MP 98B

MITRE PRODUCT

Distributed Object Computing (DOC) Security: Paradigms and Strategies

November 1998

Deborah J. Bodeau
Charles M. Schmidt
Vipin Swarup
F. Javier Thayer

MITRE

Center for Integrated Intelligence Systems
Bedford, Massachusetts

Table of Contents

1. INTRODUCTION

This report describes the status of distributed object computing (DOC) security. It proposes a strategy to enable evolution to more secure DOC systems and secure interoperability among different DOC systems.

Three DOC paradigms are discussed: the Object Management Group's (OMG's) CORBA (Common Object Request Broker Architecture); composable objects, exemplified by Microsoft's Component Object Model (COM); and mobile objects, exemplified by Java with Remote Method Invocation (RMI). Of these, only CORBA was originally intended to enable distributed object computing. Due to this objective, to the clarity and extent of its documentation, and to its maturity, CORBA concepts and strategies are influential in the other paradigms.

This report is organized as follows: Section 2 presents a framework for characterizing DOC paradigms and an overview comparison of how key concepts are used, interpreted, or refined in the three representative paradigms. This framework is needed because documentation commonly mixes motivation, conceptual models, and technical details. Sections 3 through 5 present overviews of CORBA, COM, and Java RMI using this framework. These overviews are intended to highlight security concerns and to suppress the implementation details that make most presentations of the DOC technologies lengthy, complex, and hard to understand. Section 6 identifies security issues specific to the three paradigms and to interoperability among systems that use different paradigms. Section 6 also proposes strategies for resolving some of those issues. Section 7 presents initial progress following one strategy, that of developing firmer theoretical foundations. The list of references emphasizes resources that can be found on the World-Wide Web. The appendix provides a concise presentation of information about security-relevant objects, interfaces, and attributes to facilitate the development of interoperability bridges. This information is dispersed throughout the CORBASec, COM, and Java specifications and documentation.

1.1 CONCEPTS AND TERMINOLOGY

In this subsection, we briefly survey concepts and terminology used in this paper.

We will discuss several kinds of *frameworks*: for distributed computing, for architectural definition, for system specification, for characterization of DOC paradigms, and for implementation. A framework is a structure for organizing and illuminating relationships among concepts and constructs. It includes identification of key concepts or classes of concepts, principles for defining relationships among these concepts, and techniques for creating more detailed or specific constructs consistent with these concepts and principles. The term is usually modified to indicate either the framework's goal (e.g., characterization, specification) or the domain to which it is applied (e.g., types of systems, types of enterprises or missions). A framework is typically intended to apply to a large class of systems, architectures, and/or paradigms.

An *architecture* consists of three things: a set of constructs intended to be realized as hardware, firmware, or software components of a system; a set of possible relationships among those constructs; and rules or principles for refining the constructs and defining specific relationships.

A *system* is a concrete realization of an architecture; it includes hardware, firmware, software, storage and communications media, and information. A *distributed system* consists of a set of interdependent *platforms* (hardware and/or software). A hardware platform is a collection of hardware components treated as a unit by users or administrators (e.g., a desktop computer, a server, a router). A software platform is a set of software components which provide services to applications (e.g., an operating system together with utilities and drivers). Depending on the framework used to specify or describe it, a system can also include actions performed by users or administrators, and procedures or processes which include such actions together with actions or events that occur on a platform.

Distributed object computing assumes at least a three-level architecture[1]: applications, DOC middleware, and underlying system/network services. Middleware relies on underlying hardware, communications media, and operating system (OS) and network software to provide services to applications or functions. The distinguishing characteristic of DOC systems (*vice* distributed systems in general) is that applications or functions are conceptualized as *objects*. An object has one or more *interfaces* by which it can receive *requests* to perform functions or provide information.

DOC paradigms differ in the details of how objects are conceptualized. These differences are due to different interpretations and applications of object-oriented programming concepts to the distributed computing environment. In general, object-oriented programming focuses on the definition and use of *object classes*. An object class is a characterization of or template for a set of objects with common characteristics. The primary characteristic is the set of interfaces provided by all objects in the class. Other characteristics include object-internal data structures and relationships to other object classes. Most object-oriented paradigms support at least *inheritance*: one class can be defined as a subclass of another, thereby inheriting (possibly with modifications) the characteristics of its parent class, while acquiring additional characteristics unique to its definition. Support for multiple inheritance differs among object-oriented paradigms.

The term object is used in many ways. In its most abstract meaning, an object, characterized in terms of its parent class(es), its interfaces, and possibly also its internal data structures, is a definition or template. A realization of that template, represented by bits in storage, is an object as recognized by DOC middleware, which can *expose* the interfaces to potential clients. (Such an object can be described as *inactive*.) An *object*

[1] The three-level DOC architecture must not be confused with the three-tier architecture common to many distributed systems. A three-tier architecture places the client, server, and target application whose services the client seeks to use on three separate hardware platforms (or tiers). The client typically runs on a desktop or network computer, the server on a hardware platform that facilitates networking and multitasking, and the application on a large hardware platform (e.g., a mainframe). The DOC architecture does not assume different types of hardware platforms.

instance (also referred to as an object) occurs when that object is *activated* by the DOC middleware. As it runs on a specific hardware platform, its internal data structures are populated with values, and its interfaces (represented by pointers) are exposed to potential clients by the DOC middleware.

1.2 SECURITY CONCERNS

Four kinds of security concerns are common to DOC systems: those common to all computing systems, those arising from object orientation, those arising from distribution, and those related to middleware.

The conventional security concerns are for confidentiality, integrity, availability, and accountability. In general-purpose computing systems, these concerns are addressed by such security services as access control, security management, identification and authentication (I&A), and audit. These security services must be adapted to the DOC environment. One basis for adaptation is architectural level: which security-related actions can or should be taken by the underlying OS or network, which by the DOC middleware, and which by the objects themselves?

A second basis for adaptation is object *location transparency*. The underlying concept in a DOC system is that an object be able to invoke (or send messages or requests to) another object without consideration of the location of the target object. Location includes not only process space but also hardware platform. Historically, access control was described in terms of subjects (i.e., software running in a specific process space); the security-related information on which access control decisions were made was represented (sometimes implicitly) by the process context. In a DOC system, that information must be explicitly represented and associated with the request.

That information can be represented as a set of credentials. When a client makes a request, it does so on behalf of some entity (a human user or system entity), called the *principal*. The client may be acting with some set of privileges or authorizations. These may be attributes of the principal, the client, or the operating environment or context. Key security-related attributes of a request thus typically include the identity of the principal and the privileges or authorizations currently held by the client. Additional security-relevant attributes of a request can include the identity of the sources of privileges or authenticator of the principal's identity. Thus, central to the security of a DOC system are mechanisms for identification, authentication, and binding of credentials (privilege, authorization, and principal identity) information to a request.

The second source of concern is object orientation. As noted above, an object can exist in an inactive or active form. Access control mechanisms in conventional operating systems typically allow or disallow execution of a program. A DOC system may distinguish between *activation* (creation of a running object instance, which can entail considerable processing overhead) and *invocation* (use of an already-activated object). In addition, conventional access control mechanisms need handle only a limited set of access types (e.g., read, write, execute, manage). In an object system, controls can be applied to an object's interfaces as well as to the object as a whole.

The nature and complexity of the underlying object model can also be a source of concern. Object models can support such concepts as multiple inheritance and polymorphism. Thus, multiple policies can apply to the same object. Inconsistencies among those policies must be resolved.

The third source of concern is distribution. The fact that a sequence of object invocations spans hardware platforms means that concerns historically associated with communications security arise for DOC systems. The conventional network security services of message integrity, message confidentiality, and nonrepudiation must be adapted to the DOC environment. Distribution complicates security management since changes to access rights must be propagated across multiple platforms. Audit analysis (or intrusion detection) also requires synchronization and correlation of audit trails across multiple platforms.

The concept of a *security domain* must be adapted for the distributed environment. A security domain is a set of resources to which a common security policy applies (ISO, 1996a-b). In conventional platform-centric systems, any security domain was local to the platform. For a DOC system, there is no necessary relationship between a security domain and a platform. Depending on the DOC paradigm, a domain could include resources on multiple platforms. Domains can overlap, be hierarchically organized, or be disjoint. Within overlapping or hierarchical domains, security mechanisms must be capable of resolving potential security policy differences. Between disjoint domains, mutual trust agreements must be defined or negotiated, and gateway or bridging mechanisms are needed to implement those agreements.

