeDesignator
            AttributeId=”subject:subject-id”
            DataType=”string”/>
    </AttributeAssignment>
</Obligation>
```

Listing B.4: Example usage of the extended `<Obligation>` element. Namespaces are not shown in full for readability reasons.

# Appendix C

# Correctness of the policy transformations of Chapter 5

This appendix proves the correctness of the policy transformations presented in Chapter 5 by means of their truth tables. Before that, we provide the decision tables of the combination algorithms PermitOverrides, DenyOverrides and FirstApplicable for clarity.

## C.1 Combination algorithms

In this section, we describe the behavior of the combination algorithms PermitOverrides, DenyOverrides and FirstApplicable by means of their decision tables. Notice that the decision tables are shown for two children. This does not limit the general applicability of these decision tables as any combination of more than two children has an equivalent binary representation, as shown by transformations T4, T5 and T6 of Chapter 5.

### C.1.1 PermitOverrides

PermitOverrides evaluates to Permit if one of its children returns a Permit. If not, it evaluates to Deny if one of its children returns Deny and NotApplicable otherwise. More specifically, the behavior of *PermitOverrides($P_A$,$P_B$)* is as follows:

| $P_A$ | $P_B$ | *PermitOverrides(*$P_A$*,*$P_B$*)* |
|---|---|---|
| Permit | Permit | Permit |
| Permit | Deny | Permit |
| Permit | NotApplicable | Permit |
| Deny | Permit | Permit |
| Deny | Deny | Deny |
| Deny | NotApplicable | Deny |
| NotApplicable | Permit | Permit |
| NotApplicable | Deny | Deny |
| NotApplicable | NotApplicable | NotApplicable |

### C.1.2 DenyOverrides

DenyOverrides evaluates to Deny if one of its children returns a Deny. If not, it evaluates to Permit if one of its children returns Permit and NotApplicable otherwise. More specifically, the behavior of *DenyOverrides(*$P_A$*,*$P_B$*)* is as follows:

| $P_A$ | $P_B$ | *DenyOverrides(*$P_A$*,*$P_B$*)* |
|---|---|---|
| Permit | Permit | Permit |
| Permit | Deny | Deny |
| Permit | NotApplicable | Permit |
| Deny | Permit | Deny |
| Deny | Deny | Deny |
| Deny | NotApplicable | Deny |
| NotApplicable | Permit | Permit |
| NotApplicable | Deny | Deny |
| NotApplicable | NotApplicable | NotApplicable |

### C.1.3 FirstApplicable

FirstApplicable returns the first Permit or Deny returned by a child when evaluating the children in the order in which they are given. If no child returns Permit or Deny, FirstApplicable returns NotApplicable. More specifically, the behavior of *FirstApplicable(*$P_A$*,*$P_B$*)* is as follows:

| $P_A$ | $P_B$ | *FirstApplicable(*$P_A$*,*$P_B$*)* |
|---|---|---|
| Permit | - | Permit |
| Deny | - | Deny |
| NotApplicable | Permit | Permit |
| NotApplicable | Deny | Deny |
| NotApplicable | NotApplicable | NotApplicable |

## C.2 Truth tables of the policy transformations

Following the behavior of the combination algorithms, we now prove the correctness of the policy transformations defined in Chapter 5 by means of their truth tables.

### C.2.1 Transformation T1

The left-hand side of transformation T1 is:

$$< T_1|T_2, CA, [P_1...P_n] >$$

The right-hand side of transformation T1 is:

$$< true, FirstApplicable, [< T_1, CA, [P_1...P_n] >, < T_2, CA, [P_1...P_n] >] >$$

Let *r* be the result of applying combination algorithm $CA$ to $[P_1...P_n]$, then both sides behave as follows, thus proving the correctness of this transformation:

| $T_1$ | $T_2$ | Result |
|---|---|---|
| True | True | *r* |
| True | False | *r* |
| False | True | *r* |
| False | False | NotApplicable |

### C.2.2 Transformation T2

The left-hand side of transformation T2 is:

$$< Permit, C_1|C_2 >$$

The right-hand side of transformation T2 is:

$$< true, PermitOverrides, [< Permit, C_1 >, < Permit, C_2 >] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $C_1$ | $C_2$ | Result |
|---|---|---|
| True | True | Permit |
| True | False | Permit |
| False | True | Permit |
| False | False | NotApplicable |

### C.2.3 Transformation T3

The left-hand side of transformation T3 is:

$$< Deny, C_1|C_2 >$$

The right-hand side of transformation T3 is:

$$< true, DenyOverrides, [< Deny, C_1 >, < Deny, C_2 >] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $C_1$ | $C_2$ | Result |
|---|---|---|
| True | True | Deny |
| True | False | Deny |
| False | True | Deny |
| False | False | NotApplicable |

### C.2.4 Transformation T4

The left-hand side of transformation T4 is:

$$< T, PermitOverrides, [P_1, P_2, P_3] >$$

The right-hand side of transformation T4 is:

$$< T, PermitOverrides, [< true, PermitOverrides, [P_1, P_2] >, P_3] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $T$ | $P_1$ | $P_2$ | $P_3$ | Result |
|---|---|---|---|---|
| False | - | - | - | NotApplicable |
| True | Permit | - | - | Permit |
| True | - | Permit | - | Permit |
| True | - | - | Permit | Permit |
| True | Deny | Deny | Deny | Deny |
| True | Deny | Deny | NotApplicable | Deny |
| True | Deny | NotApplicable | Deny | Deny |
| True | Deny | NotApplicable | NotApplicable | Deny |
| True | NotApplicable | Deny | Deny | Deny |
| True | NotApplicable | Deny | NotApplicable | Deny |
| True | NotApplicable | NotApplicable | Deny | Deny |
| True | NotApplicable | NotApplicable | NotApplicable | NotApplicable |

### C.2.5 Transformation T5

The left-hand side of transformation T5 is:

$$< T, DenyOverrides, [P_1, P_2, P_3] >$$

The right-hand side of transformation T5 is:

$$< T, DenyOverrides, [< true, DenyOverrides, [P_1, P_2] >, P_3] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $T$ | $P_1$ | $P_2$ | $P_3$ | Result |
|---|---|---|---|---|
| False | - | - | - | NotApplicable |
| True | Deny | - | - | Deny |
| True | - | Deny | - | Deny |
| True | - | - | Deny | Deny |
| True | Permit | Permit | Permit | Permit |
| True | Permit | Permit | NotApplicable | Permit |
| True | Permit | NotApplicable | Permit | Permit |
| True | Permit | NotApplicable | NotApplicable | Permit |
| True | NotApplicable | Permit | Permit | Permit |
