An Implementation of a Privacy Enforcement Scheme based on the Java Security Framework using XACML Policies

Thomas Scheffler, Stefan Geiß, Bettina Schnor

Abstract In this paper we discuss implementation issues of a distributed privacy enforcement scheme to support Owner-Retained Access Control for digital data repositories. Our approach is based on the Java Security Framework. In order to achieve policy enforcement dependent on the accessed data object, we had to implement our own class loader that supports instance-level policy assignment. Access policies are described using XACML and stored together with the data as *sticky policies*. Enforcement of generic policies over sticky policy objects required the extension of XACML with XPath specific functions. Our use-case scenario is the user-controlled distribution of Electronic Health Records.

1 Introduction

The continuing advances in storage technologies allows the collection and storage of substantial data collections on mobile media such as smart cards. With the introduction and use of these mobile electronic data repositories for the storage and access of personal private data comes the requirement to securely enforce access policies for these repositories. Such an attempt requires the coordination between many parties, especially when the data on such media is used by many different principals and organisation. In this paper we propose a mechanism for the creation, distribution and enforcement of data-use-policies in distributed systems based on the Java Security Framework[11].

Many existing privacy protection techniques, such as P3P [7] and EPAL [3], are implemented by the custodian of the data. This protection model assumes that there are relatively few data release cases and the data itself is relatively immobile. With the use of mobile data repositories the data owner might want to implement

Thomas Scheffler, Stefan Geiß, Bettina Schnor
Department of Computer Science, University of Potsdam, 14482 Potsdam, Germany
e-mail: {scheffler,schnor}@cs.uni-potsdam.de

Please use the following format when citing this chapter:

Scheffler, T., Geiß, S. and Schnor, B., 2008, in IFIP International Federation for Information Processing, Volume 278; *Proceedings of the IFIP TC 11 23rd International Information Security Conference*; Sushil Jajodia, Pierangela Samarati, Stelvio Cimato; (Boston: Springer), pp. 157–171.

a need-to-know policy where data is only visible to the authorised data user and it is possible to maintain selective views on the repository. It would be beneficial to create the ability for Owner-Retained Access Control (ORAC), as described by McCollum [17], for the protection of mobile private data:

> "ORAC provides a stringent, label-based alternative to DAC[1] for user communities where the original owners of data need to retain tight control of the data as they propagate through copying, merging, or being read by a subject that may later write the data into other objects. . . . The user who creates a data object is considered its owner and has the right to create an ACL *(Access Control List)* on the object. "

Enforcing data-use policies in a distributed environment requires a distributed architecture, where each distributed component supports the access control scheme. A *Reference Monitor*, as defined by Anderson [1], is a trusted component that validates each and every request to system resources against those authorised for the subject.

A distributed policy enforcement is necessary to control data access. It must be secured that access to data is only possible via a trusted intermediary that reliably enforces the defined policy. Otherwise policies and/or data could be accessed, altered and deleted without trace and protection would be lost. Maintaining a trusted, distributed reference monitor infrastructure is one of the main challenges in the proposed architecture. Every participating site needs to trust and to install the necessary components. We base our solution on the existing Java Security Framework which might already be installed and trusted by most sites.

The Java programming language already provides Reference Monitor functionality for the safe execution of untrusted code. It was our aim to re-use these proven mechanisms for the enforcement of data-use policies. Data-use policies are specified in the eXtensible Access Control Markup Language (XACML) [22]. The private data of the data owner is translated into a suitable XML record format and stored together with the corresponding policy as a single XML data object. The Reference Monitor needs to intercept data access and enforces the XACML policy through a mapping onto Java permissions for the accessing application instance.

This paper focuses on the task of expressing, managing and enforcing authorisations for distributed data access. We assume that a suitable encryption and authentication scheme, such as XML Encryption [13], is used to protect data and policies from modifications and make data securely available to the authorised data user.

The rest of the paper is organised as follows: Section 2 explains a motivating use case for the application of ORAC policies, Section 3 introduces the Privacy Enforcement Architecture and explains the use of XACML policies. In Section 4 we describe implementational details for the policy enforcement using the Java Security Framework. Section 5 presents related work and the paper concludes with Section 6.

[1] Discretionary Access Control (DAC) - is characterised by the capability of subjects with access permission to pass that permission (perhaps indirectly) on to other subjects

2 Use Case

Electronic Health Records (EHR) are a good example for mobile electronic data repositories. The work described in this paper has been motivated by the ability to store and process personal health record data on mobile media, such as a patient smart-card. The German government has mandated the use of patient smart-cards for general health care [5]. The health cards have the ability to store personal health record data of the patient, so that it can be accessed and exchanged by different practitioners participating in the treatment process and act as a repository for future diagnosis. While it is in the interest of the patient to have this data available, the data sharing needs to be controlled, since it involves sensitive private data.

Historically, health records have been created, stored and accessed locally by the practitioner or hospital. Data access was restricted through the fact that patient records were only locally available. When data will be stored in a mobile electronic repository, a similar level of separation between the different data sources needs to be maintained.

In our use case, practitioners can add medical data from examinations and treatment processes to the electronic repository. For this purpose the repository is substructured into separate compartments that will be guarded by an appropriate access policy. The implementation of a suitable policy-set guarantees the same level of privacy between the different visits to practitioners that the patient can currently expect.

2.1 Data Model

Repository data is stored in a structured way and data access policies can be applied to these structures. Several standards exist for the structured data representation in EHR (cf. [6],[14]). Since the focus of this work is not the exact representation of medical data, but rather the creation, management and enforcement of access decisions, the EHR is represented as a simple XML document which is flexible enough to incorporate standards-based data representation as necessary.

We propose to group all treatment records generated by the same practitioner into a virtual *Examination Room* (cf. Figure 1). A $1 : m$ relationship between practitioner and *Examination Room* is assumed. All treatment records generated by the practitioner are stored under this particular node and form a single zone of trust similar to the existing patient – practitioner relationship.

2.2 Use Case Policy Example

Hierarchical grouping is a widely used concept in the field of access control. It allows to minimise access rule management - rules can be defined and enforced at

```
<?xml version="1.0" encoding="UTF-8"?>
<healthRecord>
    <demographicData>
        <patient_id>CN=Homer J. Simpson, ... </patient_id>
        <dayOfBirth>19670904</dayOfBirth>
    </demographicData>
    <practitioners>
        <practitioner id="CN=Julius Hibbert, ...">
            <examinationRoom>
                <visit date="2007-11-28 15:06:37">
                    <description>X-Ray taken...</description>
                    <attachments>
                        <attachment filename="homer_brain.jpg"
                          mimetype="image/jpg" >...</attachment>
                    </attachments>
                </visit>
            </examinationRoom>
        </practitioner>
    </practitioners>
</healthRecord>
```

Fig. 1 Electronic Health Record Example

the group level, thus minimising the number of rules in the policy. The practitioner, as data author, has specific rights for his or her sub-tree in the patient health record. These can be specified as a generic rule affecting all groups of a certain type and be applied consistently for every instantiation of this type:

- Practitioner can create new examination entries in his/her personal examination room
- Practitioner can read examination entries from his/her personal examination room

A distinctive feature of the use case is the fact that data ownership and authorship are separated. The data owner determines the access policy for data access by other practitioners, but is constrained in the policy editing in order to avoid errors (e.g. revoke its own access rights) and inconsistencies (e.g. can not create a policy that allows him or her to act as a data author):

- Patients can grant access rights for practitioners to read examination entries of other practitioners
- Patients can grant the right to export entries from the health card into medical information system
- Patients have no right to create/modify entries in the examination rooms

3 Privacy Enforcement Architecture

The creation, distribution and enforcement of ORAC policies in a distributed environment requires the presence of an enforcement architecture that supports distributed policy creation and evaluation. Figure 2 shows a simplified version of the generic architecture described by the XACML standard [19]. The *Policy Administration Point* (PAP) is the entity that creates an access policy and makes this policy available to the *Policy Decision Point* (PDP). The data user tries to access a resource via a *Policy Enforcement Point* (PEP) and thus triggers a *Decision Request* to the PDP which will issue an appropriate *Decision Response* based on the available policy. The PEP then grants or denies access in accordance with this policy decision.

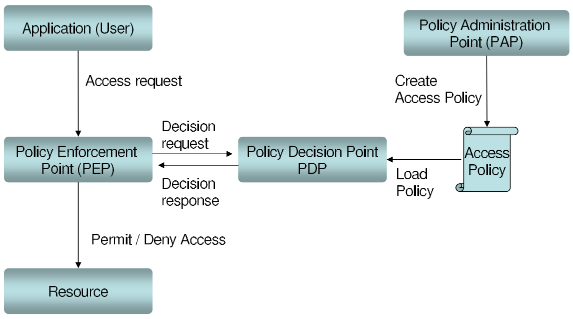

Fig. 2 Simplified XACML Access Control Architecture

Policy description languages, such as XACML are well suited to express usage policies for Electronic Health Records [2]. Policy languages have the ability to express policies for logically grouped objects and subjects and can express dependencies from environmental conditions (e.g. time of access). These properties allow the creation of concise policies that can be specified at high level of abstraction close to the intention of the policy creator.

XACML policies must be evaluated at the time of access in order to determine the access decision. This access decision is generated by the *Policy Decision Point* which implements a deterministic policy evaluation algorithm. In our architecture we use and extend a Java-based XACML implementation provided by Sun [21].

3.1 Sticky Policy Paradigm

Data access policies need to be referenced reliably throughout the distributed architecture. Policy storage and distribution becomes an important design choice for the implementation of the architecture. One possibility would be to store policies in a central repository. This requires the accessibility and availability of the policy store for every potential data user at any given time and would be suitable if the data is also centrally stored.

Our use case assumes that data is stored on a mobile media and thus needs to reference the policy independently. A policy distribution method, well suited for handling access to distributed data, is the *Sticky Policy* paradigm (cf. [15]). The data access policy is stored and distributed together with the data that it is protecting.

Together they form a sticky data object that allows the direct referencing of the policy as data needs to be accessed.

Figure 3 shows the enforcement architecture for the *Sticky Policy* model. Access to data and policy is mediated through the *Policy Administration Point*. The protected resource and its access policy are created and stored together.

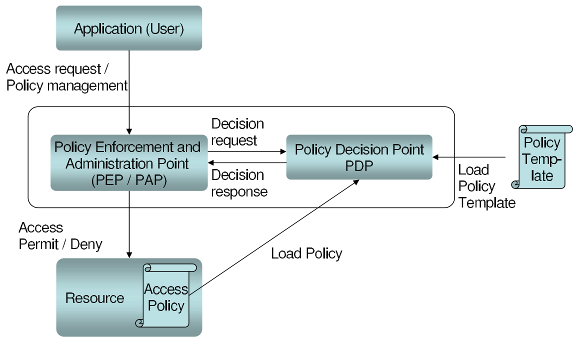

Fig. 3 Access Control Architecture using 'Sticky Policies'

The XACML standard separates the description of authorisation policies from the actual resources. However, since policy and data are XML-based resources, policies can be included directly in the EHR document and referenced via XPath [9]. It then becomes the responsibility of the PEP to select the requested resource node from the XML document and query the PDP for a policy decision regarding the authenticated subject and the requested action for this resource.

3.2 *Use case policy examples*

Policy management will be controlled by a Policy Template (as shown in Figure 3), that also guides the policy creation process. The policy template has the function to apply a default policy for newly created EHR entries that already enforces a basic privacy protection level. Secondly, the default rules are needed to limit the data owner and data author in their administrative power over data and policies (e.g. the data owner should not be able to refuse data access to the data author).

The XACML policy-base contains two types of rules:

1. **Generic access rules** which define default behaviour for policies over the set of resources. These rules are static and immutable and based on the Policy Template.
2. **Specific access rules** define resource specific policies and are managed by the data owner to create specific access decisions (e.g. granting extended access to treatment records for an external practitioner)

Example 1. **Generic Rule:** A data owner has read access to his or her own resources

```
<Rule RuleId="dataOwnerRule0" Effect="Permit">
    <Target>
      <Subjects><AnySubject /></Subjects>
      <Resources><AnyResource /></Resources>
      <Actions>
        <ActionMatch MatchId="string-equal">
           <AttributeValue DataType="string">read</AttributeValue>
           <ActionAttributeDesignator AttributeId="action-id"
               DataType="http://www.w3.org/2001/XMLSchema#string" />
        </ActionMatch>
      </Actions>
    </Target>
    <Condition FunctionId="function:xpath-node-element-x500-compare">
       <Apply FunctionId="x500Name-one-and-only">
         <SubjectAttributeDesignator DataType="x500Name"
         AttributeId="subject-id" />
       </Apply>
       <Apply FunctionId="string-concatenate">
           <Apply FunctionId="string-one-and-only">
             <ResourceAttributeDesignator AttributeId="resource-id"
              DataType="http://www.w3.org/2001/XMLSchema#string" />
           </Apply>
           <AttributeValue DataType="string">
              /parent::attachments/parent::visit/parent::examinationRoom/
                           parent::practitioner/@id</AttributeValue>
       </Apply>
    </Condition>
</Rule>
```

Based on the requirements of our use case, new resource-trees can be added to the document anytime. In order to support dynamic comparison between the data user and the data owner of the currently selected resource sub-tree, we need to compare the current data user with the data owner of the sub-tree of the requested resource.

The XACML standard provides an XPath expression-based function for the selection of XML attributes. XACML uses the `<AttributeSelector>` element to identify a particular attribute value based on its location in the request context. The `RequestContextPath` attribute of the `<AttributeSelector>` element takes a static XPath expression and evaluates to a bag of values, given by the `DataType` attribute. The drawback of this function lies in the fact, that the attribute value for the `RequestContextPath` attribute handles only fixed XPath expressions that must be fully known at policy creation time.

We define a new XPath-based function that enables referencing and comparing of XML nodes relative to the currently selected resource:

The function `<function:xpath-node-element-x500-compare>` takes two arguments. The first argument is of data type:
`urn:oasis:names:tc:xacml:1.0:data-type:x500Name`
and the second argument is of data type:
`http://www.w3.org/2001/XMLSchema#string`
which is interpreted as an XPath expression and evaluates to a
`urn:oasis:names:tc:xacml:1.0:data-type:x500Name`

This function returns an `http://www.w3.org/2001/XMLSchema#boolean` and allows the dynamic creation of an XPath expression for the second argument, using the standard XACML string manipulation functions, such as concatenation. Both arguments are treated as `X500Name` values. The function compares the arguments and if they match, the function evaluates to **true**.

Example 2. **Specific Rule:** A practitioner is granted access to a treatment record for a limited time period

```
 <Rule Effect="Permit">
    <Target>
     <Subject>
         <SubjectMatch MatchId="x500Name-match">
             <AttributeValue DataType="x500Name">CN=Julius Hibbert,
               ... </AttributeValue>
             <SubjectAttributeDesignator AttributeId="subject-id"
               DataType="x500Name"/>
         </SubjectMatch>
     </Subject>
     <Resources>
       <ResourceMatch MatchId="xpath-node-equal">/healthRecord/
                 practitioners/practitioner/examinationRoom/visit/
         attachments/attachment/@filename='homer_brain.jpg'
       </ResourceMatch>
     </Resources>
     <Action>view</Action>
    </Target>
    <Condition FunctionId="date-less-than-or-equal">
       <Apply FunctionId="date-one-and-only">
          <EnvironmentAttributeDesignator DataType="date"
               AttributeId="current-date" />
       </Apply>
       <AttributeValue DataType="date">2009-03-22</AttributeValue>
     </Condition>
</Rule>
```

Generic rules capture the default behaviour of the system and can not be changed by the data owner or the data user. Specific rules can be added and deleted by the data owners depending on the different trust relationships and data exchange needs. These two rule-sets can be maintained separately using the existing XACML policy combining mechanism.