The need to resolve inter-domain conflicts is not unique to DOC systems. A single platform might contain multiple domains, with controlled flow of control and information between them. However, possible relationships among domains within a single platform are fewer and less complex than in a distributed environment.

Finally, concerns arise from the fact that DOC architectures include middleware. DOC middleware must rely on the underlying security of the operating systems and networks on which it runs. While DOC middleware can provide access control to the objects under its control, it cannot isolate itself or those objects from malicious code. That isolation must be provided by the operating system. DOC middleware may rely on operating system security services, particularly for I&A; thus, the strength of the DOC product cannot be greater than that of the platform on which it runs. While DOC middleware can implement the communications security services identified above, the system remains vulnerable to attacks on network availability and to traffic analysis.

1.3 INTEROPERABILITY CONCERNS

Barriers to interoperability within a given DOC paradigm arise in several ways. Interoperability is needed between domains with different security policies. Developers use proprietary extensions to DOC specifications to provide richer functionality. The lack of detail in specifications results in different interpretations (e.g., different meanings of parameters) by different vendors.

Barriers to interoperability between different paradigms arise from their different goals, strategies for defining architectures, and supporting technologies. Gateways or bridges can provide some interoperability, but the translation of security parameters may be difficult.

1.4 FRAMEWORKS FOR UNDERSTANDING DISTRIBUTED OBJECT COMPUTING AND SECURITY

Several frameworks have been defined for describing or specifying distributed computing, distributed object computing, and/or DOC security. They include the Reference Model for Open Distributed Processing (RM-ODP), the Open Systems Interconnection (OSI) Security Frameworks, The Open Group's Distributed Computing Environment (DCE) and Architectural Framework, the Infospheres framework, and CORBA (described in Section 3).

RM-ODP (ISO, 1996a) is a specification framework. It consists of:

- Five *viewpoints* (enterprise, information, computational, engineering and technology). These provide a basis for specifying ODP systems.
- A *viewpoint language* for each viewpoint. These define concepts and rules for specifying ODP systems from the corresponding viewpoint.
- Specifications of the *functions* required to support ODP systems.
- *Transparency prescriptions*. These show how to use the ODP functions to achieve distribution transparency.

The RM-ODP architecture and the composition of functions are defined by the combination of the computational language, the engineering language and the transparency prescriptions.

The OSI security frameworks (ISO, 1996b) address security services in open systems environments, which include (but are not limited to) open distributed processing environments. OSI security frameworks are defined for authentication, access control, non-repudiation, confidentiality, integrity, security audit and alarms, and in (ISO, 1996c) key management.

The DCE framework consists of a middleware architecture and set of specifications for services and tools that enable distributed computing (TOG, 1998). (Background can be found in (Johnson, 1991; TOG, 1992, 1996, 1997b).) DCE components include threads, the Remote Procedure Call, the Distributed Time Service, the Distributed File Service, the Directory Service, and the Security Service. With the recent exception of C++, it does not provide standardized support for object-oriented languages. DCE thus is not a distributed *object* computing paradigm. However, some technologies developed within the DCE framework have been or are expected to be used to support DOC products.

These include the DCE Remote Procedure Call (RPC) and the Kerberos-based DCE security service.

The Open Group has defined an architectural framework as a conceptual tool to aid in the development of system architectures (TOG, 1997a). That framework is compatible with, but broader than, DCE, CORBA, and RM-ODP. That is, The Open Group Architectural Framework does not assume a distributed or object-oriented system. The Open Group has also defined a Common Data Security Architecture to address communications and data security problems for networked systems (TOG, 1997b). While this security architecture does not assume an object system, it has some similarities to CORBA, and provides an interesting threat model.

The goal of the California Institute of Technology (Caltech) Infospheres Project is to facilitate the development of distributed compositional systems, i.e., systems composed of interacting components. The Structured Framework for Distributed Object Computing (Chandry et al., 1997) is an implementation framework. That is, it includes not only concepts and rules, but also some middleware application interfaces (APIs). It is intended to enable rapid development of interactive processes that can run over the Internet. The framework defines four facets (or ways of describing DOC systems):

- Processes, i.e., persistent communicating objects. A process can be ready (active or waiting) or frozen. (This corresponds to the active/inactive distinction made above.)
- Personal networks, i.e., a conceptual wiring diagram of processes which communicate by passing messages.
- sessions, i.e., transactions or sequences of tasks performed by processes in a personal network.
- Infospheres, i.e., custom collections of processes for use in personal networks.

The framework identifies services needed by many distributed algorithms: locking, deadlock avoidance, termination detection, and resource reservation.

2. CHARACTERIZATION AND COMPARISON STRATEGIES

Comparisons among DOC paradigms, architectures, and products are difficult, because documentation commonly mixes motivation, conceptual models, and technical details. Section 2.1 therefore presents a framework for characterizing DOC paradigms. Section 2.2 provides an overview comparison of how key concepts are used, interpreted, or refined in the three representative paradigms. Section 2.3 provides an initial summary comparison, to serve as orientation to Sections 3 through 5.

2.1 CHARACTERIZATION FRAMEWORK

We have identified three levels of abstraction or "strata" that facilitate description, analysis, and comparison of DOC paradigms and of architectures, systems, and products developed within a paradigm. The three strata are conceptual, architectural, and implementation. Corresponding to each stratum is a type of documentation: model, specification, and product (or detailed design) documentation, respectively.

- Conceptual: At this stratum, we identify fundamental entities (application objects or classes). We identify security-relevant attributes those objects must have or are expected to have. We identify the security concerns the DOC paradigm is intended to address.
- Architectural: At this stratum, we describe the environment provided to application objects by the DOC middleware. We describe the basic middleware architecture; in particular, we consider which services are separable (i.e., potentially provided to the middleware via an interface) and which are inherent (i.e., required to be provided by the middleware). The technical perspective is that of the middleware developer; for some DOC paradigms, we must also take the perspective of the middleware user (i.e., application developer).
- Implementation: At this stratum, we identify specific products or standards that provide mechanisms for the DOC middleware. Depending on the architecture, the mechanisms may be inherent to or separable from the middleware, and the middleware may use one class of mechanisms to provide a given service or may support multiple mechanisms.

Considered more closely, each stratum can, of course, contain substrata, incomparable elements (boulders), and artifacts or fossils. Table 1 presents examples of questions that characterize DOC paradigms, systems, and products in terms of the three strata. Table 2 presents questions related to security concerns. Each stratum is viewed, and questions are posed, from three perspectives:

- Investment
- Technical
- Theoretical

The investment perspective focuses on the benefits each paradigm, system, or product offers (e.g., software reusability, scalability, interoperability with legacy applications). The technical (or practicioner) perspective focuses on how a DOC architecture is defined, what functionality a system or product offers, and how use can be made of existing products or technologies. The theoretical perspective focuses on what can be asserted, unambiguously and defensibly, about a DOC paradigm, architecture, system, or product. Thus, this perspective provides a foundation for assurance in DOC systems and products. In addition, it provides the foundation for secure interoperability among DOC paradigms, by ensuring a clear understanding of what security means, what security policies can be enforced, and how policy enforcement mechanisms actually work.

