Support for ANSI RBAC in CORBA

Konstantin Beznosov and Wesam Darwish

Laboratory for Education and Research in Secure Systems Engineering
lersse.ece.ubc.ca

Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

Technical report LERSSE-TR-2007-01*

`Revision: #61`

`Revision Date: 2007/07/26`

*This and other LERSSE publications can be found at http://lersse-dl.ece.ubc.ca

Abstract

We describe access control mechanisms of the Common Object Request Broker Architecture (CORBA) and define a configuration of the CORBA protection system in more precise and less ambiguous language than the CORBA Security specification (CORBASec). Using the configuration definition, we suggest an algorithm that formally specifies the semantics of authorization decisions in CORBA. We analyze support for the American National Standard Institute's (ANSI) specification of Role-Based Access Control (RBAC) components in CORBA and identify the functionality that needs to be implemented—in addition to compliance with the CORBASec—in order to support Core, Hierarchical, and Constrained RBAC. We illustrate the discussion with a single access-policy domain as well as a multi-domain examples of the CORBASec protection system configuration. We also analyze support for the functional specification of ANSI RBAC in CORBA.

Our results indicate that CORBA Security falls short of supporting even Core RBAC. Custom extensions are necessary in order for implementations compliant with CORBA Security to support ANSI RBAC required or optional components. These results can be interpreted as either a demonstration of CORBA's inadequacy in supporting ANSI RBAC, or as a sign of ANSI RBAC not being sufficiently general. This paper sets up a framework for implementing and assessing implementations of ANSI RBAC using CORBA Security, provides directions for CORBA Security implementing ANSI RBAC in their systems, and offers criteria to users for selecting these CORBA Security implementations that support required and optional components of ANSI RBAC.

Contents

1 Introduction

The American National Standard Institute's (ANSI) specification of Role-Based Access Control (RBAC) [ANS04] is a standard for access control in which permissions are associated with roles and users are assigned to appropriate roles. A role can represent competency, authority, responsibility or specific duty assignments. A major purpose of RBAC is to facilitate access control administration and review. RBAC is commonly believed to address the needs of commercial enterprises better than lattice-based Mandatory Access Control (MAC) [BL75] and owner-based Discretionary Access Control (DAC) [Lam71] models. Moreover, Osborn et al. [OSM00] show that an RBAC system can indeed be configured to enforce either a DAC or a MAC policy. Evidence of RBAC becoming a dominant access control paradigm is the approval of ANSI RBAC standard in 2004. The ANSI RBAC standard consists of two main parts: (1) the RBAC Reference Model and (2) the RBAC System and Administrative Functional Specification, each comprising core, hierarchical, and constraint components.

At the same time as RBAC was introduced and evolving into a mature model ready for standardization, commercial middleware technologies—such as Common Object Request Broker Architecture (CORBA) [OMG99], COM+ [Obe00], and Enterprise Java Beans (EJB) [DYK01]—also matured, and distributed enterprise applications became routinely developed using of middleware. The ability of particular middleware technology to support specific types of access control policy is an open and practical question, for the following three reasons.

First, different middleware technologies and their subsystems are defined in different forms and formats. For example, CORBA is specified in the form of open application programming interfaces (APIs), whereas EJB is defined through APIs as well as the syntax and semantics of the accompanying eXtensible Markup Language (XML) files used for configuring an EJB container. COM+ is defined through an implementation of APIs as well as graphical user interfaces (GUI) for configuring the behavior of a COM+ server on Windows NT, 2000, 2003, XP, and Vista operating systems. The variations in the form, terminology, and format of the middleware definitions and implementations lead to the difficulty of identifying the correspondence among the security (and other) capabilities of any two middleware technologies as well as the degree to which they can support a particular access control model.

Second, the capabilities of the middleware security controls are not defined in the language of any particular access control model. Instead, each middleware provides general access control mechanisms, which are supposed to be adequate for the majority of cases and scenarios, and could be configured to support various access control models. As a case in point, Karjoth demonstrates how CORBASec can be configured to support lattice-based MAC [Kar00]. Designed to support a variety of policy types as well as large-scale, diverse distributed applications, the controls seem to be the result of engineering compromises involving, among other factors, perceived customer requirements, the capabilities of the target run-time environment, and their expected usage. For example, CORBA access controls are defined in terms of *principal's attributes*, *required rights*, and *granted rights*, whereas EJB controls are defined using *role mappings* and *role-method permissions*. Assessing the capability of middleware controls to enforce particular types of authorization policies is harder due to the mismatch in terminology between the published access control models and the languages of the controls.

Third, the security subsystem semantics in commercial middleware is defined imprecisely, sometimes ambiguously, leaving room for different interpretations. In this paper, we clarify the semantics of the security subsystem and analyze its ability to support ANSI RBAC for one particular industrial middleware technology—CORBA.

The contribution of this paper is twofold. First, we define the *protection state* of the CORBA access control subsystem—as specified in CORBA Security v.1.8 [OMG02b] (CORBASec for short)—in a formal language through studying its descriptions and specification. Our definitions offer a more precise and less ambiguous interpretation of the CORBA access controls and fill in the gap in the CORBASec specification, which uses only English prose and Interface Definition Language (IDL) to describe access control in CORBA. The language of the CORBA protection state enables the analysis of CORBASec on the subject of its support for specific access control models.

Second, to demonstrate the utility of the protection state definition and, more importantly, to aid application developers and owners, we analyze the degree to which CORBA supports the reference model and functional specification defined by ANSI RBAC [ANS04]. We use the language of the protection state configuration to analyze CORBASec in relation to its support for particular ANSI RBAC features, e.g., role hierarchies. Where possible, we show how the corresponding ANSI RBAC construct can be expressed in the language of the CORBA protection state or CORBASec operations. In cases where support for a specific ANSI RBAC feature requires implementation-dependent functionality, we explicitly state what needs to be implemented by the CORBASec vendors, or enforced by the security administrators. When we cannot identify the means of supporting an RBAC feature, we state so. We also suggest steps for translating an arbitrary ANSI RBAC policy into CORBA protection state.

Although there has been no shortage of papers proposing ways of supporting earlier models of RBAC, e.g., RBAC96 [SCFY96], in operating systems [SGJ98, Fad99, AS01, Sun00, Cha03], databases [RS98], and distributed systems [GDS97, Bar97, Giu99, FBK99, Ahn00, Gut01, PSA01, CO02, OF02, ZM04], to the best of our knowledge, this is the first work that offers detailed analysis of support for both reference model and functional specification of ANSI RBAC by an industrial technology in general, and a middleware in particular.

The results of our analysis indicate that CORBASec falls short in supporting even functional Core RBAC due to (1) the lack of a standard mechanism for enumerating all policy objects in a CORBA deployment, (2) the lack of explicit user representation as well as the notion of user accounts and support for their management, and (3) the inability to enumerate all CORBA principals related to a specific user. Custom extensions are necessary in order for implementations compliant with CORBASec to support ANSI RBAC components. These results can be interpreted as either a demonstration of the inadequacy of CORBASec in supporting ANSI RBAC, or as a sign of ANSI RBAC being over-engineered. Examination of support for ANSI RBAC in other representative systems may clarify this question.

The work presented in this paper establishes a framework for implementing and assessing implementations of ANSI RBAC using CORBA Security. The results provide directions for CORBA Security developers supporting ANSI RBAC in their systems and offer criteria to users and application developers for selecting those CORBA Security implementations that support required and optional components of ANSI RBAC.

The rest of the paper is organized as follows. Section 2 provides background on ANSI RBAC and CORBA Security. Section 3 discusses related work. Section 4 describes the CORBA Security access control architecture and formally defines the protection state of a CORBA system. Section 5 examines the extent to which the CORBASec can support ANSI RBAC model components, and analyzes the degree to which various CORBASec programming interfaces can support the functional specification of ANSI RBAC. We discuss results of our analysis in Section 6. Section 7 concludes the paper.

2 Background

This section provides background on ANSI RBAC and CORBA Security that is necessary for understanding the rest of the paper. Readers familiar with both can skip directly to Section 3.

2.1 Overview of ANSI RBAC

Role based access control (RBAC) was introduced more than a decade ago [FK92, SCFY96]. Over the years, RBAC has gained a lot of attention as many research papers were written on topics related to RBAC; and in recent years, vendors of commercial products have started implementing various RBAC features in their solutions.

The National Institute of Standards and Technology (NIST) initiated a process to develop a standard for RBAC to achieve a consistent and uniform definition of RBAC features. An initial draft of a consensus standard for RBAC was proposed in the year 2000 [SFK00]. A second version was later publicly released in 2001 [FSG+01]. This second version was then submitted to the InterNational Committee for Information Technology Standards (INCITS), where further changes were made to the proposed standard. Lastly, INCITS approved the standard for submittal to the American National Standards Institute (ANSI). The standard was later approved in 2004 [ANS04]. The ANSI RBAC standard consists of two main parts as described in the following sections.

2.1.1 Reference Model

The RBAC Reference Model defines sets of basic RBAC elements, relations, and functions that the standard includes. This model is defined in terms of four major RBAC components as described in the following sections.

Core RBAC

Core RBAC defines the minimum set of elements required to achieve RBAC functionality. Core RBAC must be implemented as a minimum in RBAC systems. The other components described below, which are independent of each other, can be implemented separately.

Core RBAC elements are defined as follows [ANS04, pp.4-5]:

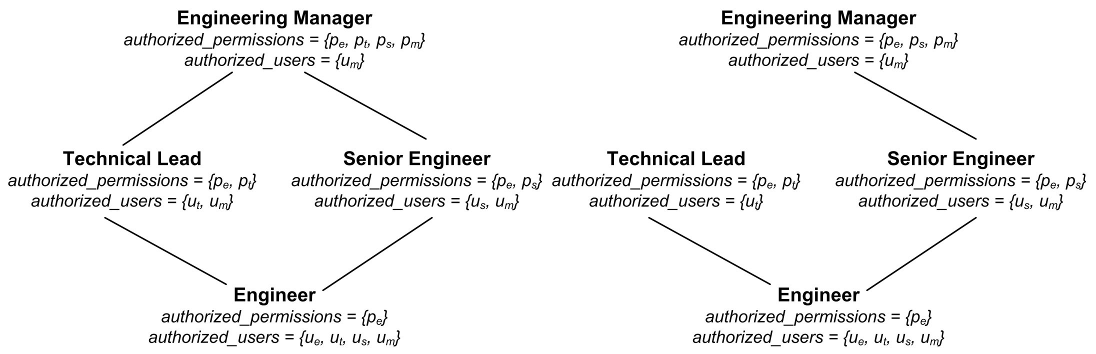
Definition 1 [**Core RBAC**]

- $USERS, ROLES, OPS$, *and* OBS *(users, roles, operations, and objects respectively)*
- $UA \subseteq USERS \times ROLES$, *a many-to-many mapping user-to-role assignment relation*
- $assigned\_users(r : ROLES) \rightarrow 2^{USERS}$, *the mapping of role* r *onto a set of users. Formally:* $assigned\_users(r) = \{u \in USRES | (u, r) \in UA\}$
- $PRMS = 2^{(OPS \times OBS)}$, *the set of permissions*
- $PA \subseteq PERMS \times ROLES$, *a many-to-many mapping permission-to-role assignment relation.*
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, *the mapping of role* r *onto a set of permissions. Formally:* $assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$
- $Op(p : PRMS) \rightarrow \{op \subseteq OPS\}$, *the permission to operation mapping, which gives the set of operations associated with permission* p
- $Ob(p : PRMS) \rightarrow \{ob \subseteq OBS\}$, *the permission to object mapping, which gives the set of objects associated with permission* p
- $SESSIONS$ = *the set of sessions*
- $session\_users(s : SESSIONS) \rightarrow USERS$, *the mapping of session* s *onto the corresponding user*
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, *the mapping of session* s *onto a set of roles. Formally:* $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$
- $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, *the permissions available to a user in a session* $= \bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$

Hierarchical RBAC

This component adds relations to support role hierarchies. Role hierarchy is a partial order relation that defines seniority between roles, whereby a senior role has at least the permissions of all of its junior roles, and a junior role is assigned at least all the users of its senior roles. A senior role is also said to "inherit" the permissions of its junior roles.

The standard defines two types of role hierarchies. These types are shown in Figure 1, and are defined as follows:


Engineering Manager
authorized_permissions = {p_e, p_t, p_s, p_m}
authorized_users = {u_m}
Technical Lead
authorized_permissions = {p_e, p_t}
authorized_users = {u_t, u_m}
Senior Engineer
authorized_permissions = {p_e, p_s}
authorized_users = {u_s, u_m}
Engineer
authorized_permissions = {p_e}
authorized_users = {u_e, u_t, u_s, u_m}
Engineering Manager
authorized_permissions = {p_e, p_s, p_m}
authorized_users = {u_m}
Technical Lead
authorized_permissions = {p_e, p_t}
authorized_users = {u_t}
Senior Engineer
authorized_permissions = {p_e, p_s}
authorized_users = {u_s, u_m}
Engineer
authorized_permissions = {p_e}
authorized_users = {u_e, u_t, u_s, u_m}

(a) General role hierarchy (b) Limited role hierarchy

Figure 1: Examples of Hierarchical RBAC

- General Role Hierarchies: provide support for arbitrary partial order relations to serve as the role hierarchy. This type allows for multiple inheritance of assigned permissions and users; that is, a role can have any number of ascendants, and any number of descendants
- Limited Role Hierarchies: provide more restricted partial order relations that allow a role to have any number of ascendants, but only limited to one descendant

In the presence of role hierarchy, the following is defined:

- $authorized\_users(r) = \{u \in USERS | r' \succeq r, (u, r') \in UA\}$ is the mapping of role r onto a set of users
- $authorized\_permissions(r) = \{p \in PRMS | r \succeq r', (p, r') \in PA\}$ is the mapping of role r onto a set of permissions

where $r_{senior} \succeq r_{junior}$ indicates that r_{senior} inherits all permissions of r_{junior}, and all users of r_{senior} are also users of r_{junior}.

Constrained RBAC

Static Separation of Duty (SSD) relations component defines exclusivity relations among roles with respect to user assignments. **Dynamic Separation of Duty (DSD) Relations** component defines exclusivity relations with respect to roles that are activated as part of a user's session.

2.1.2 Functional Specification

For the four components defined in the RBAC reference model, the RBAC System and Administrative Functional Specification defines the three categories of various operations that are required in an RBAC system. These categories are defined as follows.

The category of *administrative operations* defines operations required for the creation and maintenance of RBAC element sets and relations. Examples of these operations are listed here. A complete list of these operations, as well as their formal definition is included in the standard.

- Core RBAC administrative operations include AddUser, DeleteUser, AddRole, DeleteRole, AssignUser, GrantPermission, and so on
- Hierarchical RBAC administrative operations include AddInheritance, DeleteInheritance, AddAscendant, and AddDescendant
- SSD Relations administrative operations include CreateSsdSet, AddSsdRoleMember, SetSsdSetCardinality, and so forth

- DSD Relations administrative operations include CreateDsdSet, AddDsdRoleMember, SetDsdSetCardinality, and so on

The *administrative reviews* category defines operations required to perform administrative queries on the system. Examples of Core RBAC administrative review functions include RolePermissions, UserPermissions, SessionRoles, and RoleOperationsOnObjects. Other operations for other RBAC components can be found in the standard.

The *system level functionality* category defines operations for creating and managing user sessions and making access control decisions. Examples of such operations are CreateSession, DeleteSession, AddActiveRole, and CheckAccess.

2.2 Overview of CORBA Security

2.2.1 CORBA

This section provides a brief and informal overview of CORBA. More information can be found in the corresponding CORBA specifications. Readers familiar with CORBA are advised to proceed to Section 2.2.2.

CORBA specifications, including the CORBA Security Service [OMG02b], define a general-purpose interface definition language and OS-independent infrastructure for developing and deploying distributed applications. The distributed computing model that CORBA adheres to is outlined in the book *Object Management Architecture Guide* [SS96]. The model and all other CORBA specifications are developed by the Object Management Group (OMG), a consortium of software vendor and user organizations. Application systems and the CORBA infrastructure, including the Security Service, are defined using standard CORBA declarative facilities.

All entities in the CORBA computing model are specified by means of data structures and interfaces defined in the OMG Interface Definition Language (IDL) [OMG04]. The IDL resembles declarative elements of C++ in its syntax and constructs. A CORBA interface is a collection of three elements: operations, attributes, and exceptions. Interface definitions can inherit other interfaces to allow for interface evolution and composition. The fragment in Figure 2 illustrates a definition of an interface in IDL.

The module CompanyEmployee, shown in Figure 2, defines the following elements:

- simple data structure EmployeeName for representing an employee name
- new exception InvalidProject comprised of a textual description
- new type Experiences, which is an alias for a native type unsigned long
- new interface Employee with two read-only attributes, name and id, and the following operations:
  - assign_to_project and unassign_from_project perform steps necessary for assigning/unassigning an employee to/from a company's project. Each might throw an InvalidProject exception if the provided reference to the Project object (presumably defined in another IDL file project.idl) is invalid.
  - add_experience and get_experience add new experience or retrieve existing experience for an employee.
  - fire unassigns the employee from all projects, among other things.

The CORBA standards also define how IDL constructs are translated into various programming languages. The OMG standardized multiple language bindings, which means that CORBA objects—the implementations of the interfaces—can be coded in different programming languages and yet interoperate with clients and each other. Because of this feature, objects from different environments residing on different machines with different computing architectures and different operating systems can be integrated and shared among clients, making CORBA objects inherently distributable.

```
#include <experience.idl>
#include <project.idl>

module CompanyEmployee {
 struct EmployeeName {
    string       family_name;
    sequence<string>     middle_names;
    string       given_name;
 }

 typedef    unsigned long    EmployeeId;

 exception InvalidProject { string description; };

 interface Employee {
    readonly attribute  EmployeeName name;
    readonly attribute  EmployeeId id;

    void    assign_to_project( in Project prj )
                raises( InvalidProject );
    void    unassign_from_project( in Project prj )
                raises( InvalidProject );
    void    add_experience( in Experience new_exp );
    Experiences get_experience( );
    void    fire( );
 }
}
```

Figure 2: Defining a CORBA interface

When CORBA objects are deployed, they reside in OS processes and utilize CORBA middleware in the form of Object Request Broker (ORB) and object adapters to make their functionality available to the clients as well as to receive and process invocations and to return the results. Objects can act as clients as well, that is, make invocations on other objects, creating chains of invocations. Clients and targets may reside in the same or different processes or on different hosts. A CORBA ORB is responsible for core middleware functions, such as registering, keeping track of, and finding interface implementations, aiding clients in connecting to the objects, and providing communication transport from a client to a target.

CORBA ORBs communicate with each other, including sending object requests, by means of a special protocol for inter-ORB communications called Generic Inter-ORB Protocol (GIOP) [OMG04]. Because GIOP is a connection-oriented protocol and requires reliable service and presentation of communicated data as a byte stream, GIOP messages are delivered over the Transmission Control Protocol (TCP) in TCP/IP networks. Internet Inter-ORB Protocol (IIOP) [OMG04] is a specialization of GIOP for TCP. GIOP messages sent between the sender and receiver ORBs are translations of request/response interactions between the corresponding CORBA client and server object. From a security point of view, it is important to note the following about GIOP Request messages:

- To identify an object, the server uses an *object key* that is opaque to anybody except the hosting ORB. The client obtains the object key from the *object reference*. An operation on an object is identified by a string containing the IDL name of the operation, e.g., "assign_to_project".
- A list of service contexts accompanies all request and reply messages; it is a place for passing request-related data that different services, such as Transaction and Security, need to exchange.

Security service passes all of its data related to a particular request or reply in the form of service context elements in GIOP Request and Reply messages.

In order for a CORBA object to be accessible to its clients, it needs to have some equivalent of an address. An address of a CORBA object is presented in the form of an interoperable object reference (IOR). The ORB that hosts the object, working together with the object adapter, can create such references using the host IP address, the TCP port number, and other information essential for locating the object inside the ORB. Obviously, this information is specific to the TCP communication protocol, because the IP address and port number are part of the address. The information is also specific to the ORB that created the reference, because the object key is ORB specific.

Even though in theory, any CORBA client can invoke any CORBA object as long as the client has a valid IOR for that object, the overwhelming majority of practical scenarios involve clients and objects from same CORBA *deployments*, where all entities share the underlying security technology and often belong to same administrative domain. Such deployments are commonly limited by intranet boundaries or are subject to pre-established business relationships among organizations. An example of the latter kind is Parlay [ETS05], a standardized CORBA-based service for accessing functionality of a telecom network.

From a security perspective, the most interesting part of the IOR is the list of components, which allows additional information to be attached to the IOR so that it is available when the client establishes a connection with the server to make object invocations. We discuss several standard components specifically defined for supporting security in the following section.

2.2.2 Security Subsystem

CORBA Security service [OMG02b] (CORBASec for short) defines the content of security-specific GIOP service contexts, IOR components, and, most importantly, interfaces to a collection of objects for enforcing a range of security policies. It provides abstraction from an underlying security technology so that CORBA-based applications can be independent from the particular security infrastructure provided by the underlying computing environment.

CORBASec has an extensible model for *subject security attributes* to enable security run-time and administration scalability with possibly large numbers of subjects. Another example of grouping in CORBA security is *policy domains*, which allow scaling on the number of objects. Domains are used for most security policies in CORBA. A third grouping mechanism—also specific to access control—employs *required* and *effective* rights to allow scaling on the number of operations.

Another design goal of CORBASec architecture was to provide totally unobtrusive protection to applications. Most CORBA objects should be *security-unaware*, that is, run securely without any special programming of the application. At the same time, it should be possible for an object to exercise security policies that are application-specific and/or of finer granularity than those enforced by CORBASec run-time. Such objects are referred in CORBASec terminology as *security-aware*. For the purposes of this paper, we will focus on the means for protecting security-unaware objects.

The three main parts of CORBASec are client security service (CSS), target security service (TSS), and secure channel. The secure channel between CSS and TSS is established and managed via service context in the GIOP messages. As described in the previous section, any GIOP Request/Reply message contains a list of service context elements, which is used by different services for inserting service-specific information into the stream of communications between client and server. CORBASec defines a SecurityAttributeService (SAS) data type, which may be used in GIOP message service context to associate security-specific identity, authorization, and client authentication contexts with GIOP Request and Reply messages.

Functions performed by CSS and TSS overlap. Both sides include the following functionality:

- Creating and maintaining a secure channel with each other. While doing this, the CSS and TSS could authenticate each other if the applications' policies require them to do so

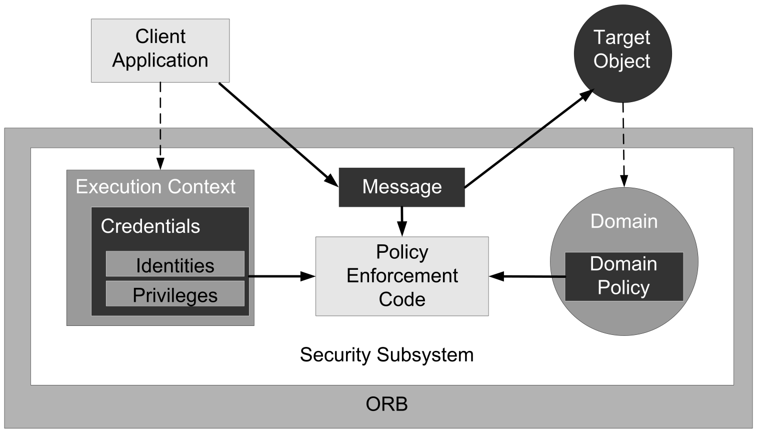


Figure 3: Enforcement of policies in CORBA security

- Protecting outgoing and verifying incoming messages according to the message protection policies (i.e., confidentiality and integrity)
- Performing a security audit of the invocations
- Implementing nonrepudiation policy, if any, on each side

There are, however, some differences between CSS and TSS. The CORBA CSS provides the following security functions, in addition to those listed above:

- Obtaining the principal's credentials by authenticating the user or retrieving credentials from the session environment if the principal has already been authenticated, and managing the principal's credentials created as a result of the authentication
- If necessary, translating the principal's credentials into those accepted by the TSS, as defined by the Authorization Token Layer Acquisition Service (ATLAS) specification [OMG02a], before they are "pushed" to the server.

The CORBA TSS provides the following security functions, in addition to the common ones:

- Authenticating clients and verifying their credentials if they are "pushed," or obtaining them if they are "pulled"
- Obtaining credentials used to authenticate the target to clients, usually by retrieving credentials from the session environment or from secure storage for principals not associated with people
- Performing an access control check on the requested object and method, based on the received credentials

Similar to other middleware security technologies, security policies in CORBA are enforced completely outside of an application system. Everything, including obtaining the information necessary for making policy decisions, is done before the method invocation is dispatched to the target object. As Figure 3 shows, the security enforcement code is executed inside a CORBA security service when a message from a client application to a target object is passed through the ORB. The CORBASec subsystem intercepts an invocation, determines what policy domain(s) a target or a client belongs to, and enforces the policies associated with the domain(s). In the rest of this section, we describe two key CORBASec functions—authentication and security administration. We describe access control in Section 4.1.

The concept of a user is absent from CORBASec. Instead, CORBASec uses the more generic and abstract notion of *principal*. "A principal is a human user or system entity that is registered in and authentic to the system" [OMG02b, p.2-3]. It is important for the analysis of RBAC support in CORBA to note that the notion of a session is indistinguishable from the notion of a

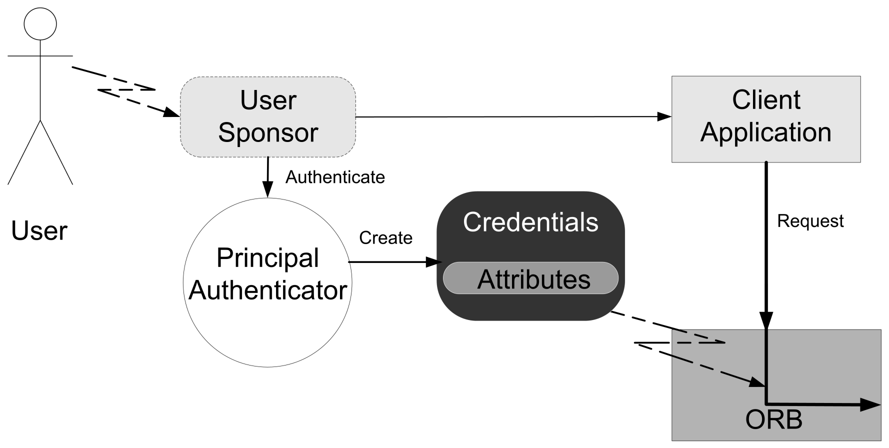


Figure 4: User authentication

principal's credentials. Thus the same principal might be represented by multiple, and possibly different, credentials. Just like a session in the ANSI RBAC model, once a Credentials object has been created, it begins to exist completely independently from other such objects, even those created to represent the same principal.

To create credentials, a CORBA application uses a UserSponsor to authenticate the principal to the CORBA Security environment (Figure 4). A UserSponsor is an implementation artifact that authenticates on behalf of a principal with and obtains authenticated credentials from a PrincipalAuthenticator. Instances of UserSponsor implement user interfaces specific to the authentication methods supported by the concrete implementations of CORBASec.

CORBASec does not mandate any particular authentication method; what it does specify, however, is the interface of a PrincipalAuthenticator. A PrincipalAuthenticator conducts the actual authentication and creates a Credentials object for a new principal. Based on the authentication data it received from the UserSponsor and on the underlying security technology (Kerberos [NT94], SESAME [PP95], or any other capable technology) as well as on the rules it adheres to, PrincipalAuthenticator instantiates the Credentials with various information. The Client's ORB associates Credentials object with requests on CORBA objects.

The authenticated security attributes of the principal are part of the information stored in the Credentials object. Hereafter, we understand attribute to mean security attribute. The TSS uses these attributes to decide which operations this principal can invoke on the target object ("target" for short). A variety of privilege attributes may be available, depending on the access policies. At any given time, the principal may be using only a subset of these permitted attributes, chosen either by the principal or by using a default set specified for the principal. There may be limits on the duration these privilege attributes are valid for and controls on where and when they can be used. These attributes, once established through principal authentication, are carried from CSS to TSS in the security-specific service context elements of GIOP messages.

CORBASec administration architecture rests on three constituents—*administrative interfaces*, defined on *policy* objects, each associated with a *policy domain*. CORBASec specifies administrative interfaces for managing most security runtime mechanisms described above, except authentication.[1] As with anything else in CORBA, these interfaces are defined in IDL. Since the mechanisms for user-account management are beyond CORBASec's scope, the interfaces for administering user-attribute assignment policies are as well. There are several types of policies; one of them is access control policy.

The policy enforcement code uses three sources of information: (1) the information from the

[1]For authentication, an administrator can still specify whether a target can be authenticated and/or requires its clients to authenticate.

client's credentials, (2) the message itself, which specifies the target object and the name of the method to be invoked, and (3) the policy of the domain to which the target belongs.

Any policy is associated with a policy domain—an abstraction that allows security administrators to group objects in groups and assign policies to the groups. Domains allow the application of access control and other policies to security-unaware objects without requiring changes to their implementations or interfaces. Policy domains are also the means by which CORBASec runtime and administration mechanisms achieve scalability on the number of objects in a system. Policies of more than one type (for example, authorization, audit, message protection) can be associated with the same policy domain.

The policy domain abstraction is represented in CORBASec by DomainManager objects. Whereas the management of domain membership is implementation dependent, an application can invoke the get_domain_managers operation on an object reference to obtain a list of the immediately enclosing domain managers for that object. The structure of the domain organization is determined by the relationships among DomainManagers. Even though an object can belong to more than one policy domain, CORBASec v1.8 specification states that it "does not require support for overlapping or hierarchical security policy domains" [OMG02b, p. F-6]. As a result, there is no standard semantics for making access control decisions for object belonging to several domains or for domain hierarchies.

Before describing access control architecture of CORBASec in detail, we review related work.

3 Related Work

Over the past decade, there has been no shortage of papers proposing ways to support RBAC. The overwhelming majority of this work, however, is about support for RBAC96 [SCFY96], which defines the reference models for plain, hierarchical, and constrained RBAC but does not specify the functions to be supported by an RBAC implementation. The paucity of analysis or proposals for supporting ANSI RBAC is not surprising, given the fact that the standard was published in 2004. Because of the lack of research on support for ANSI RBAC, and because of the significant similarities between RBAC96 and ANSI RBAC, we review related work on supporting or implementing RBAC96 in operating systems, databases, web applications, and distributed systems, including middleware.

Since the mainstream operating systems, with the exception of Solaris [Sun00], do not provide direct support for RBAC, researchers and developers have been employing either groups (e.g., [SGJ98, AS01]) or user accounts (e.g., [Fad99, Cha03]) to simulate roles. This choice determines whether more than one role can be activated in a session. Role hierarchies are either not supported [Fad99, Sun00] or are simulated by maintaining additional system files with the role hierarchy and various book-keeping data [SGJ98, AS01]. No implementations we reviewed support static SoD. Just one case of dynamic SoD comes as a side effect with those implementations that simulate roles with user accounts (i.e., [Fad99, Cha03]): the role set in this DSD is equal to the set of all roles in the system, and the cardinality of the role set is exactly one. In other words, any session can have only one role activated at any given time; the current role is deactivated while another role is activated.

We analyzed DB2 [TM06] and MySQL [MyS07] and updated the analysis of RBAC support in commercial database management systems (DBMS)—conducted by Ramaswamy and Sandhu [RS98]—with the latest versions of the corresponding systems. Commercial DBMS continue to have the most advanced support for RBAC96. Informix Dynamic Server v7.2 [IBM05], IBM DB2 [TM06], Sybase Adaptive Server v11.5 [Syb05], and Oracle Enterprise Server v8.0 [BLL03] directly support roles and role hierarchies. Only Oracle and Sybase allow users to have more than one role activated at any time. On the other hand, Informix also provides limited support for dynamic SoD, and Sybase features support for both types of SoD.

In RBAC implementations for client-server systems, including Web applications, roles are either "pushed" from the client to the server in the form of attribute certificates or HTTP cookies, as in [Gut01, PSA01], or "pulled" by the server from a local or remote database, as in [Bar97, FBK99, PSA01, CO02, ZM04]. The former enables selective activation of roles by users, and the latter simplifies the implementation of client authentication but activates all of the assigned roles for the user. However, Web implementation of NIST RBAC [FBK99] has hybrid design, which allows the user to select the roles to be "pulled" by the server. A number of implementations use a database, possibly accessible through the Light-weight Directory Access Protocol (LDAP) [WHK97] front-end, as in [Bar97, Gut01, PSA01, ZM04], to store role and other information. Role hierarchies are only supported by some implementations, using either manual assignment of permissions of junior roles to senior ones [PSA01], additional files [Giu99], a database [FBK99] or an LDAP server [CO02, ZM04]. JRBAC-WEB [Giu99] and RBAC/Web [FBK99] also support both types of SoD.

The work most relevant to ours addresses support for RBAC in middleware. Ahn [Ahn00] outlines a proposal for enforcing RBAC policies for distributed applications that utilize Microsoft's Distributed Component Object Model (DCOM) [Mic96, BK98, Mic98]. His proposal employs the following elements of Windows NT's architecture: (1) registry for storing and maintaining the role hierarchy, and permission-to-role assignment (PA), (2) user groups for simulating roles and maintaining user-to-role assignment (UA), and (3) a custom-built security provider that follows the RBAC model to make access control decisions, which are requested and enforced by the DCOM run-time. Since the support for role hierarchy is indicated but not explained in [Ahn00], we assume that the Windows NT registry can be used to encode the hierarchy so that the RBAC security provider can refer to it while making authorization decisions. Similar to the proposals for RBAC support in operating systems, the use of OS user groups for simulating roles enables activation of more than one role. Yet, like with the pull model in client-server systems, all assigned roles are activated, leaving no choice to the user. [Ahn00] does not indicate support for any kind of SoD, nor does he explain how RBAC policies can be enforced consistently and automatically in a multi-computer deployment of DCOM-accessible objects.

RBAC-JaCoWeb [WF99, OF02] utilizes the PoliCap [WdSFW$^+$02] policy server to implement CORBASec specification in a way that supports RBAC. PoliCap holds all data concerning security policies within a CORBASec policy domain, including users, roles, user-to-role and role-to-permission assignments, role hierarchy relations, and SoD constraints. Most of the authorization policy enforcement is performed by an RBAC-JaCoWeb CORBA security interceptor. At the time of the client binding to a CORBA object, the interceptor obtains necessary data from the PoliCap server and instantiates CORBASec-compliant DomainAccessPolicy and RequiredRights objects that contain the privilege and control attributes appropriate for the application object. When the client makes invocation requests later, the access decisions are then performed based on the local instances of these objects. Initially, the client security credentials object—created as part of the binding—has no privilege attributes, only AccessId, which is obtained from the client's X.509 certificate used in the underlying SSL connection. If the invocation cannot be authorized with the current set of client privilege attributes, the interceptor "pulls" additional user's role attributes from the PoliCap server. Only those roles that are (1) assigned to the user, (2) necessary for the invocation in question to be authorized, and (3) not in conflict with any DSoD constraints are activated. These role attributes are added to the client's credentials and are later re-used on the server for other requests from the same principal. The extent to which RBAC-JaCoWeb conforms to the CORBASec specification is unclear from [WF99, OF02]. Nevertheless, RBAC-JaCoWeb serves as an example of implementation-specific extensions to CORBAsec that enable better support for RBAC advanced features, such as role hierarchies and SoD, which—as will be seen from the results of our analysis—cannot be supported without extending a CORBASec implementation with additional operations.

This work builds on the methodology of Beznosov and Deng [BD99]—who analyze support for RBAC96 in CORBASec—and applies it to the analysis of CORBASec support for ANSI RBAC, which defines not only reference models but also functional specifications for each of the models. Furthermore, we extend and correct some of the results of [BD99]. We extend their definition of the CORBA protection state with the operational definition of the function access_allowed, demonstrating that the definition of the protection state is sufficient for computing access control decisions in CORBA. Beznosov and Deng also analyze support for RBAC$_{0-3}$ models in CORBA and suggest how these models could be implemented. Similarly to our work, their results indicate that—aside from conforming to the CORBASec specification—additional functionality needs to be implemented in order to support RBAC96 models in CORBA.

After analyzing the functional specification of ANSI RBAC, we identify three major shortcomings of CORBASec, specifically (1) the lack of a standard mechanism for enumerating all objects that implement the DomainAccessPolicy interface in a CORBA deployment, (2) the lack of the notion of user accounts and support for their management, as well as the lack of explicit user representation, and (3) the inability to enumerate all CORBA principals related to a specific user. Based on these findings, we conclude, unlike [BD99], that CORBASec is largely inadequate for implementing ANSI RBAC functions without resorting to vendor-specific extensions of a CORBAsec implementation.

4 CORBA Protection State

One of the two major contributions of this paper is a formalization of the CORBA protection state, which is defined in Section 4.2. We explain first the architecture of the access control mechanisms in CORBA.

4.1 CORBA Access Control Architecture

Due to its general nature, CORBASec is not tailored to any particular access control model. Instead, it defines a general mechanism that is supposed to be adequate for the majority of cases and can be configured to support various access control models. For example, implementing lattice-based mandatory access control (MAC) using CORBASec is shown in [Kar00].

Access control policies in CORBASec are expressed through *security attributes* of principals, attributes of objects, and operations implemented by those objects. Because CORBASec defines an extensible attribute model, it enables access control policies based on roles, groups, clearance, and any other security-related attributes of the principal. From the access control model point of view, a Credentials object is nothing but a set of authenticated attributes. An attribute is a four-tuple ($a = \{\tau, \alpha, \upsilon, \delta\}$) with certain type τ, defining authority α, value υ, and delegation state δ, where $\delta \in DS = \{i, d\}$. State i indicates an attribute possessed by the immediate invoker, and d – by the intermediate one (i.e., delegated). Attribute types are partitioned into two families: privilege attributes and identity attributes. The family of privilege attributes enumerates attribute types that identify principal privileges: access id, primary and secondary groups the principal is a member of, clearance, capabilities, etc. Identity attributes, if present, provide additional information about the principal: audit id, accounting id, and non-repudiation id, reflecting the fact that a principal might have various identities used for different purposes. Principal credentials may contain zero or more attributes of the same family or type.[2] An example of security attributes assigned to authenticated principals is provided in Table 1. The role attribute is one of the standard CORBA attribute types. Due to the extensibility of the schema for defining security attributes, an implementation of CORBASec can support attribute types that are not defined by the CORBASec standard. Although the normative part of CORBASec does not mandate the way attributes are managed, assignment of such attributes to users is meant to be performed by user administrators.

In the CORBA computational model, all a principal does is to invoke operations on corresponding objects. In order to make a request, one needs to know two things: object reference, which uniquely identifies an object, and operation name. An operation name is unique for an interface.[3] Thus, any operation is uniquely identified by its name and by the name of the

[2]This rule applies to all attribute types including accessid, although it is hard to foresee a useful implementation of CORBASec where a principal would have multiple access identities.

[3]Interface inheritance in CORBA does not allow inheritance from interfaces with operations of the same name. This rule resolves the problem of operation name overloading.

Principal	Attributes
p_1	a_1
p_2	a_2, a_6
p_3	a_2, a_3
p_4	a_4, a_5

Table 1: An example of security attributes possessed by authenticated principals.

Operations	**Required Rights**	**Combinator**	**Meaning**
$i_1.m_1$	r_1	all	Only a principal who is granted right r_1 can invoke the operation.
$i_1.m_2$	r_1, r_2	any	Any principal who is granted either r_1 or r_2 right can invoke the operation.
$i_2.m_1$	r_2, r_3	all	Only a principal who is granted both r_2 and r_3 rights can invoke the operation.
$i_2.m_2$	r_2, r_3, r_4	all	Only a principal who is granted all r_2, r_3, r_4 rights can invoke the operation.
$i_3.m_1$	r_1, r_2, r_3, r_4	all	Only a principal who is granted r_1, r_2, r_3, and r_4 rights can invoke the operation.

Table 2: Required rights matrix.

interface it is defined in. In this paper, we use the notation $i.m$, to refer to operation (a.k.a. method) m on interface i. There is a global[4] set of *required rights* for each operation defined by its interface's required rights mapping.[5] The required rights set, together with a combinator (*all* or *any* rights), defines what rights a principal has to have in order to invoke the operation. Table 2 provides an example of required rights for operations on three interfaces, i_1, i_2, i_3. It is assumed that required rights are defined and that their semantics are precisely documented by application developers who best know what each operation does. CORBASec Level 2 API defines the operation set_required_rights(operation, interface, rights, rights combinator) for managing required rights.

Figure 5 is useful for illustrating our discussion. Depending on the access policy (DomainAccessPolicy) enforced in a particular access control policy domain, a principal is granted different rights (GrantedRights) according to what SecurityAttributes it has.[6] Each DomainAccessPolicy defines what rights are granted for each security attribute. An example of a mapping between principal privilege attributes and granted rights is provided in Table 3. Security administrators are responsible for defining what rights are granted to what security attributes in what delegation state on a domain by domain basis. CORBASec Administrative API defines operations grant_rights(attribute, rights) (as well as revoke_rights, and replace_rights) for managing rights granted for an attribute in the scope of a particular policy domain.

[4]"Global" in the context of required rights means that they are independent of the policy domain in which the object is located.

[5]One caveat with respect to required rights relates to interface inheritance. Nothing prevents, say, operation m on interface i having one set of required rights and another set on interface j, even if i is a subtype of j. Therefore, determining the most derivative interface of a CORBA object is crucial for computing required rights at the authorization stage.

[6]For the sake of brevity, we omit the delegation state qualifier for granted rights. This omission does not change the correctness of the discussion, as we show below.

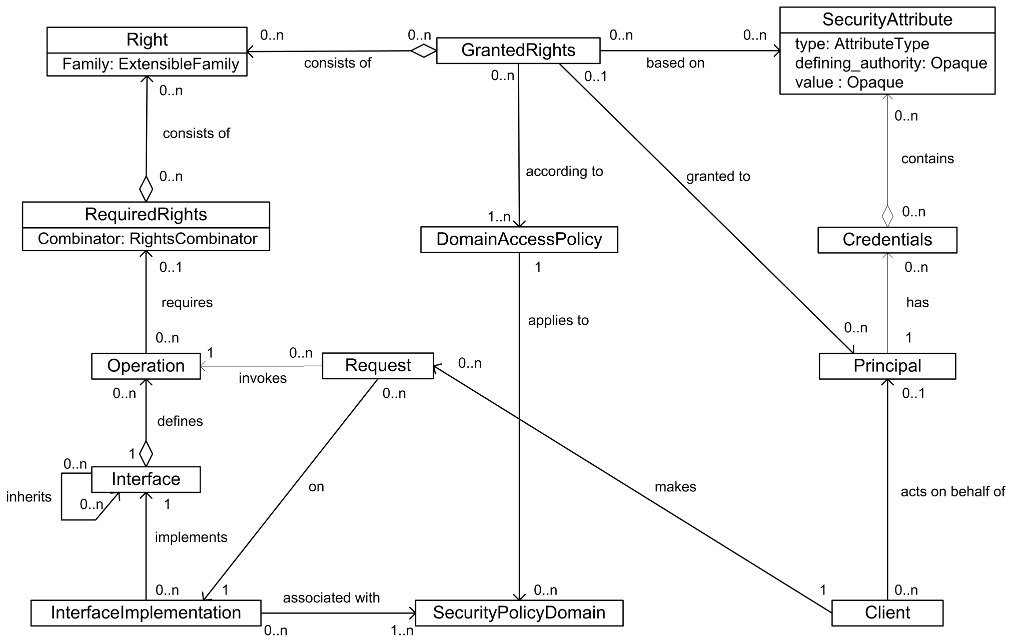


Figure 5: A model of CORBASec access control architecture in UML notation.

Attri-butes	**Granted Rights**	
	Domains	
	d_1	d_2
a_1	r_1	r_2
a_2	$-$	r_1
a_3	r_2, r_3	$-$
a_4	r_3	r_1, r_4
a_5	r_1, r_2, r_3	r_2, r_3, r_4
a_6	r_6	r_1

Table 3: Granted rights per attribute

Principal	**Granted Rights**	
	Domains	
	d_1	d_2
p_1	r_1	r_2
p_2	r_6	r_1
p_3	r_2, r_3	r_1
p_4	r_1, r_2, r_3	r_1, r_2, r_3, r_4

Table 4: Effective rights of principals in each of the two domains.

Principals	**Permitted Operations**	
	Domains	
	d_1	d_2
p_1	$i_1.m_1, i_1.m_2$	$i_1.m_2$
p_2	–	$i_1.m_1, i_1.m_2$
p_3	$i_1.m_2, i_2.m_1$	$i_1.m_1, i_1.m_2$
p_4	$i_1.m_1, i_1.m_2, i_2.m_1$	$i_1.m_1, i_1.m_2, i_2.m_1, i_2.m_2, i_3.m_1$

Table 5: Operations that principals from the example can invoke.

Whenever a principal attempts to invoke an operation, its effective rights are computed via operation AccessPolicy::get_all_effective_rights(...). CORBASec purposely does not define how the operation combines rights granted through the different privilege attribute entries in Table 3. The specifiers let CORBASec implementors define the operation's semantics ([OMG02b, p. 2-123]). The simplest implementation of get_all_effective_rights would be when the set of rights granted to a principal is a union of rights granted to every security attribute possessed by the principal. For the rest of this paper, we will assume these semantics for the operation. If we use our example of security attributes assigned to principals p_1, p_2, p_3, and p_4 (Table 1), and granted rights (Table 3), then Table 4 shows what "effective" rights the principals have in each domain.

The use of effective rights and policy domains makes the correspondence between the ANSI RBAC OBS set and CORBA objects nontrivial. Note that the effective rights of the invoking principal are computed for the object's policy domain. At the same time, all instances of the same interface implementation that belong to the same domain are indistinguishable for the purpose of making access control (and other policy) decisions. That is, a principal has exactly the same permissions on all objects that implement the same interface(s) and belong to the same domain. To accommodate this important detail, we defined the ANSI RBAC OBS set as a cross-product between CORBA interfaces and access policy domains: $I \times D$.

Once the principal's effective rights are determined, they are compared to the rights required for the operation. If the match is successful, the request is authorized. Given the required rights in Table 2 and the rights granted to the principals in Table 4, Table 5 shows what operations can be invoked by the principals from our example.

4.2 Formalization of the Protection State

In this section, we formalize the semantics of the CORBA access control architecture.

Definition 2 [CORBA privilege attributes] *CORBA privilege attributes are* $A \subseteq T \times AUTH \times V \times DS$*, where* $T, AUTH, V, DS$ *are interpreted as follows:*

- $T = \{\_Public, AccessId, PrimaryGroupId, GroupId, Role, AttributeSet, Clearance, Capability\}$ *is the set of types.*
- $AUTH$ *is the set of authorities.*
- V *is the set of values.*
- $DS = \{i, d\}$ *is the set of delegation states.*

Definition 3 [CORBA Protection State] *A configuration of a CORBA system protection state is a tuple (I, OPS, IOPS, RIGHTS, RR, D, DOBS, A, GR,* get_all_effective_rights*) interpreted as follows:*

- I *is the set of interfaces.*
- OPS *is the set of operations on CORBA objects.*

- $IOPS \subseteq I \times OPS$ *specifies which operations are defined on which interfaces.*
- $RIGHTS$ *is the set of rights.*
- $RR \subseteq IOPS \times 2^{RIGHTS}$ *defines rights required for invoking operations on interfaces.*
- D *is the set of security policy domains.*
- $Inst$ *is the set of CORBA objects.*
- $DOBS \subseteq Inst \times D$ *associates each object with a policy domain.*
- A *is the set of privilege attributes as specified in Defintion 2.*
- $GR \subseteq A \times (D \times RIGHTS)$ *associates an attribute with a domain and a right;* $(a, d, r) \in GR$ *means that attribute* a *is granted right* r *in domain* d.
- get_all_effective_rights*:* $D \times 2^A \rightarrow 2^{RIGHTS}$*, a function computing rights that are in effect for a given set of privilege attributes in a given domain. Although this function uses* GR *to obtain rights granted for each attribute, the semantics of combining the granted rights are implementation-specific.*

An implementation of security service compliant with CORBASec is supposed to yield the same access control decision as that described by Algorithms 1 and 2. Employed by Algorithms 1 and 2, functions get_domain_policy and get_all_effective_rights are defined by CORBASec.

```
access_allowed($u : 2^A, m : OPS, o : Inst, i : I) \rightarrow \{true, false\}$
Require: $(i, m) \in IOPS$
1: for all $(o, d) \in DOBS$ do
2:   {Find an access policy domain.}
3:   $p \Leftarrow get\_domain\_policy(d, AccessPolicy)$
4:   if $p \neq NULL$ then
5:     return $is\_authorized(u, i, m, d)$
6:   end if
7: end for
```

Algorithm 1: Operational definition of function access_allowed. This function makes the access control decision with regard to principal u accessing operation m on instance o of interface i.

CORBASec standard is unclear about cases when an object belongs to more than one domain that has *AccessPolicy*. To resolve the ambiguity, we chose Algorithm 1 to use first domain of the object that has *AccessPolicy*. Because a policy domain might not have *AccessPolicy*, the algorithm iterates until it finds a domain that does.

```
is_authorized($u : 2^A, i : I, m : OPS, d : D) \rightarrow \{true, false\}$
1: $er \Leftarrow get\_all\_effective\_rights(d, u)$
2: if $\exists\ (i, m, rr) \in RR : rr \subseteq er$ then
3:   return true
4: else
5:   return false
6: end if
```

Algorithm 2: Operational definition of function is_authorized.

We separated authorization logic into two functions. This separation is purely syntactical and its only purpose is to demonstrate in Section 5.4 to the reader the capability of the CORBASec to provide an implementation of ANSI RBAC's CheckAccess in the form of is_authorized. This

Subjects	**Interfaces**		
	i_1	i_2	i_3
p_1	$i_1.m_1$		
p_2	$i_1.m_1, i_1 m_2$		
p_3	$i_1.m_1, i_1.m_2$		
p_4	$i_1.m_1, i_1.m_2$	$i_2.m_1, i_2.m_2$	$i_3.m_1$

Table 6: Access matrix for domain d_2

function is the same as access_allowed, except that it makes an authorization decision for a given domain d and particular operation m on CORBA interface i to be accessed by principal u. In Algorithm 2, the operation get_all_effective_rights retrieves granted rights and combines them according to its implementation semantics. Effective rights of the principal in the object's domain are checked then against RR. If the match succeeds, then access is granted. Otherwise, access is denied. An example of an algorithm for get_all_effective_rights that returns a union of the rights granted per each attribute are shown in Algorithm 3.

```
get_all_effective_rights($d : D, u : 2^A$) → $2^{RIGHTS}$
1: $er \leftarrow \emptyset$
2: for all $a \in u$ do
3:   for all $(a, d, r) \in GR$ do
4:     $er \Leftarrow er \cup r$
5:   end for
6: end for
7: return $er$
```

Algorithm 3: Operational definition of a sample function get_all_effective_rights that returns a union of all rights granted to principal u in domain d.

We simplified the semantics of the support for the required rights combinator in the definition of RR and Algorithm 2. Combinator value "any" is supported via separate elements of RR. For example, both rights r_1 and r_2 are required for operation $i.m$, then $(i.m, \{r_1\}) \in RR$ and $(i.m, \{r_2\}) \in RR$. Whereas, combinator value "all" is supported by listing all the required rights in one element of RR, e.g., $(i.m, \{r_1, r_2\})$.

For each domain, a Lampson's access matrix [Lam71], such as that one in Table 6, can be constructed. Three general observations are worth noting regarding an access matrix constructed for any CORBASec system. First, subjects cannot be objects, i.e., the CORBA access control model does not support the concept of operations on principals. It only has the concept of operations on interfaces, which are objects according to the terminology of the access matrix [Lam71]. Second, since $i_k.m_p \equiv i_l.m_q \iff k \equiv l \wedge p \equiv q$ (i.e., just $p \equiv q$ is not enough for $i_k.m_p \equiv i_l.m_q$), the semantics of the operations with same names but defined on different interfaces in a general case might be different. Thus, for each subject s and object o, the content of cell $[s, o]$ is specific to the object. That is, no operations permitted on one object can be permitted on another, because operations are semantically different for every interface unless the interfaces are related through inheritance. Third, since those implementations of the same interface that are located in the same access policy domain are indistinguishable from the access control point of view, all such interface implementations are represented by the same object in the access matrix. This is one of the reasons policy domains are important in the CORBA access control model.

Before we proceed to our analysis of the support for ANSI RBAC in CORBA, we would like to note that not all sets from Definition 3 can be enumerated. Particularly, we could not find operations in CORBA specifications that allow enumerating $RIGHTS, Inst, A, D, OPS, I, IOPS$ sets. As a consequence, a number of ANSI RBAC functions cannot be supported without resorting to implementation specifics. However, membership in the last three sets can be tested through the operation get_required_rights specified on the interface RequiredRights.

The lack of standard mechanisms for enumerating all objects ($Inst$) in a given CORBA deployment accounted for the inability of CORBASec to support $RolePermissions$ and, consequently, $SessionPermissions$ functions of the ANSI RBAC functional specification. The implementation of these two functions requires enumeration of effective rights for a given role or principal. In order to do so, it is necessary to obtain a reference to every object that implements interface DomainAccessPolicy and to invoke the operation get_all_effective_rights on it.

5 Support for ANSI RBAC in CORBA

Recall that among the four sets of ANSI RBAC features (also referred in the standard as *model components*), Core RBAC is the required minimum for any implementation compliant with the standard. A system supporting Core RBAC must implement functions for administering user accounts, roles, sessions, objects, operations, and permissions. Hierarchical RBAC has hierarchies of roles in addition to everything Core RBAC has. The last two standard's components, Static Separation of Duty (SSD) Relations and Dynamic Separation of Duty (DDS) Relations, define relations among roles with respect to user assignments as well as role activation in user sessions.

We first examine in Section 5.1 the extent to which a CORBA protection state—as formalized in Definition 3—can support each of the four ANSI RBAC model components. Second, we describe in Section 5.2 steps for translating an ANSI RBAC policy into a CORBA protection state. Then, we illustrate the results of our analysis with two examples in Section 5.3. Finally, we analyze in Section 5.4 the degree to which the programming interfaces defined in CORBASec and other related parts of the CORBA specification support the functional specification of ANSI RBAC. We discuss results of our analysis in Section 6.

5.1 Reference Model

5.1.1 Core RBAC

The five sets of Core RBAC identities are represented in CORBA Security as follows: *Users* in RBAC map to user accounts in CORBASec; *Roles* are represented by a set of privilege attributes of type *role*; each RBAC object is a collection of CORBA objects that implement the same interface(s) and belong to the same access policy domain(s), and are thus indistinguishable from the point of view of a CORBA protection system; *Permissions* are operation-object pairs; RBAC *Sessions* are equivalent to CORBA principals, which can be reduced for the purpose of this paper to just sets of security attributes. We do not mention CORBASec access control domains because, as will be shown in the examples in Section 5.3, ANSI RBAC models can be supported in CORBA using either a single domain or multiple domains.

The Core RBAC (see Definition 1) in the language of CORBA Security is formally defined as follows:

Definition 4 [Core RBAC in CORBASec] *Core RBAC in the language of CORBA Security is defined by the CORBA system protection state outlined in Definition 3, as well as the following additional elements:*

- $USERS$ *is the set of user accounts.*
- $ROLES \subseteq A$ *roles, which are CORBA privilege attributes of type* role.
  Formally: $ROLES = \{a | a \in A \wedge T(a) \equiv Role\}$.
- $OBS \subseteq I \times D$ *set of objects distinguishable from the point of view of access control in CORBA. That is, for any two elements of OBS, there could be a CORBA protection*

system state in which the same principal p have different access rights on these elements, even if they both implement the same interface(s). An ANSI RBAC object is mapped into a tuple (i, d), i.e., a CORBA interface and the access policy domain it is a member of.

- $UA \subseteq USERS \times ROLES$, *a many-to-many user-account-to-role assignment relation.*
- $assigned\_users(r : ROLES) \rightarrow 2^{USERS}$, *the mapping of role r onto a set of user accounts, as in ANSI RBAC (see Definition 1).*
- $PRMS \subseteq OPS \times OBS$ *the set of permissions. A permission can be considered as a three-tuple* (op, i, d), *i.e., operation, interface, and domain.*
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, *the mapping of role r onto a set of permissions. Function* assigned_permissions *is specified operationally by Algorithm 4.*
- $Op(p : PRMS) \rightarrow \{op \in OPS\}$, *the permission to operation mapping, which gives operation op associated with permission p.*
- $Ob(p : PRMS) \rightarrow \{ob \in OBS\}$, *the permission to RBAC object mapping, which gives object ob associated with permission p.*
- $domain(p : PRMS) \rightarrow D$, *the permission to CORBA access policy domain mapping, which gives the domain associated with the permission. The mapping is used by Algorithm 4.*
- $interface(p : PRMS) \rightarrow I$, *the permission to CORBA interface mapping, which gives the interface associated with the permission.*
- $PA \subseteq PRMS \times ROLES$, *a many-to-many permission-to-role assignment relation, defined through the function assigned_permissions.*
- $SESSIONS \subseteq 2^A$. *RBAC sessions are represented by CORBA principals, which in their turn can be treated for the purpose of access control as sets of security attributes from A.*
- $session\_users(s : SESSIONS) \rightarrow USERS$, *the mapping of a session onto the corresponding user account.*
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, *the mapping of a session onto a set of roles. Formally:* $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$.
- $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, *the permissions available to a session* $= \bigcup_{r \in session\_roles(s)} r \in assigned\_permissions(r)$.

```
assigned_permissions$(r : ROLES) \rightarrow \{p \in 2^{PRMS}\}$
 1: $AP \leftarrow \emptyset$ {Initialize the set of assigned permissions to return}
 2: for all $p \in PRMS$ do
 3:   $i \leftarrow interface(p)$
 4:   $m \leftarrow Op(p)$
 5:   $d \Leftarrow domain(p)$
 6:   if is_authorized$(\{r\}, i, m, d)$ then
 7:     $AP \leftarrow AP \cup p$
 8:   end if
 9: end for
10: return AP
```

Algorithm 4: Operational definition of function *assigned_permissions*, which determines permissions assigned to a given role in a CORBA system.

Definition 4 specifies all elements of Core RBAC. The elements $PRMS$, Op, and Ob require further elaboration. The definition of $PRMS$ in ANSI RBAC (Definition 1) allows each permission to comprise multiple operation-object pairs. A CORBA permission, on the other hand, consists of only one such pair, which can be considered as a more restricted case of ANSI RBAC $PRMS$, i.e., $OPS \times OBS \subset 2^{OPS \times OBS}$. The ranges of functions *Op* and *Ob* are elements, not subsets, of OPS and OBS, respectively. Given that an element of a set also comprises a subset of the set, ANSI RBAC versions of *Op* and *Ob* functions can be substituted by their counterparts from Definition 4. Thus, $PRMS$, *Op*, and *Ob* from Definition 4 can be used instead of the corresponding elements in Definition 1.

In the rest of this section, we explain how elements of Definition 4 are or can be supported by CORBASec. Because the notion of user accounts is missing from CORBASec, the set $USERS$, relation UA, and functions $assigned\_users$ and $session\_users$ have to be implementation-dependant. The enumeration of elements from $SESSIONS$, which we have defined as a set of CORBA principals (Definition 4), was found to be not supported by CORBA specifications either. However, CORBASec's SecurityLevel1.Current interface does define the operation get_attributes, which returns a list of security attributes of the principal responsible for the current invocation. For the purposes of our analysis, the returned list of security attributes was sufficient to represent the current principal and therefore the current session.

An implementation of function *session_roles* is straightforward because an ANSI RBAC session is a principal in CORBA, and a principal is a set of security attributes, each of a particular type. Thus, all *session_roles* needs to do is to return those principal's attributes whose type is *role*.

As we explained at the end of Section 4.2, the CORBA specification does not define operations sufficient for enumerating all CORBA objects in a given CORBA deployment. An affirmative answer is necessary in order to enumerate the elements of the $PRMS$ set, on which our operational definition of the $assigned\_permissions$ function depends (specifically, line 3). Thus, the functionality necessary for enumerating the $PRMS$ set would have to be implementation-dependent.

The interfaces and data structures defined by CORBASec enable, however, the construction of individual elements of set $PRMS$. To demonstrate, consider data used for making access control decisions in CORBA. For any given request on a CORBA object, the CORBA security subsystem intercepts the request and invokes access_allowed (Algorithm 1) with the following parameters: subject's credentials, object's reference, the operation to be invoked, and the name of the interface on which the operation is defined. In its turn, access_allowed obtains the access policy domain that object o is a member of, before invoking *is_authorized* (defined by Algorithm 2). Thus, at the time when *is_authorized* is invoked, all data necessary for constructing a corresponding permission, as specified in Definition 4, are available to the CORBASec subsystem.

Due to the structure of permission, (op, i, d), valid implementations of the functions *Op*, *Ob*, *domain*, as well as $interface$, could just return the corresponding parts of the permission argument. For example, *Ob* needs to return the (i, d) tuple.

The function *avail_session_perms* is operationally defined by Algorithm 5. Also, see the caveat about the related function *SessionRoles* in Section 5.4.

```
avail_session_perms($s : SESSIONS$) → $\{p \in 2^{PRMS}\}$
1: $AP \leftarrow \emptyset$ {Initialize the set of available permissions to return}
2: for all $r \in s$ do
3:     $AP \leftarrow AP \cup assigned\_permissions(r)$
4: end for
5: return AP
```

Algorithm 5: Operational definition of the function $avail\_session\_perms$, which determines permissions available to a given session.

As can be seen from the above analysis of Definition 4, most elements of ANSI Core RBAC can be provided by any implementation compliant with CORBA Security Main Functionality

Level 2. However, support for user-specific elements of the Core RBAC, as well as for enumerating such sets as $SESSIONS$ and $PRMS$, must be implementation-specific.

5.1.2 Hierarchical RBAC

In order to implement ANSI RBAC role hierarchies, a system—in addition to Core RBAC—has to provide support for modifying and reviewing a partial-order relation on roles, RH, and, more importantly, the functions $authorized\_users$ and $authorized\_permissions$ that are defined on RH. Specifications of all three are reproduced in Section 2.1.1. CORBASec does not provide direct support for RH and the two functions. A CORBASec implementation, however, can emulate the support for role hierarchies—either *general* or *limited*—in three different ways.

First, PrincipalAuthenticator can be implemented to activate not only those roles that can be activated through direct user-to-role assignment but also the roles junior to those activated. For example if $r' \succeq r$, and r' has been activated, then r is also activated. Proposals by [SGJ98, AS01] follow a similar path. In this case, the RH logic can be encapsulated into PrincipalAuthenticator, whereas the target security service (TSS) and other CORBASec components provide no special support for role hierarchies. A valid implementation of Hierarchical RBAC using PrincipalAuthenticator could be one (a) that allows a user to specify any role junior to those the user is a member of; and (b) in which PrincipalAuthenticator activates the specified role(s) as well as all roles junior to the specified one(s).

The second choice is to shift support for role hierarchies to the TSS. Specifically, get_all_effective_rights would be required to return not only effective rights for the activated roles, but also for all roles junior to the activated ones, as in [Giu99]. Using the above example, a call to the modified version of get_all_effective_rights($d, \{r'\}$), would be equivalent to get_all_effective_rights($d, \{r', r\}$). This option requires maintenance of RH and run-time access to it by the TSS. Since in CORBASec, the credentials of the principal are always "pushed" from the client to the server, we found no opportunity to support Hierarchical RBAC by adding role attributes to the client's credentials by TSS, as proposals [Bar97, FBK99, Ahn00, PSA01, CO02, ZM04] do.

The third option is to modify the administrative tools—similarly to [PSA01]—to ensure that the CORBASec rights that are granted to every role include the rights this role inherits from the junior roles. No special run-time support for role hierarchies would then be needed. However, this option requires not only maintaining RH but also keeping track of the reason(s) a right was assigned to a role, i.e., because of direct assignment or through inheritance from a particular role. Such assignment details would be necessary in order to perform right revocation and RH administration properly.

No matter which of the three options is selected, support for RH, $authorized\_users$, and $authorized\_permissions$ would be implementation-specific.

5.1.3 Constrained RBAC

The Constrained RBAC component of ANSI RBAC [ANS04] introduces static and dynamic separation of duty relations to the RBAC reference model. In essence, SSD constrains user-to-role assignment (UA set and $assigned\_users$ function) and the role hierarchy (RH set and $authorized\_users$ function). DSD, on the other hand, constrains the role activation ($SESSIONS$ set and $session\_roles$ function). Since user accounts, role hierarchies, and role activation are beyond the scope of CORBASec, the Constrained RBAC component, if supported, would have to be implementation-dependant.

5.2 Translating RBAC Policies to CORBA

An interesting and practical question is the translation of an arbitrary ANSI RBAC policy into a CORBA protection state. The key elements of an RBAC policy are the user-to-role and permission-to-role assignment (PA) relations. Given that the management of user accounts and their security attributes is beyond the scope of CORBA standards, the question boils down to the PA relation. In this section, we describe a simple sequence of steps that allows a given PA defined in ANSI RBAC terms to translate into the required and granted rights assignments and the assignment of CORBA objects to policy domains.

1. Split each of the compound RBAC permissions into "atomic" ones so that each permission comprises only one object-operation pair.
2. Associate every RBAC "atomic" permission with a CORBA object and an operation on that object. Unless otherwise stated, we refer in the following steps to CORBA objects and operations on them.
3. For every object, create a separate access policy domain and assign that object to its domain.
4. For every interface and its operations that are implemented by the objects, create rights, one per interface-operation pair. Make each right a required right for the pair's operation.
5. For every RBAC permission-to-role assignment relationship from PA, grant the corresponding right to the role in the policy domain of the permission's object.

The main advantage of the above approach is the straightforwardness of the initial translation and the simplicity of future incremental modifications to the policy. Granting/revoking a permission to/from a role requires adding/removing an association between a right and a role in the object's domain in the GR relation. Adding/removing an object results in creation/deletion of a policy domain (and possibly several rights, if the object implements a unique interface). The main disadvantage of this approach is the proliferation of rights and domains. The above steps result in the creation of as many rights as the number of unique interface-operation pairs and as many access policy domains as interface instances. Optimizations of this approach to policy translation, although conceivable, are not discussed further due to space limitations. Another complementary question—which could be a subject of future research—is how to determine if a given CORBA protection state enforces a given ANSI RBAC policy.

5.3 Examples

To illustrate the results of our analysis of the ANSI RBAC reference model support in CORBA systems, we show how a CORBA-based distributed system could be configured to protect access to the implementations of CORBA interfaces shown in Figure 7 according to the policies listed in Figure 8 and the general role hierarchy example shown in Figure 6. For the purposes of the example, we suppose that the get_all_effective_rights function returns a union of granted rights per attribute. We first show a single-domain solution, and then provide a solution that employs several domains to illustrate the benefit of CORBASec policy domains.

5.3.1 Single Access-Policy Domain Solution

In order to implement the role hierarchy in CORBASec without using access policy domains, we introduce two new interfaces, EngineeringProject1 and EngineeringProject2, shown in Figure 9.

The following configuration of a system protection state could be used:

- $I = \{Employee, EngineeringProject1, EngineeringProject2\}$.
- $OPS = \{get\_name, assign\_to\_project, unassign\_from\_project, add\_experience,$ $get\_experience, fire, inspect\_quality, make\_changes, report\_problem,$ $review\_changes, close, close\_problem, create\_new\_release, get\_description\}$.
- $IOPS$= $\{$

  We do not use any implementations of the interface EngineeringProject. Only derived interfaces are used.

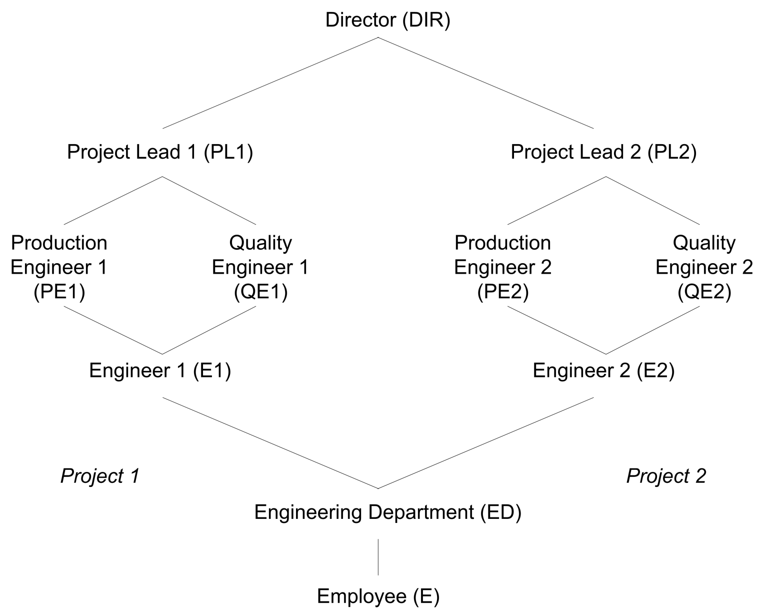
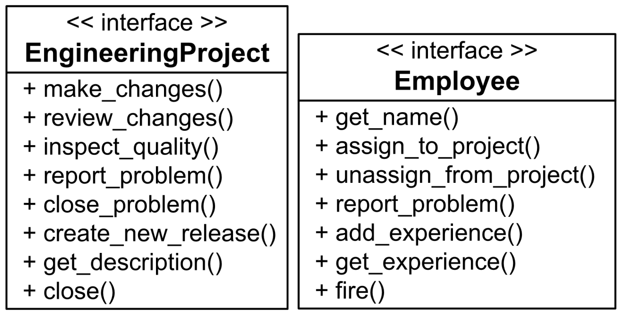


Figure 6: An example role hierarchy (from [SP98])



(a) EngineeringProject (b) Employee

Figure 7: Interfaces protected in the example

1. Anyone in the organization can look up an employee's name.
2. Everyone in the engineering department can get a description of and report problems regarding any project and look up experience of any employee.
3. Engineers, assigned to projects, can make changes and review changes related to their projects.
4. Quality engineers, in addition to being granted engineers' rights, can inspect the quality of projects they are assigned to.
5. Production engineers, in addition to possessing engineers' rights, can create new releases.
6. The project lead, in addition to possessing the rights granted to production and quality engineers, can also close problems.
7. The director, in addition to being granted the rights of project leads, can manage employees (assign them to projects, un-assign them from projects, look up experience, add new records to their experience, and fire them) and close engineering projects.

Figure 8: Sample authorization policy for the example shown in Figure 6 describes what actions are allowed. All other actions are denied.

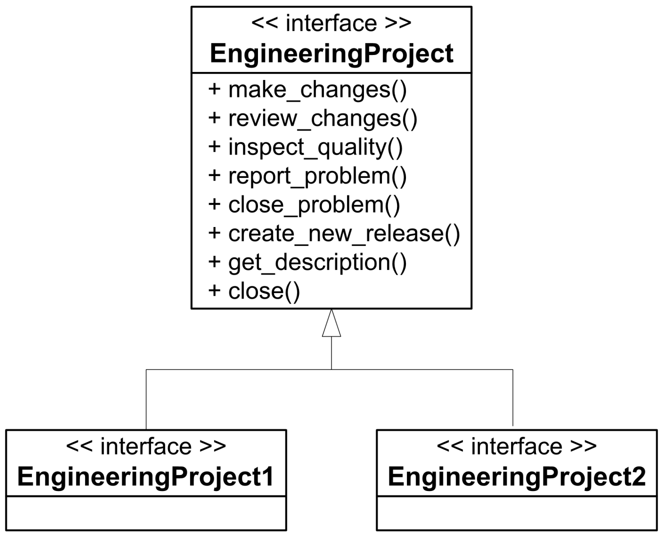


Figure 9: EngineeringProject interface hierarchy

Employee.get_name, Employee.assign_to_project,
Employee.unassign_from_project, Employee.add_experience,
Employee.get_experience, Employee.fire,
EngineeringProject1.inspect_quality, EngineeringProject1.make_changes,
EngineeringProject1.report_problem, EngineeringProject1.review_changes,
EngineeringProject1.close, EngineeringProject1.close_problem,
EngineeringProject1.create_new_release, EngineeringProject1.get_description,
EngineeringProject2.inspect_quality, EngineeringProject2.make_changes,
EngineeringProject2.report_problem, EngineeringProject2.review_changes,
EngineeringProject2.close, EngineeringProject2.close_problem,
EngineeringProject2.create_new_release, EngineeringProject2.get_description
}.

- $RIGHTS = \{gn, atp, ufp, ae, ge, f, mc1, rc1, iq1, rp1, cp1, cnr1, gd1, c1, mc2, rc2, iq2, rp2, cp2, cnr2, gd2, c2\}$. We used the first letters of each operation to create a corresponding right.
- RR is shown in Table 7.
- $D = \{C\}$
- $Inst = \{E_e, E_{ed}, E_{e1}, E_{e2}, E_{pe1}, E_{pe2}, E_{qe1}, E_{qe2}, E_{pl1}, E_{pl2}, E_{dir}, EP_{prj1}, EP_{prj2}\}$.
- $DOBS$: Object EP_{prj1} is an instance of EngineeringProject1, and EP_{prj2} is an instance of EngineeringProject2. All other elements of $Inst$ are instances of interface Employee.
- $A = \{e, ed, e1, e2, pe1, pe2, qe1, qe2, pl1, pl2, dir\}$. All these attributes have type *Role*.
- GR is shown in Table 8.
- $get\_all\_effective\_rights(d : D, u : 2^A) \equiv \bigcup_{\forall a \in u} \{r|(a, d, r) \in GR\}$—union of granted rights per attribute.

The CORBA protection system configuration described above allows enforcement of the sample policies listed in Figure 8. For example, a lead of project 1 with role pl1 activated is able to invoke the operations get_name and get_experience on all implementations of the interface Employee as well as all but close operations on all implementations of the interface EngineeringProject1.

The extension of the CORBA protection state to support the ANSI Core RBAC reference model system for this scenario is straightforward. For example, $PRMS$ is almost the same set as $IOPS$, except that the domain part of each tuple has value d1. We assume that Hierarchical RBAC support is implemented in the PrincipalAuthenticator, which activates specified role(s) as well as all roles junior to the specified one(s). No separation of duty constraints are necessary to enforce policies for this example (Figure 8).

Observe that significant administrative overhead is associated with the configuration of the CORBA protection system in this solution. The overhead is due to the gratuitous use of a separate interface (EngineeringProject(1,2)) per project. This is because we purposely limited our solution to a single-access policy domain. We show in the following section how the unnecessary redundancy of protection system configuration data could be eliminated by employing several access policy domains.

5.3.2 Multiple Access-Policy Domains Solution

Once we have an access policy domain per project, we can go back to using one EngineeringProject interface for all projects. We chose to use three domains, as shown in Table 9. The following configuration of a system protection state could be used to implement the policies listed in Figure 8:

Operation	**Right**
Employee.get_name	gn
Employee.assign_to_project	atp
Employee.unassign_from_project	ufp
Employee.add_experience	ae
Employee.get_experience	ge
Employee.fire	f
EngineeringProject1.get_description	gd1
EngineeringProject1.inspect_quality	iq1
EngineeringProject1.make_changes	mc1
EngineeringProject1.review_changes	rc1
EngineeringProject1.report_problem	rp1
EngineeringProject1.close_problem	cp1
EngineeringProject1.create_new_release	cnr1
EngineeringProject1.close	c1
EngineeringProject2.get_description	gd2
EngineeringProject2.inspect_quality	iq2
EngineeringProject2.make_changes	mc2
EngineeringProject2.review_changes	rc2
EngineeringProject2.report_problem	rp2
EngineeringProject2.close_problem	cp2
EngineeringProject2.create_new_release	cnr2
EngineeringProject2.close	c2

Table 7: Required rights for the solution with single domain

Privilege Attribute	**Rights**
e	gn
ed	ge, gd1, gd2, rp1, rp2
e1	mc1, rc1
pe1	cnr1
qe1	iq1
pl1	cp1
e2	mc2, rc2
pe2	cnr1
qe2	iq1
pl2	cp1
dir	atp, ufp, ge, ae, f, c1, c2

Table 8: Granted rights matrix for the solution with single domain

Interface Instance	**Domains**		
	C	EP_1	EP_2
E_e	✓		
E_{ed}	✓		
E_{dir}	✓		
E_{e1}		✓	
E_{pe1}		✓	
E_{qe1}		✓	
E_{pl1}		✓	
EP_{prj1}		✓	
E_{e2}			✓
E_{pe2}			✓
E_{qe2}			✓
E_{pl2}			✓
EP_{prj2}			✓

Table 9: Interface instance domain membership matrix (IDM) for multi-domain solution

- $A, OPS, get\_all\_effective\_rights$, is the same as in the single-domain solution
- $I = \{Employee, EngineeringProject\}$.

  $IOPS = \{$
  Employee.get_name, Employee.assign_to_project,
  Employee.unassign_from_project, Employee.add_experience,
  Employee.get_experience, Employee.fire,
  EngineeringProject.inspect_quality, EngineeringProject.make_changes,
  EngineeringProject.report_problem, EngineeringProject.review_changes,
  EngineeringProject.close, EngineeringProject.close_problem,
  EngineeringProject.create_new_release, EngineeringProject.get_description
  $\}$.

- $RIGHTS = \{gn, atp, ufp, ae, ge, f, mc, rc, iq, rp, cp, cnr, gd, c\}$
- RR is shown in Table 10. It is the same as in Table 7 except one interface EngineeringProject is used instead of two identical interfaces with different names.
- D=$\{C, EP_1, EP_2\}$
- $Inst$ is the same as in in the single-domain solution except that EP_{prj1} and EP_{prj2} implement the EngineeringProject interface.
- $DOBS$ is shown in Table 9.
- GR is shown in Table 11.

The CORBA protection system configuration described above allows enforcement of the same policies as the configuration in the solution for a single domain. This time, there is no need to have either separate EngineeringProject(1,2) interfaces or redundant rights per project. In addition, RR (Table 10) is more compact and therefore might be more comprehensible for administrators.

The use of separate domain per engineering project is responsible for the ability to support more flexible policies. For example, the GR in Table 11, in addition to the sample policies

Operations	**Rights**
Employee.get_name	gn
Employee.assign_to_project	atp
Employee.unassign_from_project	ufp
Employee.add_experience	ae
Employee.get_experience	ge
Employee.fire	f
EngineeringProject.get_description	gd
EngineeringProject.inspect_quality	iq
EngineeringProject.make_changes	mc
EngineeringProject.review_changes	rc
EngineeringProject.report_problem	rp
EngineeringProject.close_problem	cp
EngineeringProject.create_new_release	cnr
EngineeringProject.close	c

Table 10: Required rights matrix for multi-domain solution

Role Attribute	**Domains**		
	C	EP_1	EP_2
e	gn	gn	gn
ed		gd, rp, ge	gd, rp, ge
e1		mc, rc	
qe1		iq	
pe1		cnr	
pl1		cp, ae	
e2			mc, rc
qe2			iq
pe2			cnr
pl2			cp, ae
dir	atp, ufp, ge ae, f, c	atp, ufp f, c	atp, ufp f, c

Table 11: Granted rights for multi-domain solution

described earlier, supports a policy that allows project leads to add experience (right ae) to the records of the employees working under the supervision of the leaders. In order to enable it, whenever an employee is assigned to a project the corresponding Employee object is moved to the project's access-policy domain. This ability, however, comes at the price of allowing project leads to add experience to their own records, violating the separation of duty principle. It is also possible to enforce finer-grain policy, where employees of the engineering department can look up the experience of their colleagues from the same department only.

The extension of the CORBA protection state to support the ANSI Core RBAC reference model system is more interesting in the multi-domain scenario. Derived from the CORBA protection state, Table 12 depicts OBS (columns 2-3), $PRMS$ (columns 1-3), and the assignment of permissions to roles (PA). The table serves also as an example of Lampson's access matrix [Lam71] for a CORBA system, where roles are Lampson's domains and permissions are objects.

Having analyzed the support for the reference model of the ANSI RBAC in CORBA, we move on to presenting results of our analysis with regard to the support of ANSI RBAC functions.

5.4 Functional Specification

This section reports on the the results of our analysis of CORBASec support for system and administrative functional specifications of ANSI RBAC. We examined each ANSI RBAC function defined in Section 6 of [ANS04] on the subject of its support by a CORBASec implementation conforming to Security Functionality Level 2. That is, we did not assume any CORBASec functionality other than that required for Level 2 conformance [OMG02b, p.374]. Particular implementations of CORBASec might provide additional functionality, and, as a result, support more ANSI RBAC functions. Examining support for ANSI RBAC on an implementation-by-implementation basis was, however, beyond the scope of this paper.

Results of our examination suggest that the CORBASec functionality, as defined through the data structures and interfaces in Version 1.8, is largely insufficient for implementing ANSI RBAC functions. Specifically, Hierarchical and Constrained RBAC functions cannot be supported without extending an implementation beyond what CORBASec defines.

Even for Core RBAC, we found that most functions cannot be supported as is, as the summary of our analysis in Table 13 indicates. Because the CORBASec specification is not concerned with the administrative and run-time management of user accounts, user attributes, and principals (which are sessions in RBAC terms), the following functions prescribed for Core RBAC implementations cannot be supported without implementation-specific extensions: $AddUser$, $DeleteUser$, $AssignUser$, $DeassignUser$, *AddRole*, *DeleteRole*. The rest of this section discusses implementation of the other Core RBAC functions using CORBASec and its application programming interfaces (APIs), and identifies the functionality necessary for supporting these functions that is missing from CORBASec.

GrantPermission, RevokePermission functions enable changes to the permission assignment (PA) set. The CORBASec operations set_required_rights, grant_rights, revoke_rights, and replace_rights (described in Section 4.1) allow modifications to RR and GR, and therefore PA, leading us to conclude that these CORBASec operations are sufficient for implementing $GrantPermission$ and $RevokePermission$ functions. As an illustration, in Algorithm 6 we provide an operational definition of the function $GrantPermission$, using CORBASec API. $RevokePermission$ can be defined likewise, except that more care needs to be exercised in making sure that granted rights are revoked in all domains associated with the given permission.

CreateSession function creates a given session with a given user account as the owner. CORBASec utilizes the notion of the PrincipalAuthenticator—described in Section 4.1—whose functionality is expected to encompass the authentication of the CORBA principal, and create the principal's credentials, the equivalent of the ANSI RBAC session. Thus, even though CORBASec does not define an operation for creating sessions, a functional im-

Permissions			**Roles**										
Operations	Objects												
	Interface	Domain	e	ed	e1	pe1	qe1	pl1	e2	pe2	qe2	pl2	dir
get_name	Employee	C	√										
assign_to_project													√
unassign_from_project													√
add_experience													√
get_experience													√
fire													√
get_name	Employee	EP_1	√										
assign_to_project													√
unassign_from_project													√
add_experience								√					
get_experience				√									√
fire													√
get_name	Employee	EP_2	√										
assign_to_project													√
unassign_from_project													√
add_experience												√	
get_experience				√									√
fire													√
get_description	EngineeringProject	EP_1		√									
inspect_quality							√						
make_changes					√								
review_changes					√								
report_problem				√									
close_problem								√					
create_new_release						√							
close													√
get_description	EngineeringProject	EP_2		√									
inspect_quality											√		
make_changes									√				
review_changes									√				
report_problem				√									
close_problem												√	
create_new_release										√			
close													√

Table 12: RBAC objects (OBS), permissions ($PRMS$), and the assignment of roles to permissions (PA) for the multi-domain case.

plementation of CORBASec would either rely on the underlying security infrastructure or implement an equivalent of *CreateSession* utilized by PrincipalAuthenticator. Since the notion of user accounts is missing from CORBA, this function cannot be completely supported without an implementation-specific extension.

DeleteSession function deletes a given session with a given user account as the owner. Even though CORBASec's Credentials interface defines the operation destroy, this and other operations on Credentials can be invoked only within the operating system process where the Credentials object resides. Another limitation stems from the fact that there can be multiple copies of the same Credentials object, making their complete deletion difficult to implement; the CORBA specification does not provide a means for enumerating all copies of a given Credentials object. For the above reasons, and because the notion of user

Core RBAC Functions		**Functionality that needs to be defined to support this function**						
	Completely supported in CORBASec	user management	user representation	attribute management	credentials deletion	enumeration of access policy domains	attribute to credential translation	retrieving attributes of a given principal
Administrative Commands								
AddUser		√						
DeleteUser		√						
AssignUser		√						
DeassignUser		√						
AddRole				√				
DeleteRole				√				
GrantPermission	√							
RevokePermission	√							
Supporting System Functions								
CreateSession		√						
DeleteSession		√			√			
AddActiveRole		√						
DropActiveRole		√						
CheckAccess	√							
Review Functions								
AssignedUsers			√					
AssignedRoles			√					
Advanced Review Functions								
RolePermissions						√		
SessionPermissions						√		
UserPermissions		√				√		
SessionRoles								√
RoleOperationsOnObject							√	
UserOperationsOnObject			√					

Table 13: Functions defined by ANSI Core RBAC and their support in CORBA

accounts is missing from CORBA, we concluded that *DeleteSession* would have to be implementation-specific.

AddActiveRole, DropActiveRole functions add/delete a role as an active role of a session whose owner is a given user account. Even though CORBASec does not define a function with semantics compatible to *AddActiveRole*/*DropActiveRole* according to Liskov's *substitution principle* [LW94], it does specify the set_attributes operation on the SecurityLevel2.Credentials interface, enabling a privileged caller to modify attributes on a credential associated with a particular principal. However, the logic for checking the

```
GrantPermission($rbacObject, operation, role : NAME$)
 1: $permission \leftarrow (operation, rbacObject)$
 2: $interface \leftarrow interface(permission)$
 3: $(rights, combinator) \leftarrow$ get_require_rights($interface, operation$)
 4: if $combinator$ is "any" then
 5:    $R \leftarrow$ any right from $rights$
 6: else
 7:    $R \leftarrow rights$ {Combinator is "all"}
 8: end if
 9: Select any access-policy domain $d$ from $domains(permission)$
10: grant_rights(d, $role, R$)
```

Algorithm 6: Operational definition of the function *GrantPermission*, which grants a role the permission to perform an operation on an ANSI RBAC object.

preconditions $session \in user\_sessions(user)$ and $(user \mapsto role) \in UA$ would have to be implementation-specific due to the lack of standardized support for user-account management in CORBA deployments.

CheckAccess returns a Boolean value indicating whether the subject of a given session is allowed, or not, to perform a given operation on a given object. This function is equivalent to is_authorized, which is defined by Algorithm 2.

AssignedUsers, AssignedRoles return the set of users/roles assigned to a given role/user, respectively. Both functions require the notion of user, which is missing from CORBASec, making these functions implementation-specific.

RolePermissions, SessionPermissions return the set of permissions (op, obj) granted to a given role or session, respectively. Implementations of these functions would require querying each instance of the DomainAccessPolicy interface in the given CORBA deployment in order to determine the content of the role's row in the granted rights matrix (see Tables 3 and 11). However, due to the lack of standard mechanisms for enumerating all objects in a CORBA deployment in general and all DomainAccessPolicy objects in particular, the querying would have to be implementation-specific.

UserPermissions returns the permissions a given user gets through his/her assigned roles. This function would be implementation-specific due to the lack of both standard mechanisms for enumerating all access-policy domains and a standardized support for managing user accounts.

SessionRoles returns the active roles associated with a session. This function is partially supported by the CORBASec. Any compliant implementation of CORBASec must implement the Current.get_attributes operation, which allows retrieving security attributes of a specific type (e.g., role) for the principal associated with the current execution thread. However, the standard does not define an operation for retrieving attributes for an arbitrary principal or associating it with the current execution thread.

RoleOperationsOnObject returns the set of operations a given role is permitted to perform on a given RBAC object. Unlike the case of *RolePermissions*, support for this function does not require enumerating all DomainAccessPolicy objects. A reference of the corresponding CORBA object in question is sufficient for employing the AccessDecision.access_allowed operation for determining if a given role is allowed to invoke a particular operation on a given object. In order to determine the rights of a given role on all operations of the RBAC object, the CORBA Reflection facility [OMG06] can be used for enumerating operations implemented by the object. CORBASec, however, does not define operations for creating a valid credential—a required format of the input parameter for access_allowed—out of a security attribute (e.g., particular role). Therefore, the translation would have to be implementation-specific.

UserOperationsOnObject returns the set of operations a given user is permitted to perform on a given RBAC object. This function would be implementation-specific due to the lack of a standardized notion of user.

6 Discussion

The results of our analysis suggest that the CORBASec functionality—as defined through the data structures and interfaces in Version 1.8—is largely inadequate for implementing ANSI RBAC functions without resorting to vendor-specific extensions of the CORBAsec implementation. Even in the case of Core RBAC alone—the mandatory part of any compliant implementation of ANSI RBAC—there are three major causes of this inadequacy.

One is the lack of a standard mechanism for enumerating all objects that implement the DomainAccessPolicy interface in a CORBA deployment, which is necessary for enumerating all permissions granted to the corresponding user/principal/role. This limitation is due to the lack of support for enumerating all CORBA objects in a deployment. CORBA was originally positioned as a generic middleware architecture scalable to Internet-wide deployments [Sie00]—where partial failures that are hard to detect and recover from are endemic—of potentially massive numbers of fine-grained. Such objects would range from intermittent (e.g., shopping cart for an online store customer) to long-lived and persistent (e.g., Parlay [ETS05]). Thus, its design intentionally avoids requirements for maintaining a view of the global state of a CORBA deployment—a prerequisite for a standardized capability to enumerate (and therefore register) all objects, or just all instances of DomainAccessPolicy. Although maintaining a view of the global state of an Internet-wide deployment for any application is clearly unfeasible, the reality is that CORBA deployments enjoy small-to-moderate scale [Hen06], are confined by enterprise boundaries—with the notable exception of Parlay—and commonly feature only course-grained [YD96], persistent objects. It seems reasonable to expect a standardized capability for enumerating all instances of the DomainAccessPolicy interface given that recent results demonstrate the feasibility of distributed lock [Bur06], table [CDG$^+$06], and hash table [ZHS$^+$04] data structures capable of holding tens of thousands of records and serving similar-sized populations of active clients. Such a capability might be featured , for example, only in enterprise-scale deployments of CORBA.

However, even with scalable data structures, *strict consistency* among multiple views of the system's global state is commonly believed to be essentially impossible [TS01], leaving only weaker consistency models to choose from. The semantics of the weaker models, however, varies widely, from data-centric *linearizability* [HW90]—which requires a globally available clock with finite precision—to client-centric *eventual consistency* [TS01]—which guarantees that all views eventually become consistent, but only if no updates take place for a long time. The choice of the acceptable consistency model(s) has to be explicit in ANSI RBAC in order for it to be applicable to those distributed systems where the protection state is distributed, as is the case with CORBA.

Another caveat is that other commercial-grade distributed technologies—e.g., COM+ [Obe00], EJB [DYK01], Grid [JBFT05], Web Services, and the HTTP-based Web—also lack a standardized capability for enumerating all resources (or just resources of a particular type) in a deployment. If most mainstream distributed technologies do not define this capability, is the reliance of ANSI RBAC on it realistic? Can the ANSI RBAC standard be revised to avoid the assumption that it is possible to enumerate all resources (and therefore permissions) in a system?

The second major limitation of CORBASec is its lack of the notion of user accounts and support for their management (i.e., adding, deleting, (un)assigning to/from roles), as well as the lack of user representation. According to our analysis, which is summarized in Table 13, this limitation results in over one-half of Core RBAC functions being dependent on vendor-specific extensions. The architects of CORBASec intentionally left the notion of user and support for user management beyond the scope of the specification. The abstraction of PrincipalAuthenticator serves as an implementation-specific and technology-specific bridge between CORBASec run-time, which is concerned with principal credentials, and users, on behalf of which CORBA clients invoke operations on objects. PrincipalAuthenticator also performs user authentication and, if successful, activates roles at run-time. In order to provide standard support for administering and reviewing user accounts, their roles and their sessions, the corresponding administrative interfaces would need to be added to CORBASec. However, such a revision would be contrary to the emerging state of practice for application systems.

The notable trend in IT systems design is to re-allocate functionality for administering user accounts, and in some cases permissions, to single sign-on (SSO) [PM03] solutions for new applications [Got05] and to identity management (IDM) solutions for existing applications [BS03]. As a result, user accounts, and sometimes permissions, are administered across multiple application instances and types "outside" of the applications themselves. Therefore, an application system can only be successfully evaluated for compliance with ANSI RBAC when the application is considered together with the corresponding SSO or IDM solution. This condition makes evaluation of support for ANSI RBAC prohibitively expensive for systems designed to work in conjunction with multiple SSO or IDM solutions, as the evaluation would have to be performed for every combination of the system and the supporting SSO and/or IDM. Defining a separate ANSI RBAC profile for SSO and IDM solutions is a possible alternative to explore further.

Even if CORBASec supported user accounts and their management, the inability to enumerate all CORBA principals related to a specific user (e.g., those with the same value of the auditId or accessId CORBASec attributes) would still prevent CORBASec from providing complete support for such Core RBAC functions as *AddUser*, *CreateSession*, *DeleteSession*, *AddActiveRole*, *DropActiveRole*. All of their definitions use the helper function *user_sessions*, which can only be implemented if a CORBA deployment keeps track of all principals—CORBASec surrogates of sessions—for every user. However, principal tracking is prohibitively expensive for CORBA, and we believe for other distributed systems, as well, due to the need for maintaining a view of the global state in the presence of partial failures. Thus, we echo the suggestion made by Li et al. [LBB06] to remove the notion of sessions from Core RBAC and introduce it in a separate, optional ANSI RBAC component.

Results of our analysis discussed in Sections 5.1.2 and 5.1.3 indicate that most functions for Hierarchical and Constrained RBAC options of the ANSI RBAC standard cannot be supported without extending a CORBASec implementation with additional operations. Even though there are at least three options for supporting role hierarchies, additional operations would have to be added to CORBASec in order to provide standard support for modification and review of the role hierarchy and for the functions *authorized_users* and *authorized_permissions*. Since support for Constrained RBAC is contingent on the support for user accounts, role hierarchies, and role activation, the standardization of support for these three in CORBASec is a prerequisite for standardization of Constrained RBAC in CORBASec.

In summary, while generic and versatile, the access control architecture of CORBASec does not define standard functionality for enumerating all DomainAccessPolicy objects in a deployment and all sessions for a given user. It also lacks the notion of user accounts and their run-time representation, as well as support for their management. These are three major roadblocks on the path of CORBASec conforming to ANSI RBAC. Our results are not conclusive, however, as to whether this mismatch between CORBASec and ANSI RBAC is exclusively due to the shortcomings of the former or also involves the failure the latter to be sufficiently general.

7 Conclusion

Understanding middleware access control mechanisms is critical for protecting the resources of enterprise applications. In this paper, we described in detail the architecture of access control mechanisms in CORBA Security and defined a configuration of the CORBA protection system in precise and unambiguous terms of set theory. Using the configuration definition, we suggested an algorithm that formally specifies the semantics of authorization decisions in CORBASec.

We analyzed CORBASec in relation to its support for ANSI RBAC components and discussed what functionality needs to be implemented, besides compliance with the CORBASec standard, in order to support Core and Hierarchical RBAC. We suggested steps for translating an arbitrary ANSI RBAC policy into CORBA protection state. We illustrated our discussion with a single access-policy domain and a multi-domain examples of the CORBASec protection system configuration, which supports a sample role hierarchy and access policies. Finally, we analyzed CORBASec support for the functional specification of ANSI RBAC.

The results indicate that CORBASec falls short of supporting even functional Core RBAC due to (1) the lack of a standard mechanism for enumerating all DomainAccessPolicy objects in a CORBA deployment, (2) the lack of explicit user representation as well as the notion of user accounts and support for their management, and (3) the inability to enumerate all CORBA principals related to a specific user. Custom extensions are necessary in order for implementations compliant with CORBASec to support ANSI RBAC required or optional components. These results can be interpreted as either a demonstration of the inadequacy of CORBASec in supporting ANSI RBAC, or as an indication of ANSI RBAC not being sufficiently general. Examination of other representative systems on the subject of their support for ANSI RBAC may clarify this question.

The work presented in this paper establishes a framework for implementing as well as for assessing implementations of ANSI RBAC using CORBA Security. The results provide directions for CORBA Security developers implementing ANSI RBAC in their systems, and offer criteria to users for selecting such CORBA Security implementations that support required and optional components of ANSI RBAC.

Acknowledgments

Philippe Kruchten and Matei Ripeanu provided helpful feedback on the early versions of this paper, and Craig Wilson helped us to improve its readability. Special thanks to Jason Crampton whose very valuable suggestions and detailed comments on the paper drafts were instrumental in improving it.

References

[Ahn00] Gail-Joon Ahn. Role-based access control in DCOM. *Journal of Systems Architecture*, 46(13):1175–1184, 2000.

[ANS04] ANSI. *ANSI INCITS 359-2004 for Role Based Access Control*. American National Standards Institute, 2004.

[AS01] Gail-Joon Ahn and Ravi Sandhu. Decentralized user group assignment in Windows NT. *The Journal of Systems and Software*, 56(1):39–49, 2001.

[Bar97] Larry S. Bartz. hyperDRIVE: leveraging LDAP to implement RBAC on the web. In *RBAC '97: Proceedings of the Second ACM Workshop on Role-based Access Control*, pages 69–74, New York, NY, USA, 1997. ACM Press.

[BD99] Konstantin Beznosov and Yi Deng. A framework for implementing role-based access control using CORBA security service. In *Fourth ACM Workshop on Role-Based Access Control*, pages 19–30, Fairfax, Virginia, USA, 1999.

[BK98] Nat Brown and Charlie Kindel. Distributed component object model protocol (DCOM/1.0). Technical Report draft-brown-dcom-v1-spec-03.txt, Microsoft Corporation, January 1998.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, MITRE, March 1975.

[BLL03] Ruth Baylis, Paul Lane, and Diana Lorentz. Oracle© database administrator's guide, December 2003. 10g Release 1 (10.1).

[BS03] D.A. Buell and R. Sandhu. Identity management. *IEEE Internet Computing*, 7(6):26–28, Nov.-Dec. 2003.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation*, pages 335–350, Seattle, WA, USA, November 6-8 2006.

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *The 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI06)*, pages 205–218, Seattle, WA, USA, November 6-8 2006.

[Cha03] Thomas M. Chalfant. Role based access control and secure shell - a closer look at two solaris^{TM}operating environment security features. *Sun BluePrints^{TM}OnLine*, June 2003.

[CO02] David W. Chadwick and Alexander Otenko. The PERMIS X.509 role based privilege management infrastructure. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 135–140, New York, NY, USA, 2002. ACM Press.

[DYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, 2001.

[ETS05] ETSI. *Open Service Access (OSA) - Application Programming Interface (API) - Part 1: Overview*, v1.1.1 edition, 2005. pp. 61.

[Fad99] Glenn Faden. RBAC in UNIX administration. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 95–101, New York, NY, USA, 1999. ACM Press.

[FBK99] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):34–64, February 1999.

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, 1992. National Institute of Standards and Technology/National Computer Security Center.

[FSG$^+$01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[GDS97] Mats Gustafsson, Benoit Deligny, and Nahid Shahmehri. Using NFS to implement role-based access control. In *WET-ICE '97: Proceedings of the 6th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 299–304, Washington, DC, USA, 1997. IEEE Computer Society.

[Giu99] Luigi Giuri. Role-based access control on the Web using Java. In *RBAC '99: Proceedings of the Fourth ACM Workshop on Role-based Access Control*, pages 11–18, New York, NY, USA, 1999. ACM Press.

[Got05] G. Goth. Identity management, access specs are rolling along. *IEEE Internet Computing*, 9(1):9–11, Jan.-Feb. 2005.

[Gut01] Kurt Gutzmann. Access control and session management in the HTTP environment. *IEEE Internet Computing*, 5(1):26–35, 2001.

[Hen06] Michi Henning. The rise and fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[IBM05] IBM. IBM informix dynamic server administrator's guide, December 2005. Informix Dynamic Server 10.0; Document ID: G251-2267-02.

[JBFT05] Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi. *Introduction to Grid Computing*. IBM Press, 2005.

[Kar00] Gunter Karjoth. Authorization in CORBA security. *Journal of Computer Security*, 8(2/3):89–108, 2000.

[Lam71] Butler W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, page 437, New York, NY, USA, 1971. ACM Press.

[LBB06] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ansi standard on role based access control. CERIAS and Department of Computer Science, March 3 2006.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mic96] Microsoft. Dcom technical overview, 1996.

[Mic98] Microsoft. Dcom architecture, 1998.

[MyS07] MySQL AB. MySQL. http://www.mysql.com, 2007.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[Obe00] Robert J. Oberg. *Understanding & programming COM+: a practical guide to Windows 2000 DNA*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[OF02] Rafael R. Obelheiro and Joni S. Fraga. Role-based access control for CORBA distributed object systems. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 53, Washington, DC, USA, 2002. IEEE Computer Society.

[OMG98] OMG. CORBAservices: Common object services specification, 1998.

[OMG99] OMG. The common object request broker: Architecture and specification. Specification formal/99-10-08, Object Management Group, 1999.

[OMG02a] OMG. *Authorization Token Layer Acquisition Service (ATLAS) Specification*. The Object Management Group (OMG), October 2002.

[OMG02b] OMG. CORBAservices: Common object services specification, security service specification v1.8, 2002.

[OMG04] OMG. Common object request broker architecture: Core specification v3.0.3, 2004.

[OMG06] OMG. CORBA Reflection v.1.0. OMG document# formal/06-05-03, May 2006.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, 2000.

[PM03] Andreas Pashalidis and Chris J. Mitchell. A taxonomy of single sign-on systems. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia*, volume 2727 of *Lecture Notes in Computer Science*, pages 249–264. Springer, July 9-11 2003.

[PP95] Tom Parker and Denis Pinkas. Sesame v4 - overview. Technical report, SESAME, December 1995.

[PSA01] Joon S. Park, Ravi Sandhu, and Gail-Joon Ahn. Role-based access control on the web. *ACM Transactions on Information and System Security (TISSEC)*, 4(1):37–71, February 2001.

[RS98] C. Ramaswamy and R. Sandhu. Role-based access control features in commercial database management systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 503–511, Arlington, VA, USA, 1998. National Institute of Standards and Technology/National Computer Security Center.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SFK00] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *ACM Workshop on Role-Based Access Control*, Berlin, Germany, 2000. ACM.

[SGJ98] R. Sandhu and Ahn. G-J. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 486–502, Arlington, Virginia, USA, 1998. National Institute of Standards and Technology/National Computer Security Center.

[Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley & Sons, 2000.

[SP98] Ravi Sandhu and Joon S. Park. Decentralized user-role assignment for web-based intranets. In *the Third ACM Workshop on Role-Based Access Control*, pages 1–12, Fairfax, Virginia, USA, 1998. ACM Press.

[SS96] Richard Mark Soley and Christopher M. Stone. *Object Management Architecture Guide.* John Wiley & Sons, 492 Old Connecticut Path, Framingham, MA 01701 USA, 3 edition, 1996.

[Sun00] Sun Microsystems Inc. RBAC in the Solaris^{TM} operating environment. http://www.sun.com/software/whitepapers/wp-rbac/wp-rbac.pdf, 2000. White Paper.

[Syb05] Sybase Inc. System administration guide: Volume 1 – Adaptive Server © Enterprise 15.0, October 2005. Document ID: DC31654-01-1500-02.

[TM06] Samantha Tran and Manoj Mohan. Security information management challenges and solutions. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0607tran/index.html, 2006.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[WdSFW$^+$02] C. M. Westphall, Joni da Silva Fraga, M. S. Wangham, R. R. Obelheiro, and Lau Cheuk Lung. PoliCap—proposal, development and evaluation of a policy service and capabilities for CORBA Security. In *SEC '02: Proceedings of the IFIP TC11 17th International Conference on Information Security*, pages 263–274, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[WF99] C. Westphall and J. Fraga. A large-scale system authorization scheme proposal integrating Java, CORBA and web security models and a discretionary prototype, December 1999. IEEE Latin American Network Operations and Management Symposium (LANOMS'99), Rio de Janeiro, Brazil.

[WHK97] M. Wahl, T. Howes, and S. Kille. RFC 2251: Lightweight directory access protocol (v3), 1997.

[YD96] Zhonghua Yang and Keith Duddy. CORBA: a platform for distributed object computing. *SIGOPS Oper. Syst. Rev.*, 30(2):4–31, 1996.

[ZHS$^+$04] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[ZM04] Wei Zhou and Christoph Meinel. Implement role based access control with attribute certificates. In *The 6th International Conference on Advanced Communication Technology (ICACT2004)*, volume 1, pages 536–541, Korea, Feb 2004. National Computerization Agency, Electronics and Telecommunications Research Institute, Korea.