An Access Control System for Web Service Compositions

Mudhakar Srivatsa[†], Arun Iyengar[‡], Thomas Mikalsen[‡], Isabelle Rouvellou[‡] and Jian Yin[‡]

College of Computing, Georgia Institute of Technology, Atlanta, GA − 30332[†]

IBM T. J. Watson Research Center, Yorktown Heights, NY − 10598[‡]

mudhakar@cc.gatech.edu, {aruni, tommi, rouvellou, jianyin}@us.ibm.com

Abstract. Service composition has emerged as a fundamental technique for developing Web applications. Multiple services, often from different organizations or trust domains, may be dynamically composed to satisfy a user's request. Access control in the presence of service compositions is a challenging security problem. In this paper, we present an access control model and techniques for specifying and enforcing access control rules on Web service compositions. A key advantage of our approach is that past histories of service invocations can be used to make access control decisions. Our approach allows role hierarchies and separation of duty constraints. Access controls rules may be parameterized by one or more arguments. We have implemented our access control model via a declarative policy specification language which uses pure-past linear temporal logic (PPLTL). We describe an implementation of our approach using a supply chain management (SCM) application. Our experiments show that our approach can enforce expressive and flexible access control policies while incurring reasonable performance overhead on the application.

1 Introduction

Service-oriented computing (SOC) has emerged as a powerful paradigm for building complex Web applications from simpler components known as services [6]. In SOC, independently developed services interoperate with each other via well-defined interfaces. The services may be heterogeneous and possibly implemented in different languages. This approach provides considerable flexibility in building applications as different component services may be used at different times for implementing parts of the application. It can also provide isolation and fault tolerance for individual services. SOC has been widely used for integrating both business and scientific applications that operate in distributed heterogeneous environments.

There are a number of challenges that arise in implementing SOC. One of them is access control. An application may need to provide a wide variety of different privileges for allowing clients to access services and data. The level of access a client is granted would typically depend on the identity of the client. Further complications are introduced by service compositions. Service composition is one of the fundamental aspects of SOC [22, 7]. When a client attempts to access a Web service composed of one or more services, it is often desirable to consider the history of previous Web service invocations in order to decide whether to grant access; simply considering the identity of the client may not be sufficient.

This paper presents an access control system for Web services which is particularly well-suited to Web service composition. A key advantage of our approach over past ones is that service composition is treated as a first class entity. For example, consider a supply chain management application composed of three entities: customers, a retailer system and a manufacturer system. The application may include a `retail manager` role and the following services: a `retail service`, a `warehouse service` and a `database service`. The retailer system needs to distinguish a `retail manager` accessing the `database service` through the `retail service` and a `retail manager` accessing the `database service` through the `warehouse service`. While the former may have `retail manager`-like privileges on the database service, the latter may have more restricted privileges (detailed examples follow in Section 2). Our system allows an application writer to easily specify access control rules which take past service invocations into account.

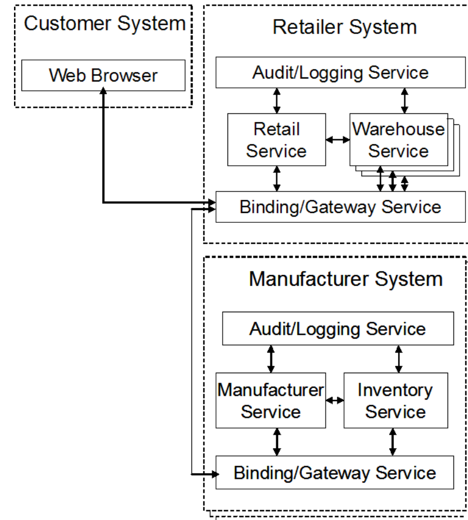

Figure 1: SCM Application

We present an access control policy specification language based on pure-past linear temporal logic (PPLTL). It includes standard Boolean predicates as well as support for temporal predicates which formalize concepts such as *sometime in the past, subsequently, last time, since, etc.* Access control rules may be dependent on one or more parameters associated with a service invocation. The application writer may also define *role hierarchies* in which relationships between the access privileges of different types of clients may be specified.

2 Access Control Requirements

In this section we discuss several access control requirements for Web service compositions. We provide concrete scenarios with illustrations from a supply chain management (SCM) application which has been defined by the Web Services Interoperability Organization WS-I [24] (see Figure 1). The SCM application consists of at least three systems: the customer, the retailer and the manufacturer(s). Each system refers to an autonomous organization or a trust domain. The retailer system is composed of the following services: `retailer service`, and one or more `warehouse services`. The `retailer service` accepts orders, processes them and schedules deliveries for the ordered items from one or more of the warehouses using the `warehouse service`. The manufacturer system is composed of the following services: `inventory service` and a `manufacturer service`. The `inventory service` is responsible for maintaining item inventory levels, delivering items to retail warehouses, and throttling the production rate of items using the `manufacturer service`. In addition, all systems include a standard set of services including: a binding or `gateway service`, a `database service` and an audit or `logging service`. All inter-organizational communications are routed using the `gateway service`, and all transactions are logged by the `logging service`.