Table 1. Multiple Strata and Perspectives:
Characterizing Distributed Object Computing Paradigms

Stratum	Investment Perspective	Technical (Practicioner) Perspective	Theoretical (Rigorous) Perspective
Conceptual (Model)	· What goals is this paradigm intended to meet? What is the implicit cost model? · What assumptions about the operational environment or surrounding architecture are made? · How are those goals and assumptions reflected in the system and object models? · With which standards or frameworks is the system model consistent?	· What is the system model? · What is the (informal) object model?	· What is the underlying object theory? · How can the object model in this paradigm be expressed as a realization of the theory?
Architectural (Specification)	· With which system architecture(s) is the DOC paradigm compatible? · What assumptions about services and mechanisms provided in surrounding architecture are made? · What (services and mechanisms) must be where (in a notional decomposition of the system)? · In what terms (e.g., processes, transactions) is system behavior best described?	· What services are provided by the DOC middleware? Which are required and which are optional? · What are the interface specifications? Which interfaces are required and which are optional? · What off-the-shelf services and mechanisms are available to provide the specified functionality? How does that fit into the system architecture?	· What modeling constructs (e.g., specific types of objects or attributes) must be introduced for the interface specifications to be meaningful and realizable? · How is the DOC middleware specified (e.g., syntactically, functionally, using behavioral semantics)? · How are the underlying services and resources specified (e.g., in terms of middleware interfaces, by reference to standards)?
Implement-ation (Product or System)	· What's required of the system operating environment - functionally, procedurally, and in terms of products? · With which standards does the product or system comply?	· What products or components are integrated into the product or system? · What systems, products, or components interface with the product or system? · Which features of those products are used? · What are the configuration requirements or constraints?	· How do the semantics of the implemented interfaces correspond to the specifications? · How can we determine whether the implementation actually meets the specifications?

Table 2. Multiple Strata and Perspectives:
Characterizing Security in Distributed Object Computing Paradigms

Stratum	Investment Perspective	Technical (Practicioner) Perspective	Theoretical (Rigorous) Perspective
Conceptual (Model)	· How can security concerns be translated into statements of policy or requirements?	· How do the usual security policy objectives apply to this paradigm? · What security concerns arise in this paradigm? · What object attributes are security-relevant?	· How are security concerns expressed in terms of the underlying object model?
Architectural (Specification)	· What protections must be present in the operational environment or surrounding architecture? · What security-relevant capabilities are assumed for, allowed in, or precluded from applications?	· What security services are defined in the DOC paradigm? Which are required and which are optional? · What is the correspondence between services, interfaces, and security requirements or policies? For a given policy, which services and interfaces are required to enforce it, and for a given interface, which services does it support and which security policies is it intended to help enforce?	· How can security requirements be expressed unambiguously in terms of services, modeling constructs, and specified interfaces? In particular, for a given security requirement, which services, constructs, and interfaces must be implemented, and are there any constraints on the implementation? · How can the specified interfaces be expressed with sufficient rigor to ensure that they will enforce the security policies as claimed?
Implement-ation (Product or System)	· What security controls are required of the system operating environment - functionally and in terms of products? · What security controls are required of the physical environment? · What procedural and administrative controls are required?	· What security products or components are integrated into the product or system? · What security systems, products, or components interface with the product or system? · Which features of those products are used?	· How can we determine whether the implementation actually enforces the stated security requirements or policies?

Comparison and interoperability analysis at each stratum requires an understanding from all three perspectives. A number of comparisons have been made, typically at the architectural stratum, emphasizing the investment (DISA, 1998; Marcus, 1998; Tallman, 1998) or the technical (Chung et al, 1997; Csurgay, 1998) perspective.

2.1.1 Relationship to Other Frameworks

This characterization framework is intended to be more generic than RM-ODP. The RM-ODP viewpoints are intended to provide insight into *systems*, including operating environments and functional flows, while the characterization framework encompasses paradigms, architectures, and products. In RM-ODP, the term "object" must be modified by a viewpoint (e.g., enterprise object, computational object) to be meaningful.

The RM-ODP viewpoints, by providing languages, functions, and rules (or *transparency prescriptions*) are intended to enable the *specification* of distributed systems or components of such systems. Each view uses concepts presented in Part 2 of (ISO, 1996a), refines those concepts, and defines prescriptive rules and additional viewpoint-specific concepts. The ODP architecture is expressed by the computational language, the engineering language and the transparency prescriptions. The RM-ODP views correspond to the architectural stratum in the characterization framework, from the technical perspective. Depending on the detail with which they are used to specify a system, the RM-ODP viewpoints may also correspond to the top portions of the implementation stratum. The enterprise view overlaps both the investment and technical perspectives.

2.2 OBJECT-RELATED CONCEPTS IN DIFFERENT DOC PARADIGMS

This subsection provides an overview comparison of how key concepts are used, interpreted, or refined in the three representative paradigms. In terms of the comparison framework described above, it assumes a technical perspective at the conceptual stratum.

RM-ODP provides a paradigm-independent set of object-related terms. These terms are used differently (and often in multiple ways) in the different DOC paradigms.

The following "modeling concepts" are taken from (ISO, 1996a):

- *Object:* A model of an entity, characterized by its behavior and its state, distinct from any other object. An object is *encapsulated*, i.e., any change in its state can only occur as a result of an internal action or as a result of an *interaction* with its *environment*, which can occur only at one of its interaction points. An object may be informally said to *perform functions* and *offer services*.
- *Environment (of an object):* the part of the model which is not part of that object.
- *Action:* Something which happens. The set of actions associated with an object is partitioned into *internal actions* and *interactions*. An internal action always takes place without the participation of the environment of the object. An interaction takes place with the participation of the environment of the object.
- *Interface:* An abstraction of an object's behavior, consisting of a subset of the interactions of that object together with a set of constraints on when they may occur.
- *Activity:* A single-headed directed acyclic graph of actions, where occurrence of each action in the graph is made possible by the occurrence of all immediately preceding actions (i.e., by all adjacent actions which are closer to the head).
- *Behavior (of an object):* A collection of actions with a set of constraints on when they may occur.

- *State (of an object):* At a given instant in time, the condition of an object that determines the set of all sequences of actions in which the object can take part. State is typically represented by an assignment of values to a set of object attributes.
- *Communication:* The conveyance of information between two or more objects as a result of one or more interactions, possibly involving some intermediate objects.
- *Location in space:* An interval of arbitrary size in space at which an action can occur. Typically, location can be a hardware platform or subset thereof (e.g., a process space), or a communications network or subdivision thereof (e.g., node, link).

- *Location in time:* An interval of arbitrary size in time at which an action can occur.
- *Interaction point:* A location at which there exists a set of interfaces.

RM-ODP provides terminology for describing the structure of an activity. For our purposes, two of these terms suffice:

- *Chain (of actions):* A sequence of actions within an activity where, for each adjacent pair of actions, occurrence of the first action is necessary for the occurrence of the second action.
- *Thread:* A chain of actions, where at least one object participates in all the actions of the chain.

RM-ODP provides terminology for describing the roles an object may play in the course of an activity. For our purposes, two of these terms suffice:

- *Client object:* An object which requests that a function be performed by another object.
- *Server object:* An object which performs some function on behalf of a client object. Informally, a server provides a service requested by a client.

RM-ODP relies on Part 2 of (ISO, 1996b) to define the term *principal* in a manner different from other paradigms. Part 3 of (ISO, 1996b) defines the term *initiator*: an entity (e.g., human user or computer-based entity) that attempts to access other entities. This corresponds to the term *principal*, as used in CORBA.

Finally, RM-ODP, consistently with (ISO, 1996b), provides the following management concepts:

- *Domain:* A set of objects, each of which is related by a characterizing relationship to a controlling object.
- *Security domain:* A domain in which the members are obliged to follow a security policy established and administered by a security authority (i.e., an administrator). The security authority (or the security management object used by the administrator) is the controlling object for the security domain.

These concepts are applied or interpreted in the different DOC paradigms as indicated in Table 3. Note that the same term can be used in different ways within a single paradigm, and that uses or connotations in different paradigms can diverge significantly. In particular, a Java protection domain characterizes a set of principals *against* which uniform protection is needed; this contrasts sharply with the connotation in other paradigms, for which a domain characterizes a set of principals or objects *to* which uniform privileges are granted.

Table 3. Concepts and Terms in DOC Paradigms

Term/uses	**CORBA**	**COM**	**Java RMI**
Object	· A subclass of the CORBAobject class, with interfaces specified in IDL · An implementation or realization of such a subclass · The reference to a running instance of a realization of a CORBAobject	· A piece of compiled code with at least the IUnknown interface · A running instance of such code (running in a server)	· An object class definition · An object class, an implementation or realization of such a definition · A running instance of an object class
Initiating entity	An entity, representing a principal, that invokes a CORBA object	An entity, representing a principal, that invokes a COM object	
Principal	Principal: a human user or system entity that is registered in and authentic to the system. Has identity (may have multiple identities). May have privilege attributes.	Security ID (SID): a unique identifier for a user or group of users. May have privileges.	The source or signatory of an object class
Credentials	Credentials: a representation of the security attributes associated with a client	Token: a representation of the security attributes associated with a process or thread	Attributes of a stack frame (principal, privileges)
Client	The role played by an entity that makes a request	The role played by an entity that makes a request	· The role played by a platform to which an applet is downloaded (basic Java model) · The role played by an object that makes an RMI invocation
Server	The role played by an object that responds a request	· The role played by an object that responds a request · A piece of compiled code intended to provide an environment in which an object can run · A running instance of such code	· The role played by a platform (Web server) from which an applet is downloaded (basic Java model) · The role played by an object that responds to an RMI invocation
Security policy	One or more of the following:	One or more of the following:	· Safety (basic access control) policy

	· Access control policy · Audit policy · Secure invocation policies: · Authentication policy · Integrity policy · Confidentiality policy · Nonrepudiation policy	· Access control policies: · Activation security policy · Call security policy · Communications security policies: · Authentication policy · Integrity policy · Confidentiality policy	(enforced by byte-code verifier) · Access control policy
Domain	· Security policy domain: a set of object references to which a common security policy applies · Security technology domain · Security environment domain	NT domain: a collection of platforms with a common set of accounts and a shared authentication mechanism	Protection domain: a set of sources to which access to specific types of platform resources must be denied

2.3 HIGH-LEVEL COMPARISON

At the conceptual stratum, each DOC paradigm uses three models:

- A system model (or architectural framework), which identifies the fundamental building blocks of a DOC system.
- An object model, which identifies the determining characteristics of objects.
- A cost model, which assesses the relative costs of developing, maintaining, and using applications and systems. In general, cost models are implicit or informal, and are used to motivate the investment perspective.

From the investment perspective, the three paradigms can be compared conceptually as follows:

- Objective. CORBA is intended to facilitate the development and reuse of distributed applications, and the reuse of legacy applications in a distributed environment. COM+ is intended to facilitate the development and reuse of distributed applications. Java is intended to facilitate the development of portable and mobile applications.
- System model. For all three paradigms, the system model defines three levels: applications, middleware, and underlying services and resources (operating system, hardware, storage, communications media, networking). In CORBA, the applications are CORBA objects, and the middleware consists of an object request broker (ORB) and ORB services and facilities. In COM, the applications are COM objects, the middleware is an instance of the COM Library, and the underlying services and resources are conceptually those of Windows NT. In Java, the applications are objects written in the Java language and running in the confines of the Java Virtual Machine (JVM), the middleware is the JVM, and the underlying services and resources are those of the platform on which the JVM

runs. Because mobile code has the potential to damage platform resources, a major objective of the JVM is to isolate those resources from the applications.

- Object model. The CORBA and Java object models were derived from object-oriented programming models, and are realized in the CORBA Interface Definition Language (IDL) and Java language specifications respectively. The COM object model arose from object linking and embedding (OLE); a COM object is a piece of code intended to be used by multiple applications.
- Cost model. In CORBA, costs are associated not only with the development, maintenance, and use of CORBA-compliant applications and middleware, but also with the maintenance and use of legacy applications and with the need to support interoperability between technology domains and security domains. CORBA applications are therefore intended to span a wide range of size and complexity, from small special-purpose object-oriented applications to large legacy applications, possibly running on mainframes or dedicated servers, wrapped by software that makes them appear as CORBA objects. In COM, the focus is on development and maintenance costs, particularly for small units of software which perform functions common to many applications. In Java, the focus has been on specific development, maintenance, and use costs: the cost of developing the same application for multiple platforms and of ensuring current and consistent versions of application across a large number of user platforms.

3. CORBA

In this section, we first provide an investment perspective on CORBA. We next describe CORBA from a technical perspective, at the conceptual and architectural strata. Our presentation minimizes aspects of CORBA that do not bear on security, and highlights security considerations for interoperability. The CORBA architecture separates security services from other services (e.g., object life-cycle services) quite effectively. This enables our presentation to be quite generic and free from implementation details.

“CORBA” refers to an architecture, to a growing set of specifications for services and facilities within that architecture, and to the technologies used to implement the architecture or its components. It was developed under the auspices of, and continues to evolve and grow via the efforts of, a large consortium of organizations with disparate interests and technical biases. CORBA is thus intentionally flexible, extensible, and interpretable, sometimes to the point of being underspecified. Currently, considerable attention is being paid to its application to different usage domains, notably electronic commerce and medicine. That CORBA has not collapsed under the weight of the many interests it serves is a tribute to the Object Management Group (OMG) process (which emphasizes the importance of a reference implementation for each specification), the simplicity of the underlying Object Management Architecture (OMA), and the prime contributors to the CORBA specifications and supporting literature.

An increasing number of CORBASec-compliant products are available. These include IONA Technologies’ OrbixSecurity (IONA, 1997), PeerLogic’s offering of International Computers Limited’s (ICL’s) DAIS (PeerLogic, 1998), and Inprise’s VisiBroker (Inprise, 1998).

We therefore refine the conceptual framework to provide examples of questions intended to help determine whether a CORBA-based architecture, product, or system can meet security needs and/or be integrated with or interoperate with other products or systems.

The information presented in this section is derived primarily from documents available through the Object Management Group’s website (OMG, 1997a-c, 1998a-d). Additional sources include (Appelbaum et al, 1996; Chapin, 1997a; Mowbray, 1995; Mowbray et al., 1997).

3.1 CORBA MOTIVATION AND BACKGROUND

The motivating vision for CORBA is that of location-transparent plug-and-play software, based on object technology. An entity (typically a piece of software, but possibly an individual or system) can assume the role of *client* vis-à-vis a CORBA object, requesting the services provided by that object through its published interfaces, without needing to consider such implementation details as the language in which the object was written or the platform on which it runs. The CORBA object thus invoked can itself act as a client, invoking other CORBA objects. Chains of invocations can be conceptualized as *threads*.

This vision is set forth in architectural descriptions, specifications, and white papers published by the Object Management Group (OMG), an international consortium with over 800 member organizations. Its members include major vendors of systems and software from around the world, as well as independent software vendors, consulting companies, and a number of end-user companies. The OMG produces specifications; its members produce implementations of those specifications.

3.1.1 CORBA Object Model

The OMG CORBA object model is a model of object *interfaces*, not of object *implementations*. An interface is a set of definitions of constants, types, attributes, and operations. An operation is specified by an identifier (its name), the type of the return value of the operation, and a parameter list. This is a list of formal identifiers together with their types and modes. The type of a parameter specifies the values that can be passed as the parameter. The mode of a parameter specifies the direction in which the parameter is passed.

Values can be primitive (e.g., numbers, bytes, identifiers, booleans), compound (e.g., structures, unions, strings, sequences), or object references (i.e., references to object instances and not to object interfaces).

Types are sets of values with common structure. IDL defines basic, constructed, array, template, and interface types. Basic types include integers, floating point numbers, characters, booleans, bytes, strings, and tag types. Constructed types include structures, enumerated types, and discriminated unions. Interface types define the structure of object interfaces.

Attributes are encapsulated variables whose values can be accessed via read and write operations only. Thus attributes can be viewed as abbreviated specifications of read and write operations of particular types.

3.1.2 Architectural Overview

At the architectural stratum, the motivating vision is expressed by the Object Management Architecture (OMA). The fundamental components of this architecture are the CORBA Core or Object Request Broker (ORB), CORBA services, and CORBA facilities. Conceptually, the ORB provides an *object bus* which mediates communications (requests) between objects. The CORBA services provide basic functionality that would be required by most types of objects: object lifecycle services such as move and copy, naming and directory services, and other basics, notably security. CORBA facilities provide services for applications such as application level data manipulation and storage.

More specifically, the ORB provides the following services:

- Implementation repository services, which provide persistent storage for object implementations.

- Installation and activation services, which support the distribution and installation of objects.
- Interface repository services, which support a database of information about the interfaces supported by objects.
-

OMG is working to define additional services an ORB should provide, notably:

- Replication services, which manage replicated objects in a distributed environment.

CORBA services provide a set of supporting functionality to all objects. CORBA services include:

- Change management services, which manage the identification of different versions and configurations of objects.
- Collections services, which deal with sets of objects in various forms including lists, trees, stacks and queues.
- Concurrency control services, which coordinate access to shared resources.
- Data interchange services, which support exchange of information between objects.
- Event management services, which support management and delivery of events to objects.
- Externalization services, which define protocols for the transmission and receipt of object state information.
- Licensing services, which support licensing of objects.
- Life cycle services, which support object creation, deletion and maintenance.
- Naming services which provide both naming and lookup services for objects.
- Persistent object services, which provide ways to capture, store and manage the state of objects.
- Properties services, which support dynamic object properties.
- Query services, which provide ways to identify subsets of collections of objects.
- Relationship services, which allow relationships between objects to be represented as new objects.
- Security services (described in more detail below).
- Startup services, which support automatic startup and shutdown of services when the ORB starts or stops.
- Time services, which provide synchronization for distributed systems.
- Trading services, which support location of objects by the services that they provide, rather than by name.
- Transaction services, which provide atomic transaction facilities for groups of operations on objects.

Within the limits defined by the corresponding specification, these services may be provided by the ORB or by a third-party product. The number and variety of these services can make CORBA as a whole hard to understand. CORBA specifications are written to minimize interdependencies. However, an assessment of the security provided by a CORBA system (ORB, services, and facilities) must address the dependencies of security services on other supporting functionality. In particular, system security depends on time, startup, naming, life cycle, and data interchange services.

3.1.3 Security Services in the Architecture

CORBA specifies three ways for an ORB to provide security services to applications. First, security services can be tightly integrated into an ORB. Second, an ORB can support *security replaceability*, by enabling the transparent substitution of platform security services for those it provides. Third, it can be *security ready*, providing no security functionality itself, but providing interfaces for third-party security services. A security-ready or security-replaceable ORB applies security controls to requests using an *interceptor* mechanism. That is, the ORB intercepts the request as it mediates the flow of communications between the client and the target, and invokes security services.

The CORBA Security Specification defines two levels of security services. At Level 1, the ORB provides security to security-unaware applications. At Level 2, the ORB supports *security-aware* applications. That is, an application object can invoke the CORBA security services directly, through their specified interfaces.

CORBASec provides multiple options for secure interoperability between ORBs, including the Secure Inter-ORB Protocol (SECIOP), the Distributed Computing Environment Common Inter-ORB Protocol (DCE-CIOP), and Secure Sockets Layer (SSL). When two CORBASec-compliant ORBs use the same interoperability protocol, enforce consistent security policies, and use the same security mechanisms, they are expected to be able to interoperate securely.

3.2 TECHNICAL PERSPECTIVE ON CORBA AT THE CONCEPTUAL STRATUM

At the most abstract level, we simply have objects calling (or invoking) objects, with all architectural and implementation details concealed. More precisely, an object—a user, process, or CORBA object—assumes the role of *client*, *requesting* the services of a *target* (a CORBA object) via one of that object's published *interfaces*. The client typically expects a *response* from its target. Conceptually, the response can be viewed as a request for one of the original object's interfaces; that is, the target object turns around and assumes the role of client.

In a chain of requests, the first object to assume the role of client is the *initiator*. The initiator is an object in the most general sense; all other objects in the chain of requests are CORBA objects. There are two distinguishing characteristics of CORBA objects. First, they are instances of the CORBA object class. They therefore inherit the interfaces of that class, so that an ORB can interact with them. Second, they are described using the CORBA IDL (interface definition language). (Subsection 3.2.1 provides more detail on CORBA objects and IDL.) Interfaces are specified in terms of input parameters and return values, for which meanings are usually informally described; the range of values is usually extensible. The intended behavior of an object is described informally, rather than clearly specified. This provides implementors a wide degree of flexibility, at the cost of potential barriers to interoperability.

An object in CORBA is a very general construct. At a minimum, it includes the code that provides the services offered by its interfaces. It may also include the logical or even physical resources needed to provide those services. For example, a database can be a CORBA object, with interfaces provided by a database application and/or specialized from those provided by a database management system. A device such as a printer can be a CORBA object.

The OMA model, and CORBA as an architecture, are so constructed as to allow us to blur the different possible interpretations of the term "object" in discussing security. We need not distinguish between an abstract object (an IDL specification), a realization of that specification in code, a specific implementation (combining code, data, and perhaps even physical resources) stored on a given platform, or that implementation running (encapsulated in a server process) on the platform.

At this level of abstraction, there are simply objects calling objects calling objects. At this level, CORBA allows us to address four major security concerns: What restrictions should apply on which objects can call which other objects? How should communications between objects be secured? What events should be audited, and what information about those events should be recorded? How can we ensure that actions cannot be disavowed by the individuals responsible for them?

The first concern is restated in terms of a traditional computer security concern: On what basis should access to objects be controlled? To address this, the concepts of *credentials* and *required rights* are introduced. A client has an associated set of credentials, which represents the *principal* (e.g., individual user, system service) on whose behalf the client is currently acting, and the *privileges* (e.g., groups, user rights) currently associated with the client. The target the client seeks to use has a set of required rights. For each interface, a set of principals and/or privileges is specified.

Concerns related to *delegation* arise in this context. When an initiating client requests that a target perform some function, the target may have to act as a client towards another object. On whose behalf is it then acting? This question can be asked at each link in a chain of invocations. An object could act on its own behalf, using credentials specific to itself. It could act on behalf of the initiator's principal, inheriting the initiator's credentials and thereby *impersonate* the initiator. (Impersonation is also known as *simple delegation*.) It could also acquire a set of credentials computed from itself, the initiator, and/or some subset of the objects invoked earlier in the chain.

The second concern involves *associations* between objects. It may be security policy for a client not to make certain requests unless it can be assured of a certain level of integrity and/or confidentiality for its communications with the target. The client may therefore need to *negotiate* a secure association with the target via a series of policy-related requests.

CORBA supports auditing of requests based on the attributes of the client (e.g., audit all activities related to a specific user) and/or the target. To enable auditing, access control, and establishment of secure associations, CORBA security services must support I&A of principals and link credentials to objects and the requests those objects make.

3.2.1 CORBA Object Model

This subsection provides more detail on the CORBA object model. In terms of the characterization framework, this material is at the lower end of conceptual stratum, bordering on the architectural. This material is intended for readers who need either a deeper understanding or a firmer basis for comparing CORBA with other DOC paradigms.

The OMG CORBA object model is a model of object *interfaces*, not of object *implementations*. Consequently, the model specifies the syntax and semantics of object interfaces, but it does not specify the syntax or semantics of object implementations. The behavioral semantics of object implementations must be specified by the object implementors and is considered to be outside the scope of CORBA.

In this subsection, we briefly summarize the syntax and semantics of object interfaces as defined by OMG’s Interface Definition Language (IDL). An interface is a set of definitions of constants, types, attributes, and operations. We first describe the values of the language. Then we describe types, constants, attributes and operations. Finally, we describe interfaces and modules.

3.2.1.1 Values

The primitive values include single and double precision integers, single and double precision floating point numbers, bytes, identifiers, and booleans. Compound values include tagged values (e.g., <"a1", 0>), structures, unions, strings, sequences, and arrays. Finally, object references are also values.

Note that object references are references to object instances and not to object interfaces. Also note that object instances and interfaces are not values (the call-by-value extension to CORBA adds object instances as values).

3.2.1.2 Types

Types are sets of values with common structure.

Basic types include the integer types, the floating point types, characters, booleans, bytes, strings, and tag types. The special type `any` can stand for the type of any value.

Constructed types include structures, enumerated types, and discriminated unions. Array types consist of multidimensional fixed-size homogeneous arrays. Template types include bounded and unbounded sequences and strings. Values of template types have a current length and a list value.

Interface types define the structure of object interfaces. Values of interface types are object references.

3.2.1.3 Constants

The value of a constant cannot change. Constants can be of type `char, boolean, float, double, string` or any integer type. Constants can be defined to be value literals or simple expressions constructed from value literals.

3.2.1.4 Operations

[`oneway`] <type> <identifier> (<parameter-list>) `raises` <exception-list> `context` <context>

<identifier> is the name of the operation. <type> is the type of the return value of the operation. <parameter-list> is a list of formal identifiers together with their types and modes. The type of a parameter specifies the values that can be passed as the parameter. The mode of a parameter specifies the direction in which the parameter is passed:

- An **`in`** parameter is passed from the caller to the object. The parameter identifier is initialized to the value passed by the caller before performing the operation. If the operation modifies the value of this parameter, the modified value is not returned to the caller.

- An **`out`** parameter is passed from the object to the caller. The parameter identifier is not initialized before performing the operation. The operation is required to assign a value to such a parameter; the final value of this parameter is returned to the caller. Note that the order in which parameters are returned to the caller is not specified.

- An **`inout`** parameter is passed in both directions. The parameter identifier is initialized to the value passed by the caller before performing the operation. When the operation returns, the final value of this parameter is returned to the caller. As with **`out`** parameters, the order in which parameters are returned to the caller is not specified.

The parameter passing style is call-by-value. That is, the values which are to be passed as parameters or results of an operation are first copied and the copies are then passed. The copying operation is not guaranteed to preserve the graph structure of values. Consider a structure value with two components that are the same sequence value. Copying the structure value will result in a new structure value; however, in this new structure, the two components could either share the same copy of the original sequence value, or they could contain two distinct copies of the original sequence value.

`oneway` is an optional qualifier. If it is absent, the invocation semantics is at-most-once if an exception is raised and exactly-once if the operation invocation returns successfully. If it is present, the invocation semantics is at-most-once. This is best-effort semantics where the ORB must make a best effort to complete the invocation, but no guarantee is provided as to whether the operation is invoked and completed. Oneway operations must have a `void` return type, no **`out`** parameters, and no exceptions.

An operation invocation can be *blocking* or *nonblocking*. With blocking semantics, the client that invoked the operation blocks until the operation is complete and the return value is returned. With nonblocking semantics, the client that invoked the operation continues executing past the call and gets no indication whether the operation executed successfully. Note that the presence or absence of the `oneway` qualifier does not imply blocking or nonblocking semantics; whether an operation invocation is blocking or non-blocking depends on the language binding being used.

The exception list consists of a list of identifiers of exception types.

The context is a list of name-value pairs. Its use is discouraged.

3.2.1.5 Attributes

Attributes are encapsulated variables whose values can be accessed via read and write operations only. Thus attributes can be viewed as abbreviated specifications of read and write operations of particular types.

An attribute definition is of the form: [**`readonly`**] **`attribute`** <type> <identifier>

It maps to two operations:

<type> <identifier> ()	This operation returns the value of the attribute.
`void` <identifier> (<type>)	This operation sets the value of the attribute to the argument value.

An attribute can have a **`readonly`** qualifier, in which case it only maps to the first of the above two operations.

3.2.1.6 Interfaces

An interface is a set of constant, type, attribute, and operation definitions.

Some points of note:

- An interface can inherit from other interfaces (multiple inheritance of interfaces is allowed). If an interface A inherits from interface B, then B is called a base interface of A, and A is called a derived interface of B. In this case, the interface

A includes all the constants, types, attributes, and operations of interface B (and of all other base interfaces of A, if any).

- Overloading of attribute or operation names is illegal. Constant, type, or exception names can be overloaded provided they have unique fully-qualified names (i.e., provided they are defined in different interfaces which are inherited).

- All definitions in an interface are public; the definitions can thus be inherited by any derived interfaces and used by any client objects.

- Member (instance) variables cannot be declared; rather, the variables must be declared as attributes and manipulated via the read and write accessor functions of the attributes.

3.2.1.7 Modules

Interfaces can be collected and packaged into modules. A module defines a naming scope (e.g., module1::module2::interface). Modules have hierarchical structure.

3.3 TECHNICAL PERSPECTIVE ON CORBA AT THE ARCHITECTURAL STRATUM

At its most simple, a CORBA-compliant system consists of a collection of application objects, an ORB which provides services and facilities to those objects, and the underlying OS and network which provide functionality used by the ORB, its services, and facilities. The services and facilities may be part of the ORB, or the ORB may provide interfaces to them and they may be supplied by third-party vendors. In the following subsections, we first provide background on the various security-related meanings of the term “domain.” We then provide an overview of the CORBA threat-mitigation model defined in (Williams et al, 1996). This model shows how the CORBA security services can be used (or extended) to counter threats that arise in the DOC environment. In the appendix, we identify the security services, and indicate security-relevant attributes used by these services.

3.3.1 Security Domains in CORBA

In an architectural context, CORBAsec defines three types of security-relevant domains: a security policy domain, a security technology domain, and a security environment domain. CORBA defines many different types of domains, most of which have no security relevance. Examples which can have security management implications include ORB technology domains and naming domains.

A CORBA *security policy domain* is a set of objects to which a common security policy applies. Conceptually, a security policy is a set of requirements on access control, delegation, audit, nonrepudiation, or message protection. More specifically, a CORBA security policy consists of required conditions to be enforced by, or actions to be taken by, the CORBA security services. A given object can belong to multiple domains. This can occur in several ways: Distinct policies (e.g., access control *vice* audit) can be defined for different sets of objects. The object can fall in the intersection of two domains for the same type of policy, such as access control based on role and based on identity. Domains can be hierarchically ordered, and the object can fall into a subdomain of a subdomain. An implementation of the CORBA security services must include a strategy for resolving differences in policy for an object in multiple domains. Typically, the intersection, or most stringent application, of the multiple policy requirements on an object is used.

Interoperation between security policy domains requires the definition of interdomain security policies. When the domains are under the control of a single ORB, those policies could be enforced by ORB security services. (However, the CORBASec Specification does not currently define an Interdomain Service.) More generally, interdomain security policy enforcement is expected to be provided by a Secure Interoperability Bridge. Conceptually, a Secure Interoperability Bridge is similar to a trusted process in a conventional multilevel secure system.

A *security technology domain* is defined by the use of a single security technology to enforce a security policy. It thus may include one or more ORBs, the objects managed by those ORBs, and (if the ORBs are in different ORB technology domains) interoperability bridges between the ORBs. CORBASec does not specify architectural details for a security technology domain. Thus, the ORBs could use the same (replaceable) security services or could implement the services independently, while relying on the same underlying technology.

Interoperation between security technology domains requires either that no security is required on communications between them, or that a security technology gateway mediates their interactions. CORBASec does not define gateway services; such services are expected to be technology-specific (and possibly policy-specific as well). Because different policies (e.g., authentication, communications protection) are enforced by different technologies, an ORB could fall into multiple security technology domains. Thus, it could need multiple gateways to interoperate with other ORBs.

Security policies are enforced by mechanisms and controls which have an inherent scope. A *security environment domain* is the scope of any security policy enforcement mechanisms that are “local to the environment.” “Local” can have several meanings, including physical (e.g., local to a hardware platform), logical (e.g., “identity domains”), technological (e.g., using the same technology for message protection), and administrative. Security environment domains are not visible to application objects and security services, and are implementation-specific.

3.3.2 The CORBA Threat-Mitigation Model

(Williams et al, 1996) defines a model of how the CORBA security services can be used (or extended) to counter threats that arise in the DOC environment. The model provides an approach to event mediation that integrates access control with audit, non-repudiation, and other CORBA security services. Event mediation involves deciding which messages (requests and responses) are acceptable from a security perspective. It also involves managing the granularity of references to targets and principals. In particular, it addresses the operational need to group sets of principals and object references.

3.4 CHARACTERIZING CORBA-COMPLIANT ARCHITECTURES, SYSTEMS, AND PRODUCTS

In the following subsections, we provide questions that can be used to characterize CORBA-compliant architectures, systems, and products from the standpoint of security. These questions can be used as a starting point for assessing whether and how well a CORBA system meets the security requirements of a given program, can be integrated into a given architecture, or can interoperate with a given system.

3.4.1 Conceptual Stratum

At the conceptual stratum, the CORBA paradigm is expressed in the OMG Object Model, the Object Management Architecture, and the models in the CORBA and CORBAsec Specifications. Product developers interpret and refine this paradigm differently, to meet the needs of specific markets and to provide differentiating value-added. From an investment perspective, questions include:

- What, if anything, differentiates your interpretation of the central concepts (objects, threads, and principals) from that of any other CORBA-compliant system?
- What additional goals is your system intended to meet? For example, is it intended to support component reuse/composability? Is it intended to support mobile code?
- Is your system intended to support a specific application domain (e.g., medical)? If so, how does this affect the security policies it supports and the security services it provides?
- With which other DOC paradigms is your system intended to interoperate (e.g., DCOM)? How, if at all, have your interoperability goals influenced your interpretation of the central concepts?

From the technical perspective, the conceptual stratum is expressed in the Introduction and Security Reference Model sections of the CORBAsec Specification. The Introduction briefly describes how conventional security policy concerns (threats and types of security functionality) apply to distributed object environments. At this stratum, the fundamental terms for discussing system behavior are objects, requests, principals, and threads. Security concerns can be expressed in terms of relations between them: client/target, authorization, delegation, impersonation. Security policies can be expressed by associating specific types of attributes and properties with these constructs, and by stating constraints on the results of requests (or requirements on the use of security functionality, e.g., to secure communications) based on the values of such attributes or properties. Questions include:

- What credentials (security-relevant attributes) does a principal have (e.g., unique identity, membership in groups or authority to assume roles, current group or role, security level or authorization for types of data, privileges)? Which credentials are required, and which are optional? Which must be authenticated, and which are public?
- What privilege delegation options are available?
- Will nonrepudiation be supported?
- For an object that belongs to multiple security policy domains, how are differences between security policies resolved?

3.4.2 Architectural Stratum

At the architectural stratum, we consider the components of a product, product line, or system, their interrelationships, and their relationships to other information technology components (e.g., operating systems, communications). From an investment perspective, questions beyond those in Table 1 include:

- How does the architecture, system, or product fit into specific enterprise, proprietary, or open systems architectures?
- What security replaceability option is supported?

The CORBAsec Specification provides two frames for the technical perspective: the structural model and the set of Security Services that form the Security Replaceability conformance option. The structural model defines four levels; security functionality and thus responsibility for security policy enforcement can be allocated to the application level, to components that implement the Security Services, to components that implement specific security technology, or to the basic protection and communications level. In terms of this model, the following questions can aid in determining whether an architecture (typically realized in a product or system) can meet security needs and/or be integrated with or interoperate with other architectures, products, or systems:

- What does it mean for an application to be security-aware? Aware of which properties and attributes? Aware of which security policies?
- In what ways, if any, are application-level components required to be security-aware? In what ways are they expected to be security-aware, and what compensatory actions (involving use of security functionality at another level) are necessary if they are not?
- Is the complete set of Security Services provided? If not, are specific services missing, or are services implemented incompletely? For missing services or interfaces, what is assumed or required of the technology and basic levels to compensate?
- What security functionality is provided by the technology level? What security technologies are required in the environment (e.g., from third-party vendors)?
- At the basic level, how are Protection Domains provided? What is required or expected in terms of physical, temporal, and logical isolation mechanisms? What

compensatory actions (e.g., location or configuration of Security Services) are necessary if those mechanisms are not provided by the environment?

The Security Services that form the Security Service Replaceability conformance option are authentication, access control, audit, non-repudiation, and secure invocation. For each service, questions include:

- What mechanisms (if any) does the ORB include to provide this service?
- What mechanisms (if any) at the technology or basic level does the ORB use to provide this service? Are those mechanisms provided by a specific (proprietary) product, or by any product that conforms to a given standard? Which one, and are there any restrictions on how it is used?

3.4.3 Implementation Stratum

From an investment perspective, the focus is on whether and how a product or system can be used effectively in different environments. Questions include:

- What other products must acquired (e.g., third-party security services)?
- What other products are desirable (e.g., for audit data reduction and analysis)?
- What physical controls are required of the hardware on which the product resides? Of the media on which it stores data? Of communications media?
- How is security administration of the product or system related to that of other products at the technology or basic level? For example, is there a common administrator interface?

At this stratum, implementations not specific to CORBA commonly provide the capabilities allocated to the technology or basic level. These implementations are based on security architectures for distributed systems. From the technical perspective, questions include:

- Does the product or system depend on a variant of Kerberos or on an X.509-based public key infrastructure (PKI)? If so, which one?
- How are attributes associated with a principal? Are multiple mechanisms used to associate attributes with a principal? (For example, does a principal's credential include unauthenticated attributes?) Where (at which level) is the functionality that associates a given attribute with a principal provided? Are some attributes defined implicitly or computed? If so, how?
- For each attribute, what is the range of allowable values? How and where is range-of-value checking done? What is the default value, and when is it established?
- Under what circumstances can each possible form of privilege delegation be used?

4. COM, DCOM, AND COM+

In this section, we describe the motivation for Microsoft's Component Object Model (COM) and its offspring, Distributed COM (DCOM) and COM+. (For brevity, we will use the term "COM" to cover all DCOM and COM+ as well, noting differences only when relevant to security.) We next describe COM from a technical perspective, at the conceptual and architectural strata. We seek to minimize aspects of COM that do not bear on security, and to highlight security considerations for interoperability. However, the COM architecture does not cleanly separate security-relevant attributes from attributes related to other services. Thus, our presentation will include details that were not necessary in our discussion of CORBA.

The information presented in this section is derived primarily from documentation at the Microsoft Software Developer Network (Microsoft, 1995, 1997, 1998a-b; Goswell, 1995; Robinson, 1997; Williams, 1994). Additional sources include (Chapin, 1997a-b; Chappell, 1996; Gonsalves, 1998; Grimes, 1997; Moeller, 1998; Petersen, 1998). Our conceptual and architectural presentations must be understood as an abstraction and extrapolation of the information in these sources. While "COM" refers to an architecture and a specification as well as an implementation, the architecture and specification are moving targets; the Windows NT reference implementation is the definitive source. In particular, Microsoft does not plan on specifying COM+ for platforms other than Windows NT 5.0 (InfoWorld, 1998).

From a technical perspective, COM can be understood at several levels of abstraction. The COM Specification does not cleanly distinguish among these levels, leading to potential confusion. At the conceptual stratum, COM can be discussed in terms of clients, objects, servers, and the COM Library. This level provides a starting point for understanding COM and allows discussion of a few basic security issues. However, because COM is grounded in its reference implementation, any description soon must include some implementation details.

At the architectural stratum, host location, server "flavors," and authentication services must be considered. This is the level at which a COM/DCOM security model can be presented. The architectural stratum can be refined to add a variety of additional terms; these include storage objects, class factory objects, handlers or proxy objects, and initialization and uninitialization of the COM Library. At this level, security issues common to COM implementations can be discussed.

4.1 COM MOTIVATION AND BACKGROUND

The current motivation for COM is that of location-transparent plug-and-play software, running in an environment essentially equivalent to Windows NT. Microsoft has provided the COM specification to The Open Group as a proposed open standard, and third-party vendors are implementing COM on non-Microsoft operating systems. However, the Windows NT reference implementation exerts a powerful influence on the COM specification.

The original Component Object Model expressed Microsoft's strategy for enabling different software components running on a single platform to provide services to one another. Conceptually, COM derived first from Microsoft's object linking and embedding (OLE) technology, then from object-oriented programming concepts, and finally from the Windows NT architecture.

Distributed COM, or DCOM, extended COM to multiple hardware platforms. COM+ is Microsoft's planned COM execution environment. COM+ will provide supporting features such as automatic reference counting (eliminating burdens on both clients and objects) and interceptors. Currently, developers of COM-compliant software must build such functionality themselves. In effect, COM+ will incorporate some of the functionality currently provided by the Microsoft Transaction Server (MTS). COM+ is therefore expected to be more CORBA-like. Specifically, COM+ services will include:

- Event Services, to create a publish and subscribe model for components
- Load Balancing, to enable dynamic routing of component processing
- Security, using a role-based security model consistent with Windows NT
- Queued Request Services, to provide asynchronous queuing services
- In-Memory Database, to improve performance for accessing components

4.1.1 Architectural Overview

The fundamental components at the architectural stratum are the application objects, the COM Library (referred to simply as COM), and the underlying OS and network.
COM provides the following services:

- Object creation, activation, and management.
- Persistent storage, used by objects to save their state for restoration later or to save and share data. To enable persistent storage of data, COM provides "a file system within a file," relying on OS services to manage the containing file.
- Persistent, intelligent names (monikers). These enable a client to connect to a specific instance of an object (rather than to just any instantiation of that object) and/or to perform a specific operation. These correspond roughly to CORBA persistent interfaces.
- Uniform data transfer standards to facilitate communications between clients and objects.
- Security services (described below).

COM+ is expected to incorporate many MTS services, thus becoming more CORBA-like.

4.1.2 Security Services in the Architecture

COM provides one security service directly: access control. It provides two forms of access control: activation security and call security. COM relies on the underlying OS to identify and authenticate principals and to bind a principal’s identity and the current set of privileges to a client. COM does not provide security audit services; object implementors can make use of OS logs, but this does not involve COM.

Via its Security Support Provider Interface (SSPI), DCOM uses functionality provided by a security subsystem to protect communications between objects on different platforms. The security subsystem is external to COM. Currently, it is provided by Windows NT (NTLMSSP). In NT 5.0, the default will be the Kerberos security system, but other security subsystems will be supported: NTLMSSP, Distributed Password Authentication (DPA), and SSL3/TLS.

The COM specification does not currently address secure interoperability between different vendor’s implementations of the COM Library.

4.2 TECHNICAL PERSPECTIVE ON COM AT THE CONCEPTUAL STRATUM

As noted above, COM arose from a series of implementations; its continued evolution is strongly influenced by the emerging Windows NT reference implementation of COM+. Thus, the description at the architectural stratum has been abstracted and extrapolated from COM documentation. The description at the conceptual stratum has entailed further abstraction, extrapolation, and application of the emerging distributed objects model. It must be understood as an expository device, and not as inherent to Microsoft's strategy for defining and refining COM.

At the highest level of abstraction, the COM object model consists of two constructs: *objects* and *messages*. An object represents binary executable code and has an associated set of interfaces. One object requests that a second object take some action by sending a message to one of the second object’s interfaces.

At this level of abstraction, there is simply code calling code calling code. Two security concerns are addressed by COM: What restrictions should apply on what code can call what other code? How should communications between pieces of code be secured? (COM does not address audit concerns, leaving these to applications and the underlying OS.)

Two aspects of the first concern can be identified: authorization and integrity. Authorization involves the question: Who is allowed to use a piece of code? To address this, the concepts of *token* and *permissions* are introduced. An object acting as a client has an associated token, which represents the *principal* (e.g., individual user, system service) on whose behalf the object is currently acting, and the *privileges* (e.g., groups, user rights) currently associated with the object. The object the client seeks to use has a set of permissions. COM identifies two types of permissions: permission to use a specific interface and permission to use an object at all. The corresponding security policies are referred to as *call security* and *activation security*, respectively.

Integrity involves the question: In what order should objects execute to ensure correct operation and accurate data? The concept of *threads* is added to the COM object model to address this. However, the COM Specification does not treat threads as security-relevant.

To address concerns for communications between objects, the concept of object location is needed. This is represented at the architectural stratum.

At a less abstract level, the COM paradigm can be described in terms of clients, objects, servers, and the COM middleware. A client is a running piece of software that meets two conditions. First, it knows about and needs the services provided by one or more objects. Second, it meets the minimum requirement to be a COM client: it includes a mechanism to release an object when it is done with that object's services. A client has an associated principal, i.e., an entity on whose behalf the client is running. A principal can be an individual, a process, or an abstract entity (e.g., "system"). A client also has an associated set of privileges, derived from the identity of its principal.

The term "object" has several meanings. First, it refers to "a piece of compiled code that provides some service to the rest of the system" (Williams, 1994). This is also known as an object class, since multiple instances of it can run simultaneously. Second, it refers to an executing instance of that piece of code. These COM-specific meanings must not be confused with other meanings. In particular, many object-oriented concepts do not apply; a Windows NT object is typically not a COM object, and *vice versa*.

The term "server" is similarly overloaded. First, it refers to a software module that can provides service to objects. Those services include memory management. A server software module is either *security-aware* or *security-unaware*; security awareness is relevant to object activation. A server software module has an associated set of object classes it can serve. Second, the term refers to an executing instance of that software. A server instance encapsulates one or more objects, and has an associated principal.

The term "COM" refers first to the COM Specification and its underlying paradigm. It refers to the COM Library in the abstract: a set of services provided to clients and servers. It refers to the various operating system-specific implementations of this set of services. Finally, it refers to an instance of the COM Library executing on a specific host.

4.3 TECHNICAL PERSPECTIVE ON COM AT THE ARCHITECTURAL STRATUM

At its most simple, a COM-compliant system consists of a collection of application objects, a running implementation of the COM Library which provides services and facilities to those objects, and the underlying OS and network which provide functionality used by COM, its services, and facilities. COM relies largely on services and facilities provided by the underlying OS, particularly the Service Control Manager (SCM). In the following subsections, we identify the security services, describe how they appear in action, and identify security-relevant attributes and properties that are used by the security services.

4.3.1 Security Services

An instance of the COM Library provides an operating environment for COM-compliant clients and objects. According to the COM specification, that environment is expected to maintain the sets of Inanimate Objects and Animate Objects. It is also expected to provide access (via the file system of the underlying operating system) to persistent storage.

Security policy enforcement actions can be taken by

- COM
- A server instance
- An object instance

Policy decisions take into consideration the principal associated with the client. Thus, COM must provide access to a minimum set of identification and authentication services. (These are expected to be provided by the underlying operating system, but could be provided by a security plug-in to COM.) In addition, a client might enforce a limited security policy by deciding whether to use a server or object instance based on the identity of the object's author or source. To support this, an implementation of COM must provide access to a mutual authentication mechanism.

COM makes a fundamental distinction between security-aware and security-unaware servers, which applies to object activation. If an object instance exists and a client requests of COM one of that object's interfaces, COM must decide whether to create a new object instance or to provide the client with an interface pointer to the existing object instance. If the server is security-unaware, COM creates a new server instance and object instance; COM sets the server instance's principal to be the client's principal. If the server is security-aware, it is its own principal; COM provides the client with an interface pointer to the existing object instance and expects the server instance to perform any security decisions related to the client's use of that interface.

COM is expected to enforce an activation security policy, which restricts client activation of object classes and connection to server instances based on the client's principal. The COM Specification suggests a minimum set of options for expressing an activation security policy. A server instance or an object instance can enforce a call security policy, which restricts client use of interfaces (and the specific functions or methods provided by an interface) based on the client's principal. COM is expected to provide an API to support such decisions, by providing I&A services. (The phrase "expected to" is used because some implementations do not provide mechanisms.)

4.3.2 Processing Overview

The following description of the life-cycle of an object assumes a homogeneous environment. That is, while the environment may include multiple hosts, running different operating systems, all instances of the COM Library are compatible and provide the same services, including the DCOM services that make object location transparent. Thus, we can speak of COM rather than distinguishing among COM instances.

Object creation: This is an infrequent activity, representing the installation of new compiled code on a host. A client requests that COM add an object class to its Known Objects. The request identifies the object class, its interfaces, and its server. COM adds the object class to its Inanimate Objects.

Object activation: A client requests of COM use of an interface. COM checks its Animate Objects to determine whether an object instance exists that the client may use. If not, COM determines the server class the object requires, creates an instance of that server, and the server instance creates an object instance. If the server is *in-process*, COM activates the server within the client's process space. If the server is *out-of-process*, COM creates a new process and activates the server within that process space. (A security-aware server is always out-of-process.) COM provides the client with the interface pointer to the object instance which corresponds to the requested interface. COM establishes the principal for the server instance, based on the server's security-awareness.

When the object resides on a host different from the client, the client may explicitly specify the location of the remote host. Otherwise, the COM instance on the client host identifies the object's location from its Inanimate Objects. It creates a proxy server within the client's process space. It passes the request for object activation to the COM instance on the object's host. That COM instance creates a new process and activates the object's server within that process space.

Object use: The client establishes a session with the object instance, via the interface to which it has a pointer. The client uses the object services offered by that interface. The client can request (of the object instance) the use of additional interfaces. If the object instance makes an additional interface available to the client, the client establishes a new session with the object instance. Thus, the client can have multiple sessions with the object instance. Note that once an object instance exists, it can act as a client for other objects. Thus, complex relationships among object instances are possible.

If the server is security-aware, it relies on COM's authentication services to identify the principal associated with the calling client. It then can restrict the client's use of interfaces in accordance with the security policy it enforces. For example, it can restrict the range of input parameter values it will accept, based on the requesting principal. