Toward efficient and confidentiality-aware federation of access control policies

Maarten Decat, Bert Lagaisse, Wouter Joosen
IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
first.last@cs.kuleuven.be

ABSTRACT

This paper presents our work in progress on efficient and confidentiality-aware access control for Software-as-a-Service applications. In SaaS, a tenant organization rents access to a shared, typically web-based application. Access control for these applications requires large amounts of fine-grained data, also from the remaining on-premise applications, of which often sensitive application data. With current SaaS applications the provider evaluates both provider and tenant policies. This forces the tenant to disclose its sensitive access control data and limits policy evaluation performance by having to fetch this data. To address these challenges, we propose to decompose the tenant policies and deploy them across tenant and provider in order to evaluate parts of the policies near the data they require as much as possible, while taking into account the tenant confidentiality constraints. We present a policy decomposition algorithm based on a general attribute-based policy model and describe a supporting middleware system. In the future, we plan to refine this work and evaluate the impact on performance using real-life policies from research projects.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.4.6 [**Security and Protection**]: Access controls

General Terms

Security, Performance

Keywords

federation, access control, Software-as-a-Service, performance, policy decomposition

1. INTRODUCTION

Access control is becoming increasingly complex, as shown by past and current industrial research projects [2, 3, 4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MW4NG '12 December 3-7, 2012, Montreal, Canada
Copyright 2012 ACM 978-1-4503-1607-1/12/12 ...$15.00.

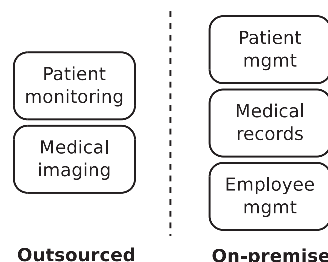

Figure 1: Large organizations such as hospitals employ both SaaS applications and on-premise applications, leading to a federated setup.

Firstly, access control infrastructures have become large-scale, distributed systems with strict performance and availability requirements as a result of the complexity of the applications they protect. Secondly, access control policies are growing larger and more complex, for example, having to reason about complex organizational structures and frequently changing business relationships. Access control policies thus require more data, more fine-grained data and often sensitive data. Thirdly, with the current ecosystem of business coalitions, applications involving multiple organizations, referred to as *federated applications*, have grown in importance. Access control should allow all parties to express their access control requirements, thereby adhering the relationships between them, resulting into *federated access control*.

Software-as-a-Service applications are becoming an important example of such federated applications. Software-as-a-Service or SaaS is a form of cloud computing in which the *tenant*, i.e., an organization representing multiple end-users, rents access to an application hosted by the *provider* for use through a thin client, e.g., a web browser. Originally, SaaS applications were mainly used by small and medium enterprises looking for a fully outsourced IT infrastructure. Typical examples are Google Apps[1] (an office suite) and Salesforce[2] (CRM). Recently however, large enterprises and non-profit organizations have started to adopt SaaS as well, for example health care organizations such as large hospitals [3]. For these organizations, SaaS applications outsource specific functionality and have to integrate with remaining on-premise applications such as patient management or medical record systems, as illustrated in Figure 1. This evolution

[1] http://www.google.com/enterprise/apps/business/
[2] http://www.salesforce.com/

leads to a federated setup between tenant and provider.

This federated setup poses key challenges for SaaS access control. For the tenant, SaaS is a form of outsourcing. While the SaaS application belongs to provider, the application data, although hosted by the provider, still belongs to the tenant. Therefore, SaaS access control should allow both the provider and the tenant to enforce access control upon the application. Within the scope of this paper, we focus on the tenant policies because of two reasons. Firstly, the tenant policies require data from both the tenant and the provider: its subjects are the tenant end-users and the subject data is hosted by the tenant, its objects are the data in the application and the object data is hosted by the provider. Because of the increasingly large amounts of data required by current policies, fetching this data is becoming the main bottleneck for tenant policy evaluation performance. Secondly, the tenant policies often employ data from the remaining on-premise applications which may contain sensitive data. Although the tenant may trust the provider with the data in the SaaS application, it does not necessarily trust the provider with this sensitive on-premise application data and wants to keep it confidential.

To address these challenges, we propose to decompose the tenant policies and deploy them across tenant and provider in order to evaluate parts of the policies near the data they require as much as possible, while taking into account the tenant confidentiality constraints. This process of decomposing policies and deploying them across the parties involved, we call *policy federation*.

This paper describes the results of our work in progress on policy federation. In our approach, we define a confidentiality-aware policy decomposition and deployment algorithm for optimal policy evaluation time using an abstract attribute-based policy model. We elaborate on the design of supporting technology such as a policy language to express confidentiality constraints and a middleware architecture to support policy federation. In the future, we plan to refine this work and evaluate the impact of policy federation on evaluation time using a larger set of policies from real-life case studies.

The rest of this paper is structured as follows. Section 2 further motivates this work from a case study in e-health. Section 3 describes the policy model we employ and Section 4 describes the policy federation algorithm and the notion of decomposition equivalence. Section 5 describes the required supporting technology in terms of meta-policies to express the confidentiality constraints and middleware to support the policy federation process. Section 6 gives an overview of related work and Section 7 finally concludes and elicits future work.

2. MOTIVATION

As stated in the introduction, large enterprises and non-profit organizations have started to adopt SaaS, for example in the domain of e-health. An example of such an application inspired on a number of research projects is a home patient monitoring system provided to hospitals as a service (see Figure 2). The system allows patients to be monitored continuously after leaving the hospital, for example by wearing a chest band or a wrist sensor. The measurements (the application data) are sent from the patients to the application back-end using a smart-phone as an intermediary device. The measurements are stored and processed by the provider, which notifies the patient's physician at the hospital of important evolutions. A patient's status can also be viewed by other physicians and nurses and by the patients themselves. The hospital is the tenant of the application, representing multiple end-users, i.e., patients and physicians. Next to the monitoring system, the hospital employs other SaaS applications, e.g., for medical imaging, and on-premise applications, e.g., for patient or employee management. As for all e-health applications, security is paramount for the patient monitoring system, amongst other because it handles personal data and is subject to stringent regulatory requirements (e.g., HIPAA [11] or the European DPD [7]). Of these security requirements, this paper focuses on the sub-domain of access control.

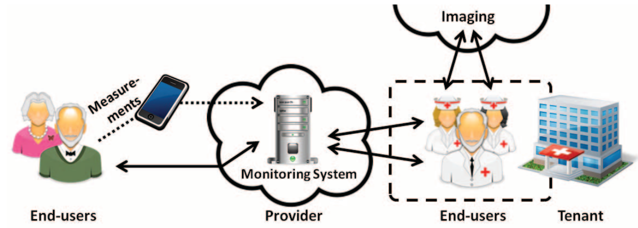

Figure 2: A home patient monitoring system offered as a SaaS application. On-premise applications are not shown for readability reasons.

Access control is an important part of application-level security that limits the *actions* (e.g., read, write) which a *subject* (e.g., an end-user) can take on an *object* in the system (e.g., a file) by enforcing access control rules generally expressed in *policies*. The most recent attribute-based access control (ABAC) [12] generalizes previous access control models and expresses these access control policies in terms of general key-value properties of the subject, the object and the environment, called *attributes*.

This paper focuses on the access control policies of the tenant, i.e., the hospital. Even though the patient monitoring system is hosted by the provider, the hospital is still accountable for the data and enforces its access control policies over the application. As a running example of a tenant access control policy, we use a policy P containing a single common rule from the case study, informally summarized as follows:

> P: *a user can only read patient data if he or she is a physician who is currently treating the patient who owns the data or if he or she is explicitly allowed to do so by the patient*

This rule requires the following attributes: (1) the id of the subject making the request, (2) the id of the requested action, (3) the id of the patient owning the data to which access is requested, (4) the roles of the user making the request, (5) the list of patients currently being treated by this user (only available in case the user is a physician) and (6) the list of subjects explicitly allowed to read the data. Of these attributes, (3) and (6) are located at the provider, (4) and (5) are located at the tenant and (1) and (2) are known to both. The tenant considers the list of patients currently being treated by a physician (attribute 5) as confidential.