| True | NotApplicable | Permit | NotApplicable | Permit |
| True | NotApplicable | NotApplicable | Permit | Permit |
| True | NotApplicable | NotApplicable | NotApplicable | NotApplicable |

### C.2.6 Transformation T6

The left-hand side of transformation T6 is:

$$< T, FirstApplicable, [P_1, P_2, P_3] >$$

The right-hand side of transformation T6 is:

$$< T, FirstApplicable, [< true, FirstApplicable, [P_1, P_2] >, P_3] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $T$ | $P_1$ | $P_2$ | $P_3$ | Result |
|---|---|---|---|---|
| False | - | - | - | NotApplicable |
| True | Permit | - | - | Permit |
| True | Deny | - | - | Deny |
| True | NotApplicable | Permit | - | Permit |
| True | NotApplicable | Deny | - | Deny |
| True | NotApplicable | NotApplicable | Permit | Permit |
| True | NotApplicable | NotApplicable | Deny | Deny |
| True | NotApplicable | NotApplicable | NotApplicable | NotApplicable |

### C.2.7 Transformation T7

The left-hand side of transformation T7 is:

$$< T, PermitOverrides, [P_1, P_2] >$$

The right-hand side of transformation T7 is:

$$< T, PermitOverrides, [P_2, P_1] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $T$ | $P_1$ | $P_2$ | Result |
|---|---|---|---|
| False | - | - | NotApplicable |
| True | Permit | Permit | Permit |
| True | Permit | Deny | Permit |
| True | Deny | Permit | Permit |
| True | Deny | Deny | Deny |

### C.2.8 Transformation T8

The left-hand side of transformation T8 is:

$$< T, DenyOverrides, [P_1, P_2] >$$

The right-hand side of transformation T8 is:

$$< T, DenyOverrides, [P_2, P_1] >$$

Both sides behave as follows, thus proving the correctness of this transformation:

| $T$ | $P_1$ | $P_2$ | Result |
|---|---|---|---|
| False | - | - | NotApplicable |
| True | Permit | Permit | Permit |
| True | Permit | Deny | Deny |
| True | Deny | Permit | Deny |
| True | Deny | Deny | Deny |

# Appendix D

# Overview of the developed prototypes

A key characteristic of our research approach is that we systematically evaluated our contributions based on prototypes. As a result, we developed a prototype of each of our contributions, of which the code is open-source and available on-line:

1. *Amusa.* The code and a live demo of the prototype of our Amusa middleware and the eDocs application running on top of this middleware are available at `https://distrinet.cs.kuleuven.be/software/amusa/`. These prototypes are written in Java, employ the Spring 3 Web MVC framework for the front-ends and employ the SunXACML engine and our STAPL engine for policy evaluation. More information is given in Section 3.4.2.

2. *Federated authorization.* The code of our prototype of federated authorization is available at `http://people.cs.kuleuven.be/~maarten.decat/doa-trusted-cloud-2013/`. This prototype is written in Java and extends the SunXACML policy evaluation engine. More information is given in Section 4.4.1.

3. *Policy federation.* The code of our prototypes of the algorithm for policy federation and the supporting middleware is available on-line at `http://people.cs.kuleuven.be/~maarten.decat/jisa2013/`. These prototypes are both written in Java and extend the SunXACML policy evaluation engine. More information is given in Section 5.5.1.

4. *Concurrent evaluation of access control policies.* The code of our prototype of our concurrency control scheme for secure concurrent evaluation of access

control policies is available at `http://people.cs.kuleuven.be/~maarten.decat/acsac2015/`. This prototype builds upon our STAPL language, is written in Scala and employs the Akka actor framework for concurrency and distributed communication. More information is given in Section 6.4.1.

In addition, we also developed an extensive prototype of our STAPL language:

5. *STAPL.* The code of our STAPL language is available at `https://github.com/stapl-dsl/`. STAPL is defined as an internal DSL in Scala and is supported by an efficient policy evaluation engine. More information is given in Section 2.3.1.

# Bibliography

[1] Health Insurance Portability and Accountability Act.

[2] Security Assertion Markup Language (SAML) v2.0. http://www.oasis-open.org/standards#samlv2.0, March 2005.

[3] A Guide to Claims-based Identity and Access Control. http://msdn.microsoft.com/en-us/library/ff423674.aspx, January 2010.

[4] 3-D Secure - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/3-D_Secure, July 2013.

[5] eXtensible Access Control Markup Language (XACML) Version 3.0. *OASIS Standard* (2013).

[6] OpenID Authentication 2.0 - Final. http://openid.net/specs/openid-authentication-2_0.html, May 2013.

[7] Sun's XACML Implementation. http://sourceforge.net/projects/sunxacml/, August 2013.

[8] A next generation federated identity management solution for a regional ecosystem of media providers (Media ID). https://distrinet.cs.kuleuven.be/research/projects/MediaID, June 2014.

[9] Chapter 3. Defining Authorization Policies. http://docs.forgerock.org/en/openam/12.0.0/admin-guide/index/chap-authz-policy.html, June 2014.

[10] Creating an XACML Policy - Identity Server 4.5.0 - WSO2 Documentation. https://docs.wso2.com/display/IS450/Creating+an+XACML+Policy, June 2014.

[11] E-Health Information Platforms (E-HIP). http://distrinet.cs.kuleuven.be/research/projects/E-HIP, June 2014.

[12] Healthcare professional's collaboration Space (Share4Health). http://distrinet.cs.kuleuven.be/research/projects/Share4Health, June 2014.

[13] IDC Predicts SaaS Enterprise Applications Will Be A $50.8B Market By 2018. http://www.forbes.com/sites/louiscolumbus/2014/12/20/idc-predicts-saas-enterprise-applications-will-be-a-50-8b\-market-by-2018/, December 2014.

[14] Namespaces Java API - Google App Engine - Google Developers. https://developers.google.com/appengine/docs/java/multitenancy/, May 2014.

[15] OCareCloudS - Overview projects - iMinds. http://www.iminds.be/en/research/overview-projects/p/detail/ocareclouds-2, June 2014.

[16] Permission, User Management and Availability for multi-tenant SaaS applications (PUMA). http://distrinet.cs.kuleuven.be/research/projects/PUMA, June 2014.

[17] Smart Plug-in Automobile Renewable Charging Services (SPARC). https://distrinet.cs.kuleuven.be/research/projects/SPARC, June 2014.

[18] Spring Security - Expression-Based Access Control. http://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html, May 2014.

[19] App Engine - Run your applications on a fully-managed Platform-as-a-Service (PaaS) using built-in services - Google Cloud Platform. https://-cloud.google.com/appengine/, July 2015.

[20] AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting. http://aws.amazon.com/ec2, July 2015.

[21] CRM Software & Cloud Computing Solutions. http://www.salesforce.com, July 2015.

[22] Google Drive - Cloud Storage & File Backup for Photos, Docs & More. https://www.google.com/drive/, July 2015.

[23] Home - WG - User Managed Access - Kantara Initiative. https://kantarainitiative.org/confluence/display/uma/Home, August 2015.

[24] OpenAz Main Page - OpenLiberty.org Wiki. http://www.openliberty.org/wiki/index.php/OpenAz_Main_Page, August 2015.

[25] SELinux Wiki. `http://selinuxproject.org/`, October 2015.

[26] SEQUOIA - iMinds. `https://www.iminds.be/en/projects/2015/03/11/sequoia`, March 2015.

[27] Spring Security. `http://projects.spring.io/spring-security/`, June 2015.

[28] Alam, M., Zhang, X., Khan, K., and Ali, G. xDAuth: A Scalable and Lightweight Framework for Cross Domain Access Control and Delegation. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2011), SACMAT '11, ACM, pp. 31–40.

[29] Alcaraz Calero, J., Edwards, N., Kirschnick, J., Wilcock, L., and Wray, M. Toward a Multi-Tenancy Authorization System for Cloud Services. *Security Privacy, IEEE 8*, 6 (Nov 2010), 48–55.

[30] Alfieri, R., Cecchini, R., Ciaschini, V., Frohner, Á., Lorentey, K., Spataro, F., et al. From gridmap-file to VOMS: managing authorization in a Grid environment. *Future Generation Computer Systems 21*, 4 (2005), 549–558.

[31] Alzahrani, A., Janicke, H., and Abubaker, S. Decentralized XACML Overlay Network. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (June 2010), pp. 1032–1037.

[32] Ardagna, C., De Capitani di Vimercati, S., Foresti, S., Neven, G., Paraboschi, S., Preiss, F.-S., Samarati, P., and Verdicchio, M. Fine-Grained Disclosure of Access Policies. 16–30.

[33] Ardagna, C., De Capitani di Vimercati, S., Neven, G., Paraboschi, S., Preiss, F.-S., Samarati, P., and Verdicchio, M. Enabling Privacy-preserving Credential-based Access Control with XACML and SAML. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (June 2010), pp. 1090–1095.

[34] Asghar, M., Ion, M., Russello, G., and Crispo, B. ESPOON: Enforcing Encrypted Security Policies in Outsourced Environments. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on* (Aug 2011), pp. 99–108.

[35] Ashley, P., Hada, S., Karjoth, G., Powers, C., and Schunter, M. Enterprise privacy authorization language (EPAL). Tech. rep., IBM, 2003.

[36] Bacon, J., Evans, D., Eyers, D. M., Migliavacca, M., Pietzuch, P., and Shand, B. Enforcing End-to-end Application Security in the Cloud (Big Ideas Paper). In *Proceedings of the ACM/IFIP/USENIX 11th International Conference*

*on Middleware*, Middleware '10. Springer-Verlag, Berlin, Heidelberg, 2010, pp. 293–312.

[37] Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., Hondo, M., Kaler, C., Langworthy, D., Malhotra, A., et al. Web Services Policy Framework (WS-Policy) . http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html, March 2006.

[38] Bajaj, S., Della-Libera, G., Dixon, B., Dusche, M., Hondo, M., Hur, M., Kaler, C., Lockhart, H., Maruyama, H., Nadalin, A., et al. Web Services Federation Language (WS-Federation). http://specs.xmlsoap.org/ws/2006/12/federation, December 2006.

[39] Bates, A., Mood, B., Valafar, M., and Butler, K. Towards Secure Provenance-based Access Control in Cloud Environments. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2013), CODASPY '13, ACM, pp. 277–284.

[40] Bauer, L., Garriss, S., and Reiter, M. Distributed proving in access-control systems. In *Security and Privacy, 2005 IEEE Symposium on* (May 2005), pp. 81–95.

[41] Bauer, L., Garriss, S., and Reiter, M. Efficient Proving for Practical Distributed Access-Control Systems. In *Computer Security - ESORICS 2007*, J. Biskup and J. Lopez, Eds., vol. 4734 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 19–37.

[42] Becker, M. Y., Fournet, C., and Gordon, A. D. SecPAL: Design and Semantics of a Decentralized Authorization Language. *J. Comput. Secur. 18*, 4 (Dec. 2010), 619–665.

[43] Becker, M. Y., and Sewell, P. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations* (Washington, DC, USA, 2004), CSFW '04, IEEE Computer Society, pp. 139–.

[44] Bell, D. E., and LaPadula, L. J. Secure computer systems: Mathematical foundations. Tech. rep., DTIC Document, 1973.

[45] Bertino, E., Bonatti, P. A., and Ferrari, E. TRBAC: A Temporal Role-based Access Control Model. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug. 2001), 191–233.

[46] Bertino, E., Catania, B., Damiani, M. L., and Perlasca, P. GEO-RBAC: A Spatially Aware RBAC. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2005), SACMAT '05, ACM, pp. 29–37.

[47] Bezemer, C.-P., Zaidman, A., Platzbeecker, B., Hurkmans, T., and 't Hart, A. Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on* (Sept 2010), pp. 1–8.

[48] Beznosov, K. K. Flooding and recycling authorizations. In *Proceedings of the 2005 workshop on New security paradigms* (2005), ACM, pp. 67–72.

[49] Biba, K. J. Integrity considerations for secure computer systems. Tech. rep., DTIC Document, 1977.

[50] Blaze, M., Feigenbaum, J., and Lacy, J. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (May 1996), pp. 164–173.

[51] Boehm, O., Caumanns, J., Franke, M., and Pfaff, O. Federated Authentication and Authorization: A Case Study. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE* (Sept 2008), pp. 356–362.