The retailer system may have several principals that can play different organizational roles such as `employee`, `retail manager`, `warehouse manager` and `chief manager`. Concretely, a principal P refers to a person, user or client, and a role R refers to a job function or title which defines an authority level. These roles are typically organized hierarchically, say, `employee` $<_R$ {`retail manager`, `warehouse manager`} $<_R$ `chief manager`. Note that $R_1 \geq_R R_2$ denotes the fact that the role R_1 dominates the role R_2, that is, the privileges of R_1 are a superset of those of R_2.

A key feature in supporting service compositions is to retain a past history of service invocations in order to make future decisions about access control. For example, consider a situation in which a client is attempting to access a service s_1 through another service s_2 which was previously invoked by another service s_3. The fact that the client accessed s_3 means that it should not access s_1 because it could use information obtained from s_3 along with the service invocation of s_1 to either cause harm or obtain information which it should not have. Therefore, the access control system should deny access to s_1. In order to make this decision, knowledge about previous service invocations is required.

In another scenario, suppose that the client normally would not have enough privileges to invoke s_1 just based on its identity. However, the fact that the invocation took place transitively through s_3 changes things because s_3 performed some additional security checks before allowing the client to proceed to the point at which it could invoke s_1. Therefore, the client can, in fact, safely invoke s_1. Hence, the primary requirement is to maintain minimal and yet sufficient information about the call invocation history such that one can determine the privilege information for a request after multiple service invocations.

In the supply chain management scenario, we may need to be able to distinguish between a `retail manager` accessing the `database service` through the `retail service` from a `retail manager` accessing the `database service` through the `warehouse service`. Other examples include: An `employee` operating through the `retailer service` should be able to read records in the `order` database but should not be able write or update these records; a `chief manager` should be able to read, write, or update records in the `order` database even if the operation is not mediated by the `retail service`. In our system, we do this using *temporal predicates* on a composite principal which allow us to formalize temporal concepts such as *sometime in the past*, *subsequently*, *last time*, *since*, etc.

Another key requirement is *role translation* which we illustrate using the SCM application. The `warehouse service` permits the manufacturer's `inventory service` to inspect the inventory level of its items in the warehouse. If the inventory of its items were to fall below a threshold, the manufacturer may automatically increase its production of that item (if needed) and ship those items to the warehouse[1]. Such inter-organizational service invocations require an entity in the manufacturer's trust domain to be recognized by the retailer system. In this example, the retailer R needs to recognize a role (say `inventory manager`) from the manufacturer M and translate it to an appropriate role in its trust domain (say, an `employee`).

While role translation allows an `inventory manager` from M to operate with `employee`-like privileges on the `retailer service` managed by R, we need to restrict the privilege to only those items in the retailer's warehouse that are procured from the manufacturer M. For instance, a manufacturer M should not able to read the inventory level of some item I that is supplied by another manufacturer M'. Hence, an access control model needs to support the notion of *scoped roles* by tagging a role (`employee`) with a scope (manufacturer M's ID) to restrict the privileges of translated roles.

Another key requirement for large applications is the enforcement of *separation of duty* (SoD) constraints. For example, an application may require that a principal should not be able to both approve an order and process its payment. However, a principal should be able to approve an order o_1 and process the payment for an order o_2 if $o_1 \neq o_2$. We support separation of duty constraints via *scoped access control rules*. Scoping allows parameterization of the access control rule and thus imposes the rule only within the scope (the order ID in this example).

Apart from the above features which are geared specifically towards service compositions, we support standard *Boolean predicates* on roles, method names and method argument values for data-driven access control. For an example rule could be: an `employee` cannot approve an order if the order cost is larger than a threshold c.

[1] This model is similar to the one used by WalMart (retailer) and Procter & Gamble (manufacturer) [23]

3 Access Control Model

In this section we present our access control model for service compositions. Our model uses the notion of composite principals to abstract the relevant temporal, causal and privilege (roles) information required for enforcing access control rules for such activities. In the rest of this section, we formally define composite principals and use it as a building block for reasoning about separation of duty constraints and inter-organizational service invocations.

3.1 Composite Roles and Principals

A composite role (CR) or a composite principal (CP) allows us to reason about Web service compositions as first class entities. Recall that a principal P refers to a person, user or client and a role R refers to a job function or title which defines an authority level. A composite role consists of a temporally ordered sequence of roles and services that are involved in a transaction. Similarly, a composite principal consists of a temporally ordered sequence of principals (playing a certain role) and service instances (acting as a certain service) that are involved in a transaction. Concretely, a composite role and a composite principal are represented in BNF form as shown below.

$$CR \; := \; (S \mid R)^{+}$$
$$CP \; := \; (SI \; as \; S \mid P \; as \; R)^{+}$$

Temporal Constraint. For example, composite roles may capture the fact that a write on the `order` database table is performed by CR = (`customer`, `retail service`). The statement CR = (`customer`, `retail service`) invokes a method M on the `database service` should be read as: a `customer` operating via the `retail service` invoked a method M on the `database service`. This allows us to explicitly deny write operations on the database to composite roles CR_1 = (`customer`) and CR_2 = (`retail service`).