In state-of-practice SaaS applications, all access control policies are evaluated by the provider. As a result, all required access control data has to be available to the provider.

This forces the hospital to share its sensitive access control data. Moreover, this limits the performance of policy evaluation since the complexity of current access control policies makes it impossible to determine the set of required attributes up-front and attributes have to be fetched dynamically while evaluating the policy.

In order address these issues, P can be decomposed and distributed over the provider and the tenant based on the location and confidentiality of the attributes, a process we call *policy federation*. For example, by having the tenant evaluate whether the user is a physician who is currently treating the patient, this data does not have to be shared with or fetched by the provider. However, in this case, the tenant does have to fetch the owner of the data from the provider. This cannot be avoided: this attribute has to be compared with the list of patients being treated by the physician, which is confidential for the tenant. More specifically, the number of attributes to be requested from the other party can be minimized while meeting the confidentiality constraints of the tenant using the following policy distribution (P_T and P_P being the parts of P evaluated by the tenant and the provider respectively):

- P_T: "a physician can only read patient data when currently treating the patient who owns the data",

- P_P: "a user can read patient data if explicitly invited by the patient".

It is up to the provider to combine the results of both policies so that the combination gives the same results as before, in this case by allowing the request if P_P or P_T allows it. This concept is called *policy equivalence* in this paper. As a result of this policy federation, the tenant has to trust the provider to correctly evaluate P_P and correctly combine P_T and P_P. However, this trust was also required with full provider-side access control. Moreover, the required trust of the provider in the tenant does not increase since the protected data belongs to the tenant and it does not have incentive to lie.

3. POLICY MODEL

In order to reason about policy decomposition, this section first defines a general attribute-based policy model, similar to the core features of current policy languages such as XACML [14]. To represent a policy, we employ the concept of a policy tree, similar to [8, 5].

The basic structure of the policy model is a three-level hierarchy consisting of policy sets, policies and rules. A rule specifies an effect ("permit" or "deny") and conditions for this to hold. Our model follows the ABAC approach and expresses conditions as constraints on the attributes of the subject, the object, the action and the environment. In case not all required attributes are present, the rule returns "not applicable". A policy can contain multiple rules and combines their effects using a rule-combining algorithm (e.g., deny overrides, permit overrides, first applicable), a policy set can contain multiple policies and combines their results using a similar policy-combining algorithm.

A rule itself 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 such as "physician". The top-level function of a rule

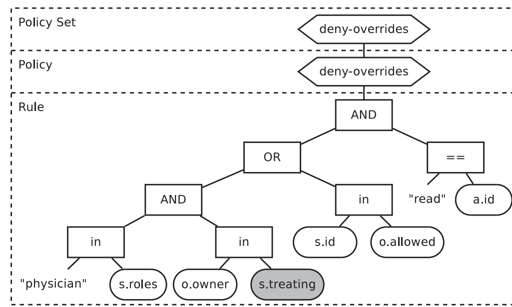

Figure 3: Policy tree representation of P as defined in Section 2. Rectangles represent functions, rounded rectangles represent attributes, plain text labels represent literal values, diamonds represent rule or policy combining algorithms, gray items are labeled confidential.

should result into a boolean (true resulting into the rule effect), such as equality predicates, numeric comparison functions, date, time and string comparison functions and set functions, but can internally contain more types of expressions, such as arithmetic operators. Most of these functions require two arguments, but the whole collection ranges from unary (e.g., string normalization) to n-ary (e.g., n-of).

In the model, two elements of a policy can be confidential for the tenant: (i) the attributes used in a policy and (ii) parts of the policies themselves. Since a function in itself cannot be confidential (the possible functions in a policy are publicly known), we limit confidentiality to rules, policies and policy sets.

Figure 3 shows the policy tree representation of P as defined in the previous section, wrapped in a policy with deny-overrides rule combining algorithm and policy-set with deny-overrides policy combining algorithm. This policy only contains one confidential attribute.

4. POLICY FEDERATION ALGORITHM

In this section, we define an algorithm to decompose and distribute the tenant access control policies across tenant and provider for optimal policy evaluation time while taking into account the tenant confidentiality constraints. In terms of policy trees, branches of the tenant policy will be split off and combined into a separate policy to be deployed at the provider.

Policy decomposition equivalence

An important property of the policy decomposition is that the combination of the sub-policies gives the same results as before. To make this more concrete, we here introduce the notion of an equivalent policy decomposition.

Definition: Policy decomposition equivalence The decomposition of a policy P into sub-policies P_T and P_P and a policy-combining algorithm PCA is equivalent to P iff for every request R and context Ctx, evaluating the decomposition gives the same result as evaluating P. The context Ctx is a collection of attribute values of the subject, the object, the action and the environment: $Ctx = (A_S, A_O, A_A, A_E)$. The request R is a subset of the context: $R \subset Ctx$.

This notion of policy decomposition equivalence translates

into policy distribution equivalence. In case the algorithm results into an equivalent policy decomposition, the distribution of these parts amongst tenant and provider is also equivalent since all attributes are available to both tenant and provider and it can be said that both policies share the same context.

Policy evaluation cost

In order to decide whether a part of a policy should best be evaluated by the tenant or the provider, the following cost functions are defined in terms of evaluation time:

$$C_P = N_{A,P} * C_{A,L} + N_{A,T} * C_{A,R} + C_E$$

$$C_T = N_{A,T} * C_{A,L} + N_{A,P} * C_{A,R} + C_{P,R} + C_E$$

The cost functions determine the cost of the provider (C_P) and the tenant (C_T) evaluating a certain policy based on the number of required provider attributes ($N_{A,P}$) and tenant attributes ($N_{A,T}$), the cost for fetching an attribute locally (C_L) or remotely (C_R) and the cost of evaluating the policy after fetching the attributes (C_E). C_T and C_P are not completely dual since a policy evaluation request from provider to tenant is required, adding an extra constant cost $C_{P,R}$. C_E is identical for C_P and C_T because both cost functions reason about the same policy and tenant and provider infrastructure is assumed to be similar. The location of every attribute determines the cost of fetching the attribute: $C_{A,L}$ will be much smaller than $C_{A,R}$, especially in federated applications such as SaaS applications because of the complex data flows and the geographical distance between the involved parties. For example, in case of a local database call, $C_{A,L}$ can be estimated around $0.1ms$ while a remote attribute fetch can be estimated around $10ms$, assuming a modest single-way latency of $5ms$ and negligible local processing time. Using similar reasoning, $C_{P,R}$ can also be estimated around $10ms$.

Algorithm

A high-level overview of the policy federation algorithm is given in Algorithm 1. The goal of the policy distribution is to minimize the total cost of evaluating the tenant policy. The algorithm requires six inputs: the policy to federate, the location of every attribute, the confidentiality label of every attribute, the confidentiality label of every rule, policy and policy set and the cost functions C_P and C_T. The confidentiality label of an attribute or policy element is a boolean that determines whether it can be shared with the provider or not. The algorithm provides two outputs: P_P, being the policy to be evaluated by the provider and P_T, being the policy to be evaluated by the tenant.