[52] Bogaerts, J., Decat, M., Lagaisse, B., and Joosen, W. Entity-based access control: supporting more expressive access control policies. In *To be published in the Proceedings of the 31th Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC '15, ACM.

[53] Bonatti, P., de Capitani di Vimercati, S., and Samarati, P. A Modular Approach to Composing Access Control Policies. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2000), CCS '00, ACM, pp. 164–173.

[54] Bonatti, P., De Capitani di Vimercati, S., and Samarati, P. An Algebra for Composing Access Control Policies. *ACM Trans. Inf. Syst. Secur. 5*, 1 (Feb. 2002), 1–35.

[55] Borders, K., Zhao, X., and Prakash, A. CPOL: High-performance Policy Evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 147–157.

[56] Brewer, D., and Nash, M. The Chinese Wall security policy. In *IEEE Security and Privacy* (May 1989), pp. 206–214.

[57] Brossard, D. Using ALFA Eclipse plugin to author XACML policies - Part 1. `http://developers.axiomatics.com/blog/index/entry/using-alfa-eclipse-plugin-to-author-xacml-policies-part-1.html`, February 2014.

[58] Brucker, A., and Petritsch, H. Idea: Efficient Evaluation of Access Control Constraints. 157–165.

[59] Butler, B., Jennings, B., and Botvich, D. XACML Policy Performance Evaluation Using a Flexible Load Testing Framework. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 648–650.

[60] Butler, B., Jennings, B., and Botvich, D. An experimental testbed to predict the performance of XACML Policy Decision Points. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on* (May 2011), pp. 353–360.

[61] Carrie, D., and Gates, E. Access Control Requirements for Web 2.0 Security and Privacy. In *Proc. of Workshop on Web 2.0 Security & Privacy (W2SP 2007* (2007).

[62] Chadwick, D. Coordinated Decision Making in Distributed Applications. *Inf. Secur. Tech. Rep. 12*, 3 (June 2007), 147–154.

[63] Chadwick, D., Su, L., Otenko, O., and Laborde, R. Coordination between distributed PDPs. In *IEEE POLICY* (June 2006).

[64] Chadwick, D. W., and Otenko, A. The PERMIS X.509 Role Based Privilege Management Infrastructure. 135–140.

[65] Chakraborty, S., and Ray, I. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *SACMAT* (2006), ACM, pp. 49–58.

[66] Cheng, Y., Park, J., and Sandhu, R. A User-to-User Relationship-Based Access Control Model for Online Social Networks. In *Data and Applications Security and Privacy XXVI*, N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro, Eds., vol. 7371 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 8–24.

[67] Chong, F., and Carraro, G. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation* (2006), 9–10.

[68] Cole, G. Service Provisioning Markup Language (SPML) Version 2.0. *OASIS Committee Specification* (2006).

[69] Colombo, M., Lazouski, A., Martinelli, F., and Mori, P. A Proposal on Enhancing XACML with Continuous Usage Control Features. 133–146.

[70] Colombo, M., Lazouski, A., Martinelli, F., and Mori, P. Access and Usage Control in Grid Systems. In *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Springer Berlin Heidelberg, 2010, pp. 293–308.

[71] Commision, E. Directive 95/46/EC , 1995. Directive of the European Parliament and of the Council of 24 Oct. 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data.

[72] Coulouris, G. F., Dollimore, J., and Kindberg, T. *Distributed systems: concepts and design.* pearson education, 2005.

[73] Crampton, J. A reference monitor for workflow systems with constrained task execution. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2005), SACMAT '05, ACM, pp. 38–47.

[74] Crampton, J., and Huth, M. An Authorization Framework Resilient to Policy Evaluation Failures. 472–487.

[75] Crampton, J., and Khambhammettu, H. Delegation in role-based access control. *International Journal of Information Security 7*, 2 (2008), 123–136.

[76] Crampton, J., Leung, W., and Beznosov, K. The Secondary and Approximate Authorization Model and Its Application to Bell-LaPadula Policies. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2006), SACMAT '06, ACM, pp. 111–120.

[77] Crampton, J., and Morisset, C. PTaCL: A Language for Attribute-Based Access Control in Open Systems. In *Principles of Security and Trust*, P. Degano and J. Guttman, Eds., vol. 7215 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 390–409.

[78] Crampton, J., and Sellwood, J. Path Conditions and Principal Matching: A New Approach to Access Control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2014), SACMAT '14, ACM, pp. 187–198.

[79] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The Ponder Policy Specification Language. 18–38.

[80] De Win, B., Piessens, F., Joosen, W., and Verhanneman, T. On the importance of the separation-of-concerns principle in secure software engineering. In *Proceedings of the ACSA Workshop on the Application of Engineering Principles to System Security Design* (2002).

[81] Decat, M., Bogaerts, J., Lagaisse, B., and Joosen, W. The e-document case study: functional analysis and access control requirements. Technical report, KU Leuven, 2014.

[82] Decat, M., Bogaerts, J., Lagaisse, B., and Joosen, W. The workforce management case study: functional analysis and access control requirements. Technical report, KU Leuven, 2014.

[83] Decat, M., Bogaerts, J., Lagaisse, B., and Joosen, W. Amusa: Middleware for Efficient Access Control Management of Multi-tenant SaaS Applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2015), SAC '15, ACM.

[84] Decat, M., Lagaisse, B., and Joosen, W. Toward efficient and confidentiality-aware federation of access control policies. In *Workshop on Middleware for Next Generation Internet Computing* (2012), ACM.

[85] Decat, M., Lagaisse, B., and Joosen, W. Middleware for efficient and confidentiality-aware federation of access control policies. *Journal of Internet Services and Applications* (2014).

[86] Decat, M., Lagaisse, B., and Joosen, W. Scalable and secure concurrent evaluation of history-based access control policies. In *To be published in the Proceedings of the 31th Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC '15, ACM.

[87] Decat, M., Lagaisse, B., Joosen, W., and Crispo, B. Introducing Concurrency in Policy-based Access Control. In *Proceedings of the 8th Workshop on Middleware for Next Generation Internet Computing* (New York, NY, USA, 2013), MW4NextGen '13, ACM, pp. 3:1–3:6.

[88] Decat, M., Lagaisse, B., Van Landuyt, D., Crispo, B., and Joosen, W. Federated Authorization for Software-as-a-Service Applications. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, and D. Dou, Eds., vol. 8185 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 342–359.

[89] Decat, M., Moeys, J., Lagaisse, B., and Joosen, W. Improving Reuse of Attribute-Based Access Control Policies Using Policy Templates. In *Engineering Secure Software and Systems*, F. Piessens, J. Caballero, and N. Bielova, Eds., vol. 8978 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 196–210.

[90] Decat, M., Van Landuyt, D., Lagaisse, B., and Joosen, W. On the need for federated authorization in cross-organizational e-health platforms. In *Proceedings of the 8the international conference on Health Informatics* (2014), HealthInf 2015, pp. 540–546.

[91] Deng, M., Wuyts, K., Scandariato, R., Preneel, B., and Joosen, W. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering 16*, 1 (2011), 3–32.

[92] Dhankhar, V., Kaushik, S., Wijesekera, D., and Nerode, A. Evaluating Distributed Xacml Policies. In *Proceedings of the 2007 ACM Workshop on Secure Web Services* (New York, NY, USA, 2007), SWS '07, ACM, pp. 99–110.

[93] di Vimercati, S. D. C., Foresti, S., Jajodia, S., Paraboschi, S., and Samarati, P. A Data Outsourcing Architecture Combining Cryptography and Access Control. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 63–69.

[94] Djordjevic, I., and Dimitrakos, T. A note on the anatomy of federation. *BT Technology Journal 23*, 4 (2005), 89–106.

[95] Edjlali, G., Acharya, A., and Chaudhary, V. History-based Access Control for Mobile Code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security* (New York, NY, USA, 1998), CCS '98, ACM, pp. 38–48.

[96] El Kateb, D., Mouelhi, T., Le Traon, Y., Hwang, J., and Xie, T. Refactoring Access Control Policies for Performance Improvement. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2012), ICPE '12, ACM, pp. 323–334.

[97] Elmasri, R. A., and Navathe, S. B. *Fundamentals of Database Systems*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[98] Erlingsson, Ú. The inlined reference monitor approach to security policy enforcement. Tech. rep., Cornell University, 2004.

[99] Fatema, K., Chadwick, D., and Lievens, S. A Multi-privacy Policy Enforcement System. In *Privacy and Identity Management for Life*, S. Fischer-Hübner, P. Duquenoy, M. Hansen, R. Leenes, and G. Zhang, Eds., vol. 352 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2011, pp. 297–310.

[100] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. Proposed NIST Standard for Role-based Access Control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug. 2001), 224–274.

[101] Filman, R., Elrad, T., Clarke, S., and Akşit, M. *Aspect-oriented Software Development*, first ed. Addison-Wesley Professional, 2004.

[102] Fischer, J., Marino, D., Majumdar, R., and Millstein, T. Fine-Grained Access Control with Object-Sensitive Roles. In *ECOOP 2009 - Object-Oriented Programming*, S. Drossopoulou, Ed., vol. 5653 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 173–194.

[103] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. Verification and Change-impact Analysis of Access-control Policies. In *Proceedings of the 27th International Conference on Software Engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 196–205.

[104] Fong, P. W. Relationship-based Access Control: Protection Model and Policy Language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2011), CODASPY '11, ACM, pp. 191–202.

[105] Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security* (New York, NY, USA, 1998), CCS '98, ACM, pp. 83–92.

[106] Foster, I., Kesselman, C., and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl. 15*, 3 (Aug. 2001), 200–222.

[107] Fotiou, N., Machas, A., Polyzos, G., and Xylomenos, G. Access control as a service for the Cloud. *Journal of Internet Services and Applications 6*, 1 (2015).

[108] Freudenthal, E., Pesin, T., Port, L., Keenan, E., and Karamcheti, V. dRBAC: distributed role-based access control for dynamic coalition environments. In *DSS* (2002), pp. 411–420.

[109] Fuchs, L., Pernul, G., and Sandhu, R. Roles in information security - A survey and classification of the research area. *Computers & Security 30*, 8 (2011), 748 – 769.

[110] Gama, P., Ribeiro, C., and Ferreira, P. A scalable history-based policy engine. In *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on* (June 2006), pp. 10 pp.–112.

[111] Gay, R., Mantel, H., and Sprick, B. Service Automata. In *Formal Aspects of Security and Trust*, G. Barthe, A. Datta, and S. Etalle, Eds., vol. 7140 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 148–163.

[112] Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2009), STOC '09, ACM, pp. 169–178.

[113] Gheorghe, G., Crispo, B., Carbone, R., Desmet, L., and Joosen, W. Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement. 350–369.

[114] Giunchiglia, F., Crispo, B., and Zhang, R. Access control via lightweight ontologies. In *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on* (Sept 2011), pp. 352–355.

[115] Giunchiglia, F., Zhang, R., and Crispo, B. RelBAC: Relation Based Access Control. In *Semantics, Knowledge and Grid, 2008. SKG '08. Fourth International Conference on* (Dec 2008), pp. 3–11.

[116] Giuri, L., and Iglio, P. Role Templates for Content-based Access Control. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (New York, NY, USA, 1997), RBAC '97, ACM, pp. 153–159.

[117] Godik, S., Moses, T., et al. eXtensible Access Control Markup Language (XACML) 1.0. *OASIS Standard* (2003).

[118] Griffin, L., Butler, B., de Leastar, E., Jennings, B., and Botvich, D. On the Performance of Access Control Policy Evaluation. In *Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on* (July 2012), pp. 25–32.

[119] Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. A Framework for Native Multi-Tenancy Application Development and Management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on* (July 2007), pp. 551–558.

[120] Hachem, S., Toninelli, A., Pathak, A., and Issarny, V. Policy-Based Access Control in Mobile Social Ecosystems. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on* (June 2011), pp. 57–64.

[121] Han, W., and Lei, C. A Survey on Policy Languages in Network and Security Management. *Comput. Netw. 56*, 1 (Jan. 2012), 477–489.

[122] Hardt, D. The OAuth 2.0 authorization framework.

[123] Hu, V., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., and Scarfone, K. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication* (2014).

[124] Jahid, S., Gunter, C. A., Hoque, I., and Okhravi, H. MyABDAC: Compiling XACML Policies for Attribute-based Database Access Control. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2011), CODASPY '11, ACM, pp. 97–108.

[125] Janicke, H., Cau, A., Siewe, F., and Zedan, H. Concurrent Enforcement of Usage Control Policies. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on* (June 2008), pp. 111–118.

[126] Jie, W., Arshad, J., Sinnott, R., Townend, P., and Lei, Z. A Review of Grid Authentication and Authorization Technologies and Support for Federated Access Control. *ACM Comput. Surv. 43*, 2 (Feb. 2011), 12:1–12:26.

[127] Jin, X., Krishnan, R., and Sandhu, R. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI*, N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro, Eds., vol. 7371 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 41–55.

[128] Jin, X., Sandhu, R., and Krishnan, R. RABAC: Role-Centric Attribute-Based Access Control. In *Computer Network Security*, I. Kotenko and V. Skormin, Eds., vol. 7531 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 84–96.

[129] Karjoth, G. Access Control with IBM Tivoli Access Manager. *ACM Trans. Inf. Syst. Secur. 6*, 2 (May 2003), 232–257.

[130] Kelbert, F., and Pretschner, A. A Fully Decentralized Data Usage Control Enforcement Infrastructure. In *To appear in Proc. 13th International Conference on Applied Cryptography and Network Security (ACNS)*. 2015.

[131] Kini, P., and Beznosov, K. Speculative Authorization. *Parallel and Distributed Systems, IEEE Transactions on 24*, 4 (April 2013), 814–824.

[132] Kohler, M., Brucker, A., and Schaad, A. ProActive Caching: Generating Caching Heuristics for Business Process Environments. In *Computational Science and Engineering, 2009. CSE '09. International Conference on* (Aug 2009), vol. 3, pp. 297–304.

[133] Kohler, M., and Brucker, A. D. Access Control Caching Strategies: An Empirical Evaluation. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics* (New York, NY, USA, 2010), MetriSec '10, ACM, pp. 8:1–8:8.

[134] Kohler, M., and Schaad, A. Proactive access control for business process-driven environments. In *2008 Annual Computer Security Applications Conference* (2008), IEEE, pp. 153–162.

[135] Kolovski, V., Hendler, J., and Parsia, B. Analyzing Web Access Control Policies. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 677–686.

[136] Komlenovic, M., Tripunitara, M., and Zitouni, T. An Empirical Assessment of Approaches to Distributed Enforcement in Role-based Access Control (RBAC). In *Proceedings of the First ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2011), CODASPY '11, ACM, pp. 121–132.

[137] Kremer, S., Markowitch, O., and Zhou, J. An Intensive Survey of Fair Non-repudiation Protocols. *Comput. Commun. 25*, 17 (Nov. 2002), 1606–1621.

[138] Kuhlmann, M., Shohat, D., and Schimpf, G. Role Mining - Revealing Business Roles for Security Administration Using Data Mining Technology. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2003), SACMAT '03, ACM, pp. 179–186.

[139] Kuhn, D. R., Coyne, E. J., and Weil, T. R. Adding attributes to role-based access control. *Computer*, 6 (2010), 79–81.

[140] Kumar, V., Cooper, B., Eisenhauer, G., and Schwan, K. iManage: Policy-Driven Self-management for Enterprise-Scale Systems. In *Middleware 2007* (2007), R. Cerqueira and R. Campbell, Eds., vol. 4834 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 287–307.

[141] Lagaisse, B., Joosen, W., and De Win, B. Managing semantic interference with aspect integration contracts. In *International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)* (2004).

[142] Lampson, B. W. Protection. *SIGOPS Oper. Syst. Rev. 8* (January 1974), 18–24.

[143] Latham, D. Department of Defense Trusted Computer System Evaluation Criteria. Tech. rep., US Department of Defense, 1985.

[144] Lawrence, K., Kaler, C., Nadalin, A., Monzillo, R., and Hallam-Baker, P. Web Services Security: SOAP Message Security 1.1 (WS-Security), 2006.

[145] Lazouski, A., Mancini, G., Martinelli, F., and Mori, P. Usage control in cloud systems. In *Internet Technology And Secured Transactions, 2012 International Conference for* (Dec 2012), pp. 202–207.

[146] Lazouski, A., Martinelli, F., and Mori, P. Usage control in computer security: A survey. *Computer Science Review 4*, 2 (2010), 81 – 99.

[147] Lepro, R. Cardea: Dynamic access control in distributed systems. *SYSTEM 3*, 4 (2003).

[148] Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., and Lin, D. Access Control Policy Combining: Theory Meets Practice. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2009), SACMAT '09, ACM, pp. 135–144.

[149] Lin, D., Rao, P., Bertino, E., Li, N., and Lobo, J. Policy Decomposition for Collaborative Access Control. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2008), SACMAT '08, ACM, pp. 103–112.

[150] Lischka, M., Endo, Y., and Sánchez Cuenca, M. Deductive Policies with XACML. In *Proceedings of the 2009 ACM Workshop on Secure Web Services* (New York, NY, USA, 2009), SWS '09, ACM, pp. 37–44.

[151] Liu, A., Chen, F., Hwang, J., and Xie, T. Designing Fast and Scalable XACML Policy Evaluation Engines. *Computers, IEEE Transactions on 60*, 12 (Dec 2011), 1802–1817.

[152] Liu, A. X., Chen, F., Hwang, J., and Xie, T. Xengine: A Fast and Scalable XACML Policy Evaluation Engine. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2008), SIGMETRICS '08, ACM, pp. 265–276.

[153] Lobo, J., Ma, J., Russo, A., Lupu, E., Calo, S., and Sloman, M. Refinement of History-Based Policies. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, M. Balduccini and T. Son, Eds., vol. 6565 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2011, pp. 280–299.

[154] Lockhart, H., Parducci, B., and Rissanen, E. SAML 2.0 Profile of XACML, Version 2.0.

[155] Lodderstedt, T., Basin, D., and Doser, J. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *UML 2002 - The Unified Modeling Language*, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds., vol. 2460 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2002, pp. 426–441.

[156] Lorch, M., Adams, D. B., Kafura, D., Koneni, M. S. R., Rathi, A., and Shah, S. The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments. In *Proceedings of the 4th International Workshop on Grid Computing* (Washington, DC, USA, 2003), GRID '03, IEEE Computer Society, pp. 109–.

[157] Lorch, M., Proctor, S., Lepro, R., Kafura, D., and Shah, S. First Experiences Using XACML for Access Control in Distributed Systems. In *Proceedings of the 2003 ACM Workshop on XML Security* (New York, NY, USA, 2003), XMLSEC '03, ACM, pp. 25–37.

[158] Marouf, S., Shehab, M., Squicciarini, A., and Sundareswaran, S. Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation. In *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on* (July 2009), pp. 118–125.