3.2 Separation of Duty Constraints

Composite roles abstract away the concrete principals (and service instances) that participate in a transaction. We capture separation of duty (SoD) constraints on concrete principals using composite principals. Unlike a service invocation based constraint, a SoD constraint encompasses an activity. An activity is modeled as a temporally ordered sequence of transactions; note that each of these transactions is in turn associated with a composite principal making that invocation. An activity A is represented in BNF as:

$$A \; := \; CP^{+}$$

Temporal Constraint. Suppose in an `order` activity, a composite principal CP^{o}_{app} = (`emp`$_1$ `as employee`, `rs`$_1$ `as retail service`) has approved a customer order o. The specifications for an order activity states that before the order o is approved, its payment needs to be verified. Let us suppose that a composite principal CP^{o}_{pay} = (`emp`$_1$ `as employee`, `rs`$_1$ `as retail service`) attempts to authorize the payment for order o. The temporal constraint that follows from the specification is that CP^{o}_{pay} should precede CP^{o}_{app}.

Scope Constraints. SoD constraints typically span across multiple service invocations (multiple composite principals), but are all related to one *scoped* activity. In the above example, the scoped activity is a `customer order` and its scope is a unique order identifier. A scoped SoD constraint is represented as the following Boolean constraint: $(o_1 = o_2) \Rightarrow CP^{o_1}_{app}.emp \neq CP^{o_2}_{pay}.emp$.

3.3 Inter-Organization Service Invocations

We implement access control in inter-organizational Web service invocations through role translation. An organization org_1 defines these role translations in the form of a table that maps a role R_2 in org_2 to some role R_1 that is understood by the access control service in org_1. Formally, this is represented as a mapping: $R_2^{org_2} \rightarrow R_1^{org_1}$.

Scoped Roles. In many instances, however, the translated role needs to be scoped by the identity of the organization the concrete principal belongs to. Formally, this is represented as a mapping: $R_2^{org_2} \rightarrow R_1^{org_1}\langle org_2 \rangle$. For example, a manufacturer M's `inventory manager` may be mapped by the retailer system to the role of a scoped employee: `employee`$\langle M \rangle$. We can represent the fact that the scoped role `employee`$\langle M \rangle$ has the status of an `employee` only for those items that are purchased from manufacturer M using Boolean constraints on the scope M.

4 Access Control Specification

Having described the basic building blocks we next present an access control specification language that integrates them with the goal of meeting all the requirements discussed in Section 2.

4.1 Specification Language

In Section 3, we modeled a composite principal as a temporally ordered sequence of principals or service instances that are responsible for a service invocation. Such temporally ordered structures may be viewed as finite models of linear temporal logic (LTL) [19]. In this paper, we specify access control policies using pure-past linear temporal logic based specification language. The pure-past variant of LTL [13] does not include *future* temporal operators and is sufficient for expressing our access control policies. In the rest of this section, we present a declarative language that is suitable for representing our access control policies.

The syntax of our access control language is specified by the following BNF for Kripke structures [19]. Note that p is an atomic proposition. The operators X^{-1} (*last time*) and S (*since*) are the past time temporal operators: $X^{-1}\psi$ is `true` if and only if ψ were `true` in the previous time step and $\psi_0 S \psi_1$ is `true` if and only if ψ_1 was `true` at some point in the past and ψ_0 has been `true` at all points in time since ψ_1 evaluated to `false` (more rigorous definition in equation 1).

$$\psi := p \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid X^{-1}\psi \mid \psi_0 S \psi_1$$

We specify access control rules on service invocations and SoD constraints using propositions that are constructed as Kripke structures. For access control rules on service invocations, we define a satisfaction relation $\models$ between the policy ψ and a composite role CR. A composite role CR can invoke the method M only if $CR \models \psi$. Let $CR = (x_1, x_2, \cdots, x_N)$, where each x_i is either a role or a service. Then, we say that $CR \models \psi$ if and only if $(CR, |CR|) \models \psi$. We define $(CR, i) \models \psi$ by structural induction [13] on ψ as follows:

$$
\begin{aligned}
(CR, i) &\models p \text{ iff } p \in x_i \\
(CR, i) &\models \psi_0 \wedge \psi_1 \text{ iff } (CR, i) \models \psi_0 \text{ and } (CR, i) \models \psi_1 \\
(CR, i) &\models \psi_0 \vee \psi_1 \text{ iff } (CR, i) \models \psi_0 \text{ or } (CR, i) \models \psi_1 \\
(CR, i) &\models \neg\psi \text{ iff } (CR, i) \not\models \psi \\
(CR, i) &\models X^{-1}\psi \text{ iff } (CR, i-1) \models \psi \\
(CR, i) &\models \psi_0 S \psi_1 \text{ iff } \exists j \leq i, [(CR, j) \models \psi_1 \text{ and } \\
&\quad \forall k, (j < k \leq i \Rightarrow (CR, k) \models \psi_0)]
\end{aligned} \quad (1)
$$

4.2 Sample Policies

We now show how one can use our policy specification language to encode some sample access control policies. Let us consider the order approval process in the retailer system with the following list of access control rules:

- An employee operating through the retail service can approve an order if the order cost is less than c.
- A retail manager operating through the retail service can approve all orders.
- A chief manager can approve all orders.