Informally, the algorithm searches the policy tree for subtrees which employ remote attributes and determines whether it is possible to externalize these. An important design decision is the granularity of policy distribution. While in theory every function in the policy tree can be externalized from the policy, we limit the granularity to sub-trees resulting into a boolean. This way, the result of the externalized sub-tree can be converted in a policy evaluation result and the existing policy combination algorithms can be used for combining the results of P_P and P_T. Thus, a sub-tree can only be externalized in case it results into a boolean, it does not handle confidential tenant attributes and it is not part of a confidential rule, policy or policy set. Then the algorithm determines whether it is beneficial to externalize the sub-tree by comparing C_P and C_T, splits off sub-trees best evaluated by the provider and combines these into P_P. P_T consists of the pruned original policy.

Algorithm 1 High-level policy federation algorithm

Inputs: P: a policy, having an effect and a policy tree annotated with confidentiality labels (true or false), A: the list of attributes, each having a value, location (local or remote) and confidentiality label (true or false), C_P: the function determining the cost of the provider evaluating a policy, C_T: the function determining the cost of the tenant evaluating a policy.

Outputs: P_P: the provider policy, P_T: the tenant policy.

```
// list of subtrees to externalize
ext = [ ]
// search for applicable subtrees
foreach Attribute a in P.tree:
  // parent function
  f = a.parent
  if f.returnType == Boolean:
    if { ∀ Attribute b in f.attributes | ! b.confidential }:
      if (! f.rule.confidential) and (! f.policy.confidential)
and (! f.policySet.confidential):
        if C_P(f) < C_T(f):
          ext.add(f)
          P.remove(f)
// combine all subtrees to externalize
P_P = combine(ext)
// result for tenant is pruned original policy
P_T = P
```

As an example, we can apply this algorithm to the example policy of Section 2 (represented in Figure 3). Of the four attribute checks, only checking the treating relationship contains a confidential attribute and should remain at the tenant. For the other three, the cost functions determine the optimal evaluation location: (1) checking the roles requires only tenant-side attributes and $C_P > C_T$, (2) checking the action requires the action id, which is available to both the tenant and the provider and $C_P = C_T$, (3) checking whether the user is explicitly allowed only requires provider-side attributes and $C_P < C_T$. Therefore, (1) is best evaluated by the tenant, (2) can arbitrarily be deployed and (3) is best evaluated by the provider. Combining the externalized subtrees results into P_P as shown in Figure 4, the pruned policy results into P_T as shown in Figure 5.The provider policy set contains a remote reference to P_T to request a policy evaluation and combines the results of P_T and P_P using permit-overrides. Notice that for a correct decomposition, checking the action has to be done in both P_T and P_P.

5. SECURITY MIDDLEWARE

In this section, we elaborate on the supporting security middleware for policy federation: (i) a way to express the confidentiality of the attributes and the policy elements and (ii) a middleware architecture to support the decomposition and deployment of the policies. This section describes an initial design of each of these elements.

5.1 Expressing confidentiality constraints

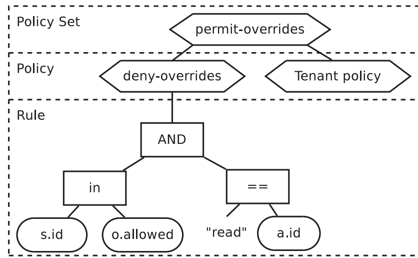

Figure 4: Result of federating P as defined in Section 2: P_P. The provider policy set contains a remote reference to the tenant policy.

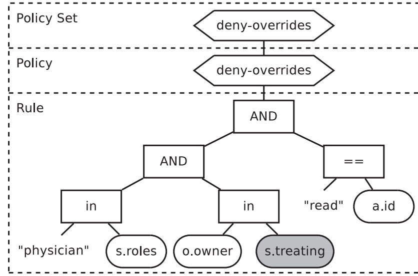

Figure 5: Result of federating P as defined in Section 2: P_T

In general, confidentiality constraints can be expressed by providing a separate meta-policy or by annotating the access control policies. Attributes are referenced multiple times throughout the tenant access control policies and using a separate meta-policy provides the advantage of central management. A similar approach has been used by the Shibboleth system for federated authentication [1], where attribute filters can be used to limit the release of an attribute to a certain party using a custom XML language. Policy elements on the other hand are best annotated in the access control policies themselves. A similar approach has been used by Gheorghe et al. [10]. They extend the XACML policy language [14] with the `AttrProps` element in order to express attribute properties such as cacheability.

In the simplest case, an attribute or policy can be labeled confidential or non-confidential, as assumed in the federation algorithm for now, and the required meta-policy or policy annotations are fairly simple. In a more extensive case, the policy can express more complex confidentiality rules, for example, limiting attribute release to some parties based on their identity or defining a certain combination of multiple attributes as confidential. Further research is required to analyze the needed expressiveness of future scenarios.

5.2 Middleware architecture

In order to support the decomposition and deployment of the tenant policies, a middleware system is required. In this section, we describe the middleware architecture using the XACML reference architecture for policy-based access control infrastructures [14].

In the reference architecture (see Figure 6), the policy decision point (PDP) makes the actual access control decision. The policy enforcement point (PEP, e.g., an API or a reference monitor) requests an access control decision from the PDP through the context handler. An access control request generally consists of information about the subject, the object, the action and the environment. The context handler gathers initially known attributes from one or more policy information points (PIPs, e.g., a database), which the PDP uses to evaluate the applicable policies loaded from the policy administration point (PAP). The PDP can request additional attributes from the context handler if needed. Eventually, the PDP returns its decision (permit or deny), which the PEP enforces.

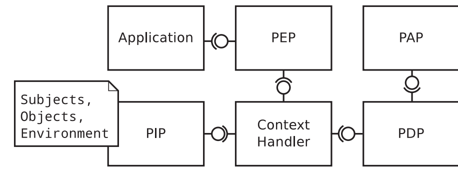

Figure 6: XACML reference architecture for access control

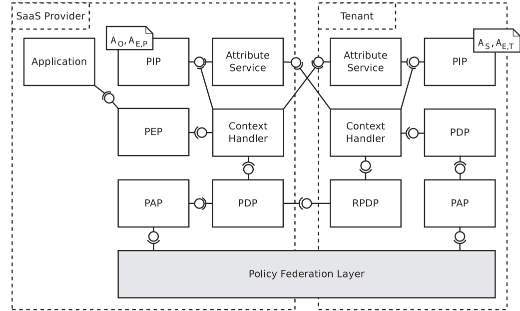

Figure 7: XACML reference architecture applied to SaaS. The Policy Federation Layer is the focus of this work.

The XACML reference architecture can be applied to SaaS as shown in Figure 7. Both the provider and the tenant can evaluate policies and have a PAP, a PDP, a context handler, one or more PIPs and an obligation service. The provider hosts the SaaS application and therefore also the PEP, no application components are located at the tenant side. The provider hosts the attributes concerning the objects in the application (A_O) and the provider part of the environment ($A_{E,P}$) and the tenant hosts the attributes concerning the subjects of the application (A_S) and the tenant part of the environment ($A_{E,T}$). All attributes are made available to the other party by means of an attribute service.

In this architecture, the tenant policies can be evaluated by the provider PDP or by the tenant PDP but in both cases, attributes have to be requested from the other party. In the former case, the provider context handler requests subject attributes from the tenant attribute service. In the latter case, the provider PDP requests a policy decision from the Remote Policy Decision Point (RPDP). Such a request is similar to a request from a PEP to a PDP. For the tenant, the RPDP acts as a PEP: it forwards the provider request to the tenant PDP through the context handler, returns the decision to the provider PDP and fulfills any tenant obligations using the tenant obligation service. The tenant context handler can request object attributes from the provider

attribute service. While being more complex, tenant-side policy evaluation does allow the tenant to evaluate its policies while not having to share any sensitive access control data.

The Policy Federation Layer shown in Figure 7 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.