4 Reference Monitor Implementation

Implementation of the *Privacy Enforcement Architecture* requires the presence of a trusted system component at every data access location. An ideal *Policy Enforcement Point* would be based on a trusted virtual machine implementation that has the ability to enforce data use policies. We choose to base our implementation on the Java Security Framework and use the permission concept of the Java security manager for the enforcement of data use policies. Client applications will be started under the control of the Java security manager that controls resource access based on an appropriate data access policy for an application instance.

4.1 Java Security Architecture

The Java programming language provides a *Security Framework* that is aimed to protect the local system user from threats arising from untrusted Java code that is executed on the local system (such as Java Applets). Local programs typically have the full set of rights to access any resource on the system. Untrusted programs run under the supervision of the Java *SecurityManager* within a sandbox environment and are restricted in their access to system resources. For each Java Virtual Machine there exists exactly one instance of the *SecurityManager*. With the introduction of the Java 2 Security Architecture [11] the rigid sandbox model became much more refined and allows now the definition of application-specific security policies through the definition of permissions also for local programs.

Policies for resource access by Java applications started under the control of the SecurityManager are established through the Java `Policy` class. The default policy implementation uses text based configuration files to determine the actual set of permissions. The policy file(s) specify what permissions are granted for code from a specified *CodeSource* and support fine grained permissions.

All permissions granted to a class are encapsulated within a `ProtectionDomain` which is determined and irrevocably bound to the class at class loading time by the Java class loader. Permissions are granted depending on the origin of the code `CodeSource` and the user of the application `Principal` and bundled as a `PermissionCollection`. A `ProtectionDomain` for a class is constructed from the `CodeSource`, the `PermissionCollection`, a `ClassLoader` reference and the `Principal` who executes the code.

4.2 Assigning a XACML-Policy

In order to be enforceable through the Java Security Framework, XACML policies need to be translated into Java permissions. Only such policies that can be mapped to a corresponding Java permission can be directly enforced through the Reference Monitor without cooperation of the application. The set of permissions for the `ProtectionDomain` will be derived from the XACML policy description of the data object that is currently accessed.

The Reference Monitor is responsible for the translation of the XACML policy into a `PermissionCollection` and the launching of a restricted application component under the protection of the Java SecurityManager. Two alternatives exist to base permissions granted to code on XACML rather than the standard Java policy:

1. Write a new implementation of the `Policy` class, that derives permissions from XACML policies, rather than Java policy files
2. Implement a class loader that is able to derive and assign a `ProtectionDomain` from XACML policies

Writing a new `Policy` class would allow us to derive permissions from XACML. However, as permissions apply to the code source, we could not distinguish between different policies for application instances derived from the same code source. We therefore implement a custom class loader, because this gives us the possibility to assign different policies for application instances as we will see later. We started our implementation with the extension of the `SecureClassLoader` class. Our class loader overrides the `getPermissions()` method in order to allow the creation of the `ProtectionDomain` from the XACML Policy rather than the standard Java policy files. A XACML policy is typically much broader in scope than a Java permission that applies for a specific data object and the currently authenticated user. However, all granted actions need to be known at class loading time to be included in the `ProtectionDomain`. To determine the full set of permissions for a specific data object we execute several XACML requests, each against the corresponding data object and data user, but with different actions. We use the mapping in Table 1 to translate XACML policy responses into Java permissions.

The Reference Monitor generates a dedicated view on the data object for the called application, that is destroyed once the application quits. The actions *append* and *delete* therefore apply only to this view and require the cooperation of the Reference Monitor to persistently change the XACML data object. The Java `SecurityManager` enforces the permissions of the `ProtectionDomain` and intercepts actions that are not authorised.

Table 1 Mapping of XACML policy actions against Java Permissions

Java Permission	**XACML policy actions**					
	read	***copy***	***save***	***print***	***append***	***delete***
AWT:accessClipboard		x				
Runtime:queuePrintJob				x		
FilePermission:read	x					
FilePermission:write			x		x	
FilePermission:delete						x

4.3 Assigning Instance-Level Permissions

The current Java Security Architecture is targeted towards class based policing. Trust settings are applied by the code source and not by the running instance based on this code. For the realisation of Owner-Retained Access Control the reference monitor needs to enforce different policies depending on the current execution environment and data source.

An instance-based policing is necessary to distinguish between different instances of an application that simultaneously load data-sets with different access policies. Data users will be restricted in their actions based on the data that they are accessing. For each data object that is been accessed it becomes necessary to reference the corresponding policy before access is granted. When the data user is accessing different data objects during one session it becomes necessary to enforce more than one access policy.

Our class loader assigns a dedicated `ProtectionDomain` and loads the application class when data access is granted. The assignment of a new `ProtectionDomain` to a class is only possible at class loading time and can not be revoked or changed. Dynamic policy enforcement therefore requires the loading of a new class, including the construction of a new `ProtectionDomain`, for every data object that is accessed.

The default implementation of the `loadClass()` method in the `ClassLoader`, as described in [10], loads a class in the following order:

1. Call `findLoadedClass()` to check if the class is already loaded. If this is the case, return that object. Otherwise,
2. call the `loadClass()` method of the parent class loader, in order to delegate the task of class loading to the parent (this is done to ensure that system classes are only loaded by system class loaders).
3. If none of the parent class loaders in the delegation hierarchy is able to load the class, the `findClass()` method of this class loader is called in order to find and load the class.

With the behaviour of the default *loadClass* method the existing class would be found and re-used. In order to load a class with a new `ProtectionDomain` we load the class with a new instance of our class loader. The class loader uses a modified `loadClass()` method and no longer calls `findLoadedClass()` to check if the parent class loader already knows this class. Instead `findClass()` is called directly to load the class with a new `ProtectionDomain`.

Namespace separation in Java is enforced through the use of different class loaders. Two classes are considered distinct if they are loaded by different class loaders.

4.4 Use Case Implementation

To validate the protection concept outlined above we developed a prototypical implementation of a medical information system. Figure 4 visualises the interworking of the framework components. A resource browser component let the data user authenticate, select interesting events in the Electronic Health Record and start the Health Record Viewer (HRV) upon the selected entries. The HRV visualises the medical data of the health record such as images and diagnostic text and will be started under the control of the Java security manager. An appropriate permission setting will be derived from the XACML-policy part of the EHR. Since the HRV is under the complete control of the security manager the implementation of this component does not need to be fully trusted. The HRV can implement a superset of functions (such as print, save, etc.) whose execution will be restricted at runtime according to the specified data-use policy.

Multiple instances of HRV can be started simultaneously for different data objects and allow the comparison of different diagnoses or illnesses. Each instance of the HRV will carry its individual set of permissions based on the data that is being accessed.

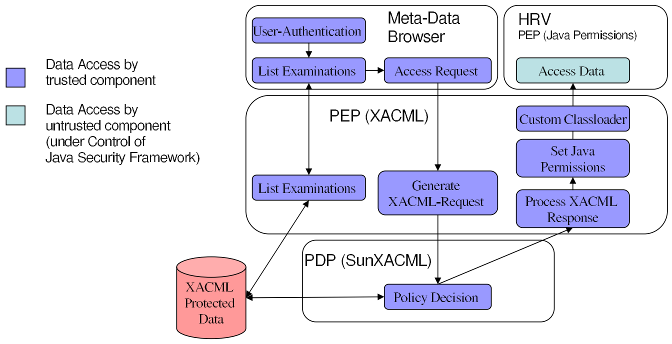

Fig. 4 Implemented Privacy Enforcement Framework

The XACML policy will be evaluated at the moment the application is loaded via the Reference Monitor. The Reference Monitor iterates through the set of actions contained in the policy-base for a given subject/resource pair to gather all the related permissions. An appropriate set of Java permissions is generated from the underlying XACML policy. The actual policy enforcement is offloaded to the Java Security Framework. No XACML requests have to be evaluated at the time of resource access of the HRV.

Policy enforcement is limited by the support of native permissions in Java – which are primarily focused on the Java threat model. Policies that can not be directly enforced through the native Java permission mechanism include the ability to control the file-append function and time based policies that enforce access time restrictions.

Time restricted policies can be handled at application start-up time by the Reference Monitor, but require reliable access to a trusted time-base.

The Java permission model can be extended through application specific extension of the policy class, however in this case the application needs to be trusted to correctly implement the necessary access checks. Implementation of new permissions would require the extension of our trust model, that currently only includes the Reference Monitor.

5 Related Work

Different policy schemes have been proposed to aid the data owner in the task of protecting his or her data.

Author-X [4] is a Java based data protection solution that allows the definition and enforcement of access restrictions on (parts of) XML-documents. It is a server-based solution where the document access is mediated by the access component, based on the collocated authorisation store. Access can be granted to parts of the complete document. No further protection mechanism exists once data access has been granted. While we realise a similar view on the document protections mechanism, our proposed protection scheme can enforce policies even after the data is released to the data user.

Damiani et al. [8] developed an access control system for XML documents that is able to describe fine grained access restrictions for the structure and content of XML documents. Their system generates a dedicated user view according to the permissions granted in a server-side authorisation file: the XML Access Sheet (XAS). The system does not implement any control over data that has been released by the server to the client. Consequently any information that is granted to be read by a user could be locally stored, copied and processed by the client. The generated view restricts data processing for single action classes only, e.g. the ’read’ action. No support is given for orthogonal action sets, e.g. restricting a document to be read, but not to be printed.

Mont et al. [18] propose a privacy architecture that uses sticky policies and obfuscated data that can only be accessed if the requester can attest compliance with the requested privacy policy for this data. Data access is mediated via a Trusted Third Party that can reliably enforce time-restricted access. Our work aims to provide similar protection but does not depend on functions provided by another party. It uses the functions of a trusted reference monitor instead.

Sevinc and Basin [20] describe a formal access control model for documents based on the sticky policy paradigm. In their work they focus on document related actions such as read, print, change and delegate. Their model supports multiple owners and sub-policies for document parts and takes document editing into account, where merging and splitting of document content also influences the attached policies. We believe that our work fits within their problem definition but we focus mainly on implementational issues.

Lehmann and Thiemann [16] have developed a field access analyser that is able to analyse existing Java programs in order to determine the points in the program code where object methods are accessed. Static policy checking code is inserted to enforce access controls in accordance with the access-control policy for the program. In our work we choose to clearly separate the policy enforcement from program execution. No access to the application source code is necessary for the policy enforcement and policies can be expressed, evaluated and enforced independently from the application.

Gupta and Bhide [12] describe an XACML based authorisation scheme for Java that extends the Java Authentication and Authorisation Service. The work describes a generic implementation that extends the Java policy class with the ability to interpret XACML policies. While this work allows the Java Security Framework to enforce permissions for different users of an application, it might not be possible to enforce ORAC policies where different permissions need to be enforced depending on the data object that is currently accessed.

6 Conclusion

We investigated whether standard techniques like the Java Security Framework and XACML are sufficient for the implementation of privacy enforcement.

Our implementation allows the start of arbitrary, untrusted Java programs under the control of the Java Security Framework. The relevant access permissions of the application are derived at runtime from the policy of the data object that is being accessed. The developed architecture provides fine grained policy support for the enforcement of document policies at application level, independent from specific OS security mechanisms. We implemented a new class loader that supports instance level policy assignment. No new access control mechanism had to be added, as we use the existing implementation of the Java Security Manager.

The XACML policy language was used for the definition of data-use policies by the original data owner. The private data of the data owner is translated into a suitable XML record format and stored together with the corresponding XACML policy as a single XML data object. Data access policies are defined and bound to the data at creation time and revised later as access decisions need to be granted or revoked. Policy management is aided through the separation of generic default-policies from user-editable specific policies. The private data is referenced from the XACML policy via XPath.

References

1. Anderson, J.P.: Computer security technology planning study. Technical Report ESD-TR-73-51 (October 1972)
2. Apitzsch, F., Liske, S., Scheffler, T., Schnor, B.: Specifying Security Policies for Electronic Health Records. In: Proceedings of the International Conference on Health Informatics (HEALTHINF 2008), vol. 2, pp. 82 – 90. Funchal/Madeira, Portugal (January 2008)
3. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise Privacy Authorization Language (EPAL 1.2) (November 2003). URL http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/
4. Bertino, E., Braun, M., Castano, S., Ferrari, E., Mesiti, M.: Author-X: A Java-Based System for XML Data Protection. In: Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions, pp. 15–26. Kluwer, B.V. (2001)
5. Bundesgesundheitsministerium: Gesetz zur Modernisierung der gesetzlichen Krankenversicherung, SGB V, §291a. In: Bundesgesetzblatt, vol. 55 (2003)

6. CEN/TS-15211: Health informatics - Mapping of hierarchical message descriptions to XML. European Committee for Standardisation (2006). URL http://www.cen.eu
7. Cranor, L., Langheinrich, M., Marchiori, M., Presler-Marshall, M., Reagle, J.: The Platform for Privacy Preferences 1.0 (P3P1.0) Specification (April 2002). URL http://www.w3.org/TR/2002/REC-P3P-20020416/
8. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for XML documents. ACM Transactions on Information and System Security **5**(2), 169–202 (2002)
9. DeRose, J.C.S.: XML Path Language (XPath). W3C Recommendation (1999). URL http://www.w3.org/TR/1999/REC-xpath-19991116
10. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security - Second Edition. Addison-Wesley, Boston (2003)
11. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit. In: USENIX Symposium on Internet Technologies and Systems. Monterey, California (1997)
12. Gupta, R., Bhide, M.: A Generic XACML Based Declarative Authorization Scheme for Java, *Lecture Notes in Computer Science: Computer Security - ESORICS 2005*, vol. Volume 3679/2005. Springer Berlin / Heidelberg (2005)
13. Imamura, T., Dillaway, B., Simon, E.: XML Encryption Syntax and Processing. W3C Recommendation (2002). URL http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/
14. ISO/HL7-21731: Health informatics - HL7 version Reference information model Release 1) (2006)
15. Karjoth, G., Schunter, M., Waidner, M.: Platform For Enterprise Privacy Practices: Privacy-enabled Management Of Customer Data. In: 2nd Workshop on Privacy Enhancing Technologies (PET2002), vol. Lecture Notes in Computer Science 2482, pp. 69–84. Springer Verlag (2003)
16. Lehmann, K., Thiemann, P.: Field access analysis for enforcing access control policies. In: Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS 2006), *Lecture Notes in Computer Science*, vol. 3995, pp. 337–351. Springer-Verlag, Berlin, Heidelberg (2006)
17. McCollum, C.J., Messing, J.R., Notargiacomo, L.: Beyond the pale of MAC and DAC-defining new forms of access control. In: IEEE Computer Society Symposium on Research in Security and Privacy, pp. 190–200 (1990)
18. Mont, M.C., Pearson, S., Bramhall, P.: Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In: Proceedings of the 14th International Workshop on Database and Expert Systems Applications, p. 377. IEEE Computer Society (2003)
19. Moses, T.: eXtensible Access Control Markup Language (XACML) Version 2.0. XACML Core Standard (2005). URL http://www.oasis-open.org/committees/xacml
20. Sevinç, P.E., Basin, D.: Controlling Access to Documents: A Formal Access Control Model. Technical Report No. 517, Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland, (May 2006)
21. SUN: Sun's XACML implementation (2005). URL http://sunxacml.sourceforge.net/
22. XACML-2.0: eXtensible Access Control Markup Language (XACML). OASIS-Standard (2005). URL http://www.oasis-open.org/committees/xacml