[159] Marouf, S., Shehab, M., Squicciarini, A., and Sundareswaran, S. Adaptive Reordering and Clustering-Based Framework for Efficient XACML Policy Evaluation. *Services Computing, IEEE Transactions on 4*, 4 (Oct 2011), 300–313.

[160] Mazzoleni, P., Crispo, B., Sivasubramanian, S., and Bertino, E. XACML Policy Integration Algorithms. *ACM Trans. Inf. Syst. Secur. 11*, 1 (Feb. 2008), 4:1–4:29.

[161] Mell, P., and Grance, T. The NIST definition of cloud computing.

[162] Minami, K., and Kotz, D. Secure context-sensitive authorization. *Pervasive and Mobile Computing 1*, 1 (2005), 123 – 156.

[163] Minami, K., and Kotz, D. Scalability in a Secure Distributed Proof System. In *Pervasive Computing*, K. Fishkin, B. Schiele, P. Nixon, and A. Quigley, Eds., vol. 3968 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 220–237.

[164] Miseldine, P. L. Automated XACML Policy Reconfiguration for Evaluation Optimisation. In *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems* (New York, NY, USA, 2008), SESS '08, ACM, pp. 1–8.

[165] Moses, T., et al. eXtensible Access Control Markup Language (XACML) 2.0. *OASIS Standard* (2005).

[166] Muthukumaran, D., Jaeger, T., and Ganapathy, V. Leveraging "Choice" to Automate Authorization Hook Placement. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 145–156.

[167] Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., and Granqvist, H. WS-Trust 1.3. http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html, Maart 2007.

[168] Neuman, B., and Ts'o, T. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE 32*, 9 (Sept 1994), 33–38.

[169] Nguyen, D., Park, J., and Sandhu, R. A provenance-based access control model for dynamic separation of duties. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on* (July 2013), pp. 247–256.

[170] Park, J., Nguyen, D., and Sandhu, R. A provenance-based access control model. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on* (July 2012), pp. 137–144.

[171] Park, J., and Sandhu, R. The UCONABC Usage Control Model. *ACM Trans. Inf. Syst. Secur. 7*, 1 (Feb. 2004), 128–174.

[172] Parnas, D. L. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.

[173] Pearlman, L., Welch, V., Foster, I., Kesselman, C., and Tuecke, S. A community authorization service for group collaboration. In *Policies for Distributed Systems and Networks, 2002. Proceedings. Third International Workshop on* (2002), pp. 50–59.

[174] Pina Ros, S., Lischka, M., and Gómez Mármol, F. Graph-based XACML Evaluation. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2012), SACMAT '12, ACM, pp. 83–92.

[175] Poortinga-van Wijnen, R., Hulsebosch, B., Reitsma, J., and Wegdam, M. Federated Authorisation and Group Management in e-Science.

[176] Rao, P., Lin, D., Bertino, E., Li, N., and Lobo, J. An Algebra for Fine-grained Integration of XACML Policies. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2009), SACMAT '09, ACM, pp. 63–72.

[177] Ribeiro, C., Zúquete, A., Ferreira, P., and Guedes, P. SPL: An access control language for security policies with complex constraints. In *In Proceedings of the Network and Distributed System Security Symposium* (1999), pp. 89–107.

[178] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. Extending Query Rewriting Techniques for Fine-grained Access Control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2004), SIGMOD '04, ACM, pp. 551–562.

[179] Sabelfeld, A., and Myers, A. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on 21*, 1 (Jan 2003), 5–19.

[180] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and Mortimore, C. Openid connect core 1.0. *The OpenID Foundation* (2014), S3.

[181] Samarati, P., and de Vimercati, S. Access Control: Policies, Models, and Mechanisms. In *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Eds., vol. 2171 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 137–196.

[182] Sandhu, R. The Authorization Leap from Rights to Attributes: Maturation or Chaos? SACMAT '12, ACM, pp. 69–70.

[183] Sandhu, R., Bhamidipati, V., and Munawer, Q. The ARBAC97 Model for Role-based Administration of Roles. *ACM Trans. Inf. Syst. Secur. 2*, 1 (Feb. 1999), 105–135.

[184] Sandhu, R., and Park, J. Usage Control: A Vision for Next Generation Access Control. In *Computer Network Security*, V. Gorodetsky, L. Popyack, and V. Skormin, Eds., vol. 2776 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 17–31. 10.1007/978-3-540-45215-7_2.

[185] Schaad, A., Moffett, J., and Jacob, J. The role-based access control system of a european bank: A case study and discussion. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2001), SACMAT '01, ACM, pp. 3–9.

[186] Schneider, F. B. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur. 3*, 1 (Feb. 2000), 30–50.

[187] Senk, C. Adoption of security as a service. *Journal of Internet Services and Applications 4*, 1 (2013).

[188] Shi, L., and Chadwick, D. W. A Controlled Natural Language Interface for Authoring Access Control Policies. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (New York, NY, USA, 2011), SAC '11, ACM, pp. 1524–1530.

[189] Sinnott, R., Chadwick, D., Doherty, T., Martin, D., Stell, A., Stewart, G., Su, L., and Watt, J. Advanced Security for Virtual Organizations: The Pros and Cons of Centralized vs Decentralized Security Models. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on* (May 2008), pp. 106–113.

[190] Sloman, M. Policy driven management for distributed systems. *Journal of Network and Systems Management 2*, 4 (1994), 333–360.

[191] Solo, D., Housley, R., and Ford, W. Internet X.509 public key infrastructure certificate and CRL profile.

[192] Sterne, D. On the buzzword 'security policy'. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on* (May 1991), pp. 219–230.

[193] Stihler, M., Santin, A., Calsavara, A., and Marcon, A. Distributed Usage Control Architecture for Business Coalitions. In *Communications, 2009. ICC '09. IEEE International Conference on* (June 2009), pp. 1–6.

[194] Su, L., Chadwick, D., Basden, A., and Cunningham, J. Automated decomposition of access control policies. In *Policies for Distributed Systems and Networks, 2005. Sixth IEEE International Workshop on* (June 2005), pp. 3–13.

[195] Sun, W., Zhang, X., Guo, C. J., Sun, P., and Su, H. Software as a Service: Configuration and Customization Perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE* (Sept 2008), pp. 18–25.

[196] Tang, B., Sandhu, R., and Li, Q. Multi-tenancy authorization models for collaborative cloud services. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on* (May 2013), pp. 132–138.

[197] Thompson, M., Essiari, A., Keahey, K., Welch, V., Lang, S., and Liu, B. Fine-Grained Authorization for Job and Resource Management Using Akenti and the Globus Toolkit. *Arxiv preprint cs/0306070* (2003).

[198] Tripunitara, M. V., and Carbunar, B. Efficient Access Enforcement in Distributed Role-based Access Control (RBAC) Deployments. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2009), SACMAT '09, ACM, pp. 155–164.

[199] Tsankov, P., Marinovic, S., Torabi D., M., and Basin, D. Fail-Secure Access Control. In *ACM CCS* (2014), pp. 1157–1168.

[200] Turkmen, F., and Crispo, B. Performance Evaluation of XACML PDP Implementations. In *Proceedings of the 2008 ACM Workshop on Secure Web Services* (New York, NY, USA, 2008), SWS '08, ACM, pp. 37–44.

[201] Verhanneman, T., Piessens, F., Win, B., and Joosen, W. Uniform application-level access control enforcement of organizationwide policies. In *Computer Security Applications Conference, 21st Annual* (Dec 2005), pp. 10 pp.–440.

[202] Walraven, S., Lagaisse, B., Truyen, E., and Joosen, W. Policy-driven customization of cross-organizational features in distributed service systems. *Software: Practice and Experience 43*, 10 (2013), 1145–1163.

[203] Wei, Q. *Towards improving the availability and performance of enterprise authorization systems.* PhD thesis, University of British Columbia, 2009.

[204] Wei, Q., Crampton, J., Beznosov, K., and Ripeanu, M. Authorization Recycling in RBAC Systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2008), SACMAT '08, ACM, pp. 63–72.

[205] Wei, Q., Ripeanu, M., and Beznosov, K. Cooperative Secondary Authorization Recycling. *Parallel and Distributed Systems, IEEE Transactions on 20*, 2 (Feb 2009), 275–288.

[206] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of*

*the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[207] Welch, V., Barton, T., Keahey, K., and Siebenlist, F. Attributes, Anonymity, and Access: Shibboleth and Globus Integration to Facilitate Grid Collaboration. In *In 4th Annual PKI R&D Workshop (To appear* (2005).

[208] Welch, V., Siebenlist, F., Foster, I., Bresnahan, J., Czajkowski, K., Gawor, J., Kesselman, C., Meder, S., Pearlman, L., and Tuecke, S. Security for grid services. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (June 2003), pp. 48–57.

[209] Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J., and Waldbusser, S. RFC 3198 - Terminology for Policy-Based Management. *The Internet Society, Network Working Group* (2001).

[210] Winsborough, W., Seamons, K., and Jones, V. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings* (2000), vol. 1, pp. 88–102 vol.1.

[211] Wun, A., and Jacobsen, H.-A. A Policy Management Framework for Content-based Publish/Subscribe Middleware. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, MIDDLEWARE2007. Springer-Verlag, Berlin, Heidelberg, 2007, pp. 368–388.

[212] Yuan, E., and Tong, J. Attributed based access control (ABAC) for Web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on* (July 2005).

[213] Zhang, X., Nakae, M., Covington, M. J., and Sandhu, R. Toward a Usage-Based Security Framework for Collaborative Computing Systems. *ACM Trans. Inf. Syst. Secur. 11*, 1 (Feb. 2008), 3:1–3:36.

# List of publications

## Journal papers

2014 M. Decat, B. Lagaisse and W. Joosen, Middleware for efficient and confidentiality-aware federation of access control policies, in Journal of Internet Services and Applications, February 2014

## International conference papers

2015 M. Decat, B. Lagaisse and W. Joosen, Scalable and secure concurrent evaluation of history-based access control policies, in Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC), December 2015

2015 J. Bogaerts, M. Decat, B. Lagaisse and W. Joosen, Entity-based access control: supporting more expressive access control policies, in Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC), December 2015

2015 M. Decat, J. Bogaerts, B. Lagaisse and W. Joosen, Amusa: middleware for efficient access control management of multi-tenant SaaS applications, in Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC), April 2015

2015 M. Decat, J. Moeys, B. Lagaisse and W. Joosen, Improving reuse of attribute-based access control policies using policy templates, in Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSOS), March 2015

2015 M. Decat, D. Van Landuyt, B. Lagaisse and W. Joosen, On the need for federated authorization in cross-organizational e-health platforms, in Proceedings of the 8the international conference on Health Informatics (HEALTHINF), January 2015

2014 P. Maenhaut, H. Moens, M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, V. Ongenae and F. De Turck, Characterizing the performance of tenant data management in multi-tenant cloud authorization systems, in Network Operations and Management Symposium (NOMS), May 2014

2013 M. Decat, B. Lagaisse, D. Van Landuyt, B. Crispo and W. Joosen, Federated authorization for Software-as-a-Service applications, in On the Move to Meaningful Internet Systems: OTM 2013 Conferences, September 2013

2010 P. De Ryck, M. Decat, L. Desmet, F. Piessens, W. Joosen, Security of web mashups: a survey, in Proceedings of the 15th Nordic Conference on Secure IT Systems (NordSec), October 2010

2010 M. Decat, P. De Ryck, L. Desmet, F. Piessens and W. Joosen, Towards building secure web mashups, in OWASP AppSec Research 2010, June 2010

## International workshop papers

2013 M. Decat, B. Lagaisse, W. Joosen and B. Crispo, Introducing concurrency in policy-based access control, in Proceedings of the 8th Workshop on Middleware for Next Generation Internet Computing (MW4NG), December 2013

2012 M. Decat, B. Lagaisse and W. Joosen, Toward efficient and confidentiality-aware federation of access control policies, in Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing(MW4NG), December 2012

## Technical reports

2014 M. Decat, J. Bogaerts, B. Lagaisse and W. Joosen, The e-document case study: functional analysis and access control requirements, volume CW654, Department of Computer Science, KU Leuven, February 2014

2014 M. Decat, J. Bogaerts, B. Lagaisse and W. Joosen, The workforce management case study: functional analysis and access control requirements, volume CW655, Department of Computer Science, KU Leuven, February 2014

## Book chapters

2014 A. Pathak, G. Rosca, V. Issarny, M. Decat and B. Lagaisse, Privacy and access control in federated social networks, in Engineering Secure Future Internet Services and Systems, pages 160-179, Springer International Publishing, 2014

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
http://www.cs.kuleuven.be