6. RELATED WORK

The problem of confidentiality-aware access control for outsourced applications has been investigated by several other authors. For example, Asghar et al. [6] employ attribute and policy encryption, extending the work of di Vimercati et al., e.g., [9]. This approach is dual to policy federation, but is limited in policy expressivity. Several other authors have also focused on the problem of policy decomposition and distribution, such as Su et al. [15] and Lin et al. [13]. To the best of our knowledge, the work of Lin et al. is the only to distribute policies based on confidentiality and has been an important influence for this work. However, their work stops at describing the algorithms and does not provide confidentiality policies, supporting middleware or an evaluation of performance impact in practice.

7. CONCLUSIONS AND FUTURE WORK

In this paper we described access control for SaaS applications and focused on the problem of efficient policy evaluation while taking into account the tenant attribute confidentiality constraints. We proposed to decompose and distribute the tenant policies across provider and tenant in order to evaluate as much parts of the policy near the data they require while keeping the tenant access control data confidential, a concept called policy federation. The paper described our work in progress on the policy federation algorithm, the supporting policy model and the supporting middleware.

In the near future, we plan to refine and extend this work in several ways. Firstly, we plan to further extended the policy model presented in this paper, amongst others to take into account obligations. Obligations represent actions to be executed after the access control decision, e.g., updating an attribute such as the subject's access control history or usage quota and complicate the decomposition further. Secondly, we plan to proof the correctness of the policy distribution algorithm in terms of policy equivalence. Thirdly, the algorithm can be extended for improved performance, for example by optimizing concurrency or by taking into account attribute properties such as attribute cacheability [10]. Finally, we plan to further evaluate the performance impact of our approach on policy evaluation time using a prototype middleware and a set of policies from real-life case studies.

Acknowledgements This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, by the Belgian Science Policy, by the Research Fund KU Leuven, by the EU FP7 project NESSoS and by the Agency for Innovation by Science and Technology in Flanders (IWT).

8. REFERENCES

[1] Shibboleth. http://shibboleth.net/, August 2010.

[2] E-Health Information Platforms (E-HIP). `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=E-HIP`, May 2012.

[3] Healthcare professional's collaboration Space (Share4Health). `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=Share4Health`, May 2012.

[4] Permission, User Management and Availability for multi-tenant SaaS applications (PUMA). `http://distrinet.cs.kuleuven.be/research/projects/showProject.do?projectID=PUMA`, May 2012.

[5] C. Ardagna, S. De Capitani di Vimercati, S. Foresti, G. Neven, S. Paraboschi, F. Preiss, P. Samarati, and M. Verdicchio. Fine-grained disclosure of access policies. *Information and Communications Security*, pages 16–30, 2010.

[6] M. Asghar, M. Ion, G. Russello, and B. Crispo. Espoon: Enforcing encrypted security policies in outsourced environments. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 99–108. IEEE, 2011.

[7] E. Commision. 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.

[8] J. Crampton and M. Huth. An authorization framework resilient to policy evaluation failures. *Computer Security–ESORICS 2010*, pages 472–487, 2010.

[9] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati. Preserving confidentiality of security policies in data outsourcing. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, pages 75–84. ACM, 2008.

[10] G. Gheorghe, B. Crispo, R. Carbone, L. Desmet, and W. Joosen. Deploy, adjust and readjust: Supporting dynamic reconfiguration of policy enforcement. *Middleware 2011*, pages 350–369, 2011.

[11] U. Government. Health Insurance Portability and Accountability Act. 1996.

[12] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering dac, mac and rbac. *Data and Applications Security and Privacy XXVI*, pages 41–55, 2012.

[13] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. Policy decomposition for collaborative access control. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 103–112. ACM, 2008.

[14] T. Moses et al. eXtensible Access Control Markup Language (XACML) Version 2.0. *OASIS Standard*, 2005.

[15] L. Su, D. Chadwick, A. Basden, and J. Cunningham. Automated decomposition of access control policies. In *Policies for Distributed Systems and Networks, 2005. Sixth IEEE International Workshop on*, pages 3–13. IEEE, 2005.