For notational convenience we introduce a temporal operator F^{-1} such that $F^{-1}(\psi)$ = true S ψ; that is, the proposition ψ was true at some point in the past. We also use a shorthand Boolean operator $\Rightarrow$ such that $\psi_1 \Rightarrow \psi_2 = \neg\, \psi_1 \vee \psi_2$. We encode the above policies as proposition ψ:

$$
\begin{aligned}
\psi &= \psi_0 \vee \psi_1 \vee \psi_2 \\
\psi_0 &= (F^{-1}(\texttt{employee}) \wedge X^{-1}(\texttt{retailservice}) \wedge \\
&\quad (\texttt{ordercost} < c)) \\
\psi_1 &= (F^{-1}(\texttt{retailmanager}) \wedge X^{-1}(\texttt{retailservice})) \\
\psi_2 &= F^{-1}(\texttt{chiefmanager})
\end{aligned} \quad (1)
$$

Note that we evaluate the policy ψ against a composite role that attempts to invoke an order approval method and the method arguments (ordercost in this example). Note that $F^{-1}(\texttt{employee})$ means that some principal playing the role of an employee should have initiated the order approval procedure. Also, $X^{-1}(\texttt{retailservice})$ means that the last step of the order approval process has been performed under the supervision of the retail service. Jointly, $F^{-1}(\texttt{employee}) \wedge X^{-1}(\texttt{retailservice})$ would ensure that the order approval process was initiated by an employee and was last verified by the retail service. A composite role CR = (retail manager, retail service) invoking the order approval method with order cost greater than c is checked against the policy ψ described above as follows. Note that since all retail managers are employees, $CR \models F^{-1}$(employee). Hence, the rule ψ_0 evaluates to false. On the other hand, the rule ψ_1 evaluates to true and rule ψ_2 evaluates to false and thus the rule ψ evaluates to true.

Let us consider the separation of duty constraint based on the following access control policies:

- An order is approved only after its payment is verified.
- A employee or a retail manager can verify a customer's payment for an order as long as the concrete principal initiating the payment verification and the order approval are not the same.
- A chief manager can approve and verify any order.

We can encode these access control policies using the proposition ψ described below.

$$
\begin{aligned}
\psi^o &= \psi_0^o \wedge (\psi_1^o \vee \psi_2^o) \\
\psi_0^o &= (F^{-1}(CP_{pay}^o) \wedge X^{-1}(CP_{app}^o)) \\
\psi_1^o &= ((CP_{app}^o \models F^{-1}(\texttt{employee}) \wedge \\
&\quad (orderApprover(o) \neq paymentAuthorizer(o)) \\
\psi_2^o &= (CP_{app}^o \models F^{-1}(\texttt{chiefmanager}))
\end{aligned} \quad (2)
$$

We evaluate the policy ψ^o against a temporally ordered sequence of composite principals (SCP) that have participated in the order activity. In this example, a valid $SCP^o = (CP_{pay}^o, CP_{app}^o)$, where CP_{pay}^o is the composite principal that verified the payment for order o and CP_{app}^o is the composite principal that makes a service invocation to approve and commit the order o. The constraint ψ_0^o ensures that an order o can be approved by CP_{app}^o only if its payment has been verified in the past by CP_{pay}^o. The predicate $CP_{app}^o \models F^{-1}(\texttt{employee})$ evaluates to true if the approval process was initiated either by an employee or by a retail manager. In this case, the rule ψ_1^o imposes a SoD constraint on the concrete principals concerned. For notational convenience, we have used $orderApprover$ and $paymentAuthorizer$ as macros defined on SCP^o.

Let us consider access control policies on inter-organizational Web service invocation with role translation:

- A manufacturer M's inventory manager has the same status as that of an employee in the retailer system.
- A manufacturer M's inventory manager is scoped to operate on only those items that are purchased by the retailer system from the manufacturer M.

$$
\begin{aligned}
\psi &= (\psi_0 \wedge \psi_3\langle M\rangle) \vee \psi_1 \vee \psi_2 \\
\psi_0 &= (F^{-1}(\texttt{employee}) \wedge X^{-1}(\texttt{retailservice}) \wedge \\
&\quad (\texttt{ordercost} < c)) \\
\psi_1 &= (F^{-1}(\texttt{retailmanager}) \wedge X^{-1}(\texttt{retailservice})) \\
\psi_2 &= F^{-1}(\texttt{chiefmanager}) \\
\psi_3\langle M\rangle &= (F^{-1}(\texttt{employee}\langle\texttt{M}\rangle) \Rightarrow (manufacturer(M) \wedge \\
&\quad purchase(\texttt{itemID}, M))
\end{aligned} \quad (3)
$$

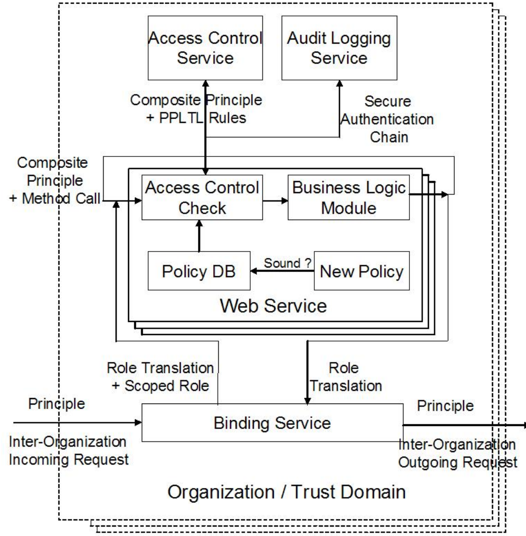

Figure 2: Access Control in Service Compositions

We encode these access control policies using a role translation table and a PPLTL rule ψ described above. The role translation table maintained by the retailer system's gateway maps M's `inventory manager` to a scoped role `employee`$\langle M \rangle$ in the retailer system.

We evaluate the policy ψ against a composite role that attempts to invoke an order approval method and the method arguments (`ordercost` and `itemID`). For notational convenience, we use the following macros: $manufacturer(M)$ is `true` if M is a manufacturer and $purchase$(`itemID`, M) is `true` if the retailer system purchases the item `itemID` from the manufacturer M. Note that the first three constraints ψ_0, ψ_1 and ψ_2 comprise a policy that applies to all `employees` in the retailer system; the fourth constraint $\psi_3\langle M \rangle$ applies to a scoped role `employee`$\langle M \rangle$, say an `inventory manager` from a manufacturer M.

4.3 Policy Evaluation

The problem of verifying a policy with respect to a composite role (or a sequence of composite principals) is a model checking problem: does $CR \models \psi$ hold? Havelund and Rosu [11] have recently presented an efficient memorization based dynamic programming algorithm for model checking PPLTL with time complexity of $\Theta(|CR| * |\psi|)$ and space complexity 2*$|\psi|$ + 1 bits. Note that $|CR|$ denotes the length of the composite role CR and $|\psi|$ denotes the length PPLTL proposition ψ.

5 Access Control Implementation

In this section, we describe an implementation of our access control module in an application overlay network (AON) hosting the SCM application. AON strongly advocates the idea of building middleware features as light-weight on-demand services. An application can dynamically discover and use these services as interoperability requirements are determined. A single logical application can itself be partitioned and managed across the network. Figure 1 shows an AON based implementation of the SCM application. The SCM application consists of two AONs: a retailer system and a manufacturer system. The retailer system has several nodes that host middleware functionality such as: binding/gateway service, audit/logging service, JavaScript engine as a service, database engine as a service and business logic services (retail service and warehouse services).

```
⟨complexType name=''Service''⟩
  ⟨sequence⟩
    ⟨element minOccurs=''0'' maxOccurs=''unbounded''
     ref=''sca:operation''/⟩
    ⟨element minOccurs=''0'' maxOccurs=''unbounded''
     ref=''sca:accessControlRule''/⟩
    ...
  ⟨/sequence⟩
⟨/complexType⟩

⟨complexType name=''accessControlRule''⟩
  ⟨attribute name=''name'' type=''string''
   use=''required''/⟩
  ⟨attribute name=''rule'' type=''string''
   use=''optional'' default=''true''/⟩
⟨/complexType⟩
```

Figure 3: Extensions to the Service Schema

```
⟨service name=''retailer''⟩
  ⟨interface.wsdl interface=''http://ws-i.org/···/Retailer.wsdl''/⟩
  ⟨operation.js name=''processOrder'' onInvoke=''order''/⟩
  ⟨accessControlRule name=''processOrder'' rule=''((F(employee)
  ∧ X(retailservice) ∧ (ordercost(cost, c)) ∧ (F(employee) ⇒
  (manufacturer(M) ∧ purchase(itemID, M))) ∨
  (F(retailmanager) ∧ X(retailservice)) ∨ F(chiefmanager))''/⟩
  ...
⟨/service⟩
```

Figure 4: Access control rule on `processOrder`

5.1 Access Control Service

Figure 2 shows our implementation of the access control service in an AON. The `binding/gateway service` first receives a service invocation. It performs role translation and forwards the request to the appropriate service (say S) on its AON. Service S on receiving a method invocation first checks if the invocation is permissible using the access control module. Note that an invocation to the access control service may require some invocation history information to be retrieved from the `logging service`. The access control module is itself implemented as a middleware feature that can be hosted as a service by the AON. The service container invokes an access control service (possible hosted on an entirely different container and a thin server) to perform the required access control check. Once the access check passes, the service S executes the invocation on its business logic module; this execution may trigger further method invocations on services in the same AON or an inter-organizational call to a service in an entirely different AON.

The `access control service` exports an interface as shown in Figure 5. In the following portions of this section, we describe our instrumentations to the underlying AON infrastructure to support the `access control service`. Our implementation requires no changes to the application code. We focus on the four parameters required by our `access control service` interface (Figure 5), namely, the access control rule ψ, the composite principal CP, the method signature sig and the method's input values $input$.

```
Boolean verify(AccessControlRule ψ, CompositePrincipal CP,
               MethodSig sig, XMLObject input)
```

Figure 5: Access control Service Interface

Access Control Rule ψ. We have modified the service description schema used by AON to include one or more access control rules for every service interface (see Figure 3). The access control rule itself is a $name$ and $rule$ tuple, where $name$ refers to the name of the operation/method and $rule$ refers to the PPLTL rule that must be satisfied in order for an invocation on method $name$ to be permissible (see example in Figure 4). This framework achieves a clear separation and language independence between the application logic and the access control policy specification. It also permits highly flexible and fine grained (method level) specification of access control rules. Finally, bundling access control rules for a service along with the service description respects the autonomous nature of each Web service.

Composite Principal CP. The composite principal invoking a service is obtained by instrumenting the service containers in the AON infrastructure. A service container in the AON receives a request from a caller, triggers the service to process the request, and handles subsequent nested service invocations or returns to the caller. We instrument the service containers to construct a temporally ordered list of principals and services involved in a service invocation.

For access control rules that span multiple composite principals (say, SoD constraint in Policy 2), the `access control service` uses the `logging service` to obtain the composite principal(s) that have participated in the activity of interest in the past. We have modified the service container to log service invocations along with the scope name and scope ID using the `logging service`. The service container infers a scope name from the policies in a service description (say, M in Figure 4). The scope ID is obtained from the argument values in the service invocation such that the argument name matches the scope name.

Method Signature sig and Inputs $input$. The method signature sig contains a WSDL description of the invoked method's signature. The `XMLObject` $input$ contains method invocation arguments. Coupled with the method signature sig, it enables the `access control service` to interpret the argument types and values in $input$. The sig and $input$ parameters are required to support data-driven access control policies (see `orderCost` constraint in policy 1).

In the following sections, we describe how we deploy and enforce access control rules in our system.

5.2 Policy Deployment

Let us consider a sample policy ψ described by Equation 3 in Section 4.2. We deploy the policy ψ through the following steps. The deployment process is completely automated other than determining the policy enforcement points in step 2 (similar to other approaches like [18, 12]) and the relevant role translation rules in step 3.

1. Given a policy ψ the administrator has to determine policy enforcement points (PEPs). In this case the admin determines that the policy must be enforced at the `processOrder` method in the `retailer service` and the `gateway service`.

2. Add the new policy ψ as an XML component to the PEP's service description as shown in Figure 4. Note that the policy ψ is stored along with the service description and not at the `access control service`.

3. The administrator adds a role translation rule to the gateway's policy DB. A role translation rule is a three tuple: ⟨organization-name, role-name, translated-role⟩. In this example a table entry would look like ⟨`PG`, `inventory manager`, `employee⟨PG⟩`⟩, where PG is the manufacturer (Procter and Gamble).

5.3 Policy Enforcement

The deployed access control policies are enforced as follows. When the retailer system's gateway service receives a Web service invocation on the method `processOrder` it automatically enforces access control policies as follows:

1. The gateway service performs role translation. The translation is achieved by replacing every occurrence of `inventory manager` in the composite principal with `employee⟨PG⟩`.

2. The gateway service forwards the method invocation to the container hosting the `retailer service`. The service runtime looks up the service description (Figure 4) to check if there are any rules associated with the operation `processOrder`. If so, it invokes the `access control service` with the following arguments: the composite principal CP, the rule ψ, and the arguments contained in the call to the `processOrder` method and a reference to the WSDL description of the `processOrder` method.

3. The model checker parses the rule ψ (one linear scan) and breaks it down into atomic predicates. For the sample policy in Figure 4, the atomic predicates are shown in Figure 6. Note that the policy ψ in Figure 4 is equivalent to ψ_{22} in Figure 6. Given an atomic predicate it can be categorized into one of the following five types: a role or service (like `employee`, `retail service`), a scoped role (like `employee`$\langle M\rangle$), Boolean operators ($\wedge$, $\vee$, $\sim$, $\Rightarrow$), temporal operators (F^{-1}, X^{-1}), and macros ($ordercost$, $manufacturer$, $purchase$).

3a. A scoped role is used to extract the scope variable. The value for a scoped variable (if any) is obtained from a linear scan on the composite principal. In this case, the scope variable is M and its value is `PG`.

3b. We evaluate macros using method calls that return a Boolean value. In this case there are three macros: $ordercost$, $manufacturer$ and $purchase$. We use reflection to determine the Java method from the macro name; then we pass arguments to these methods from three sources: (i) scope resolution (in this case, M = PG), (ii) arguments in the call on the `processOrder` method (in this case: $cost$, $itemID$), and (iii) policy specified constants (in this case, the cost threshold, c). Scope resolutions are handled as follows: (i$_a$) a scoped role is already inferred in step 3a, and (i$_b$) a scoped activity requires the `access control service` to look up the `logging service` to identify past service invocations (composite principals) that have participated in the scoped activity.

3c. A role or a service is an atomic literal. These literals are

ψ_0 = `employee`	$\psi_1 = F^{-1}\psi_0$	ψ_2 = `retail service`	$\psi_3 = X^{-1}\psi_2$
$\psi_4 = \psi_1 \wedge \psi_3$	$\psi_5 = ordercost(cost, c)$	$\psi_6 = \psi_4 \wedge \psi_5$	ψ_7 = `employee`$\langle M \rangle$
$\psi_8 = F^{-1}\psi_7$	$\psi_9 = manufacturer(M)$	$\psi_{10} = purchase(itemID, M)$	$\psi_{11} = \psi_9 \wedge \psi_{10}$
$\psi_{12} = \psi_8 \Rightarrow \psi_{11}$	$\psi_{13} = \psi_6 \wedge \psi_{12}$	ψ_{14} = `retail manager`	$\psi_{15} = F^{-1}\psi_{14}$
ψ_{16} = `retail service`	$\psi_{17} = X^{-1}\psi_{16}$	$\psi_{18} = \psi_{15} \wedge \psi_{17}$	$\psi_{19} = \psi_{13} \vee \psi_{18}$
ψ_{20} = `chief manager`	$\psi_{21} = F^{-1}\psi_{20}$	$\psi_{22} = \psi_{19} \vee \psi_{21}$	

Figure 6: Parsing an Access Control Rule

evaluated using the role hierarchy predicates. We use PPLTL to evaluate temporal operators (see Equation 1). The Boolean operators are trivial to evaluate.

4. If the access control service returns `true`, then the runtime forwards the method invocation to the `retailer service`; else the method invocation fails.

5.4 Experimental Results

In this section, we report our results from experiments with the access control service. These experiments were run on 2.4GHz Windows XP boxes hosting the SCM application. Figure 7 shows rule verification time versus rule size for a service invocation chain of length two ($|CR|$ = 2). The size of a rule ψ is defined as the sum of the number of literals and the number of operators (Boolean, temporal and macros) in ψ. For example, the policy in Figure 4 is of size $|\psi|$ = 22, see Figure 6. Observe that the policy verification overhead on an application's critical path (verify) is only of the order of a few milliseconds.

Figure 8 shows that the user perceived latency for the three policies described in Section 4.2 is about 10-24%. Figure 9 shows the cost break up for the access control overhead. The bars in this graph mean the following:

RT (role translation): Policies 1 and 2 require no role translation and thus incur no role translation cost.

RL (rule lookup): Rule lookup incurs a small hash table lookup cost.

PR (parse rule): Parsing a rule requires a linear scan of the PPLTL rule and incurs $O(|\psi|)$ cost. Note that policy 1 is the shortest policy and policy 3 is the longest policy.

SR (scope resolution): Scope resolution for roles requires a linear scan on the composite principal and incurs $O(|CP|)$ cost. Policy 1 requires no scope resolution, while policies 2 and 3 need to be dynamically instantiated for the given scope (o for policy 2 and M for policy 3).

EA (extract argument types and values using WSDL reference): Extracting arguments incurs higher costs for parsing the WSDL reference. Policy 2 has no constraints on method arguments, while policy 1 has one argument ($cost$) and policy 3 has two arguments: $cost$ and $itemID$.

EC (extract constant types and values from policy): Extracting constant values requires an inexpensive lexical analysis on the policy. The policies 1 and 3 use a constant for the order cost threshold.

EM (evaluate macros): Evaluating a macro depends on the concrete macro: policy 1 includes an $orderCost$ macro which performs a simple comparison operation, while policy 2 includes a macro on service invocations CP^o_{pay} and CP^o_{app}. The composite principal CP^o_{pay} has to be obtained from a lookup on the audit/logging service. Policy 3 includes two macros, $manufacturer$ and $purchase$, which are evaluated as database queries and are thus the most expensive.

MC (PPLTL model check): Rule verification time incurs a small $O(|CP| * |\psi|)$ cost.

6 Related Work

A large body of work exists in the field of authentication and access control for autonomous distributed systems. The seminal work by Lampson [14] established the foundations for access control in distributed systems. Role based access control models [20] introduced the notion of roles to allow clean separation between users, roles and objects.

The recent emergence of service oriented architectures and composable Web services add several new security challenges. WS-I (Web services interoperability) presents detailed design and implementation of a supply chain management (SCM) application [24]. WS-Security [17] describes enhancements to SOAP messaging to provide message integrity, confidentiality, and single message authentication in a way that can accommodate a wide variety of security models and encryption technologies. WS-Policy [3] provides a general purpose model and a specification language to describe and communicate the policies of a Web service. WS-Security Policy [8], built on the WS-Policy and the WS-Policy Assertion [4], is a declarative XML format for programming the precise techniques used by Web service implementations to construct and check WS-Security [17] headers.

Abadi et al [1] propose a framework that uses the concept of delegation for authentication and access control in loosely distributed systems. The OASIS architecture presents a policy description language to reason about roles and delegations in autonomous distributed systems [12]. XACML is a markup language designed to express well-established ideas from OASIS in access control using XML extensions [18]. However, XACML does not provide explicit constructs to reason about transactional histories. Bhargavan et. al. [16] propose techniques to verify security policies on Web services. These approaches neither treat service composition as a first class entity nor provide a unified methodology for expressing and enforcing access control rules in large and complex Web service compositions.

Security automata [10] and the notion of history based access control (HBAC) [9, 13] have been subject to a considerable amount of research. Several JVM based security papers present stack introspection techniques for implementing access control policies [2]. Several authors have developed access control policies on stand-alone Web services. Mecella et al [15] present

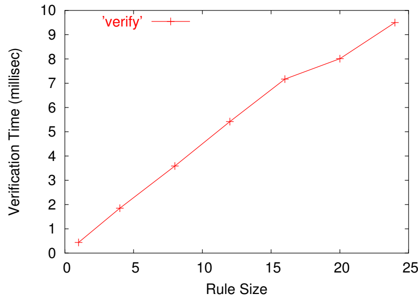

Figure 7: Policy Verification Cost

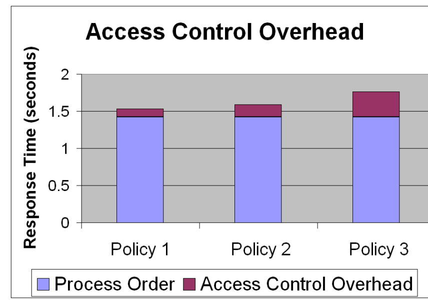

Figure 8: Application Latency

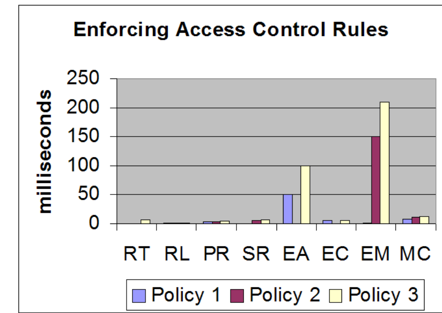

Figure 9: Cost Break Up

techniques to incrementally exchange credentials in a Web service conversation. Sirer and Wang [21] present techniques to use linear temporal logic (LTL) to enforce sanity checks and detect possible security violations on a Web server.

Carminati et al [5] describe techniques to construct a composition that satisfies a certain property given policy level specifications for several component services. Our paper focuses on one such security property, namely access control, and presents not only policy level specifications of access control policies, but also a concrete specification language, a sound and efficient verification mechanism, a deployment and enforcement methodology, and an implementation.

7 Conclusion

We have presented an access control model and a language based on this model for specifying access control policies; our approach is particularly well suited for Web service compositions. We have demonstrated techniques to handle access control rules based on composition history, separation of duty constraints, and inter-organizational roles. We have described a policy specification language for describing these access control policies using pure-past linear temporal logic (PPLTL). We have also described an implementation of the access control module as a middleware service on an application overlay network (AON). Our experiments show that our approach can enforce highly flexible and expressive access control policies while incurring reasonable performance overheads (10-24%) on the application.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. In *ACM TOPLAS, 15(4), pp 706-734*, 1993.

[2] M. Abadi and C. Fournet. Access control based on execution history. In *ISOC NDSS*, 2003.

[3] D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, J. N. Nagaratnam, M. Nottingham, C. V. Riegen, and J. Shewchuk. Web services policy framework. http://www.ibm.com/developerworks/library/ws-polfram/.

[4] D. Box, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, J. N. Nagaratnam, P. patrick, C. V. Riegen, and J. Shewchuk. Web services policy assertions language. http://www.ibm.com/developerworks/library/ws-polas/.

[5] B. Carminati, E. Ferrari, and P. C. K. Hung. Security conscious web service composition. In *ICWS*, 2006.

[6] F. Curbera, D. F. Ferguson, M. Nally, and M. L. Stockton. Towards a programming model for service-oriented computing. In *ICSOC*, 2005.

[7] DAML. Automating DAML-S services composition using SHOP2. In *2nd ISWC*, 2002.

[8] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, J. N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, and E. W. Zolfonoon. Web services security policy language. http://www.verisign.com/wss/WS-SecurityPolicy.pdf.

[9] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *5th ACM CCS*, 1998.

[10] P. W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy, pp. 43-55*, 2004.

[11] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *8th TACAS, Vol 2280 LNCS, pp 342-356*, 2002.

[12] R. J. Hayton, J. M. Bacon, and K. Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, 1998.

[13] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history based access control. In *12th ACM CCS*, 2005.

[14] B. W. Lampson. Protection. In *Proceedings of 5th Princeton Symposium on Information Sciences and Systems, pp. 437-443*, 1974.

[15] M. Mecella, M. Ouzzani, F. Paci, and E. Bertino. Access control enforcement for conversation-based web services. In *15th WWW Conference*, 2006.

[16] N. Mukhi and P. Plebani. Supporting policy-driven behavior in web services: Experiences and issues. In *ICSOC*, 2004.

[17] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. OASIS web services security: SOAP message security 1.0. http://oasis-open.org.

[18] Oasis. OASIS extensible access control markup language (XACML). http://www.oasis-open.org.

[19] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on FOCS*, 1977.

[20] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. In *IEEE Computer, Vol. 29, No. 2*, 1996.

[21] E. G. Sirer and K. Wang. An access control language for web services. In *7th ACM SACMAT*, 2002.

[22] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service description with compositional specification. In *13th WWW Conference*, 2004.

[23] B. Worthen. Supply chain management research. http://www.cio.com/research/scm/edit/012202_scm.html.

[24] WSI. Supply chain management use case model. http://www.ws-i.org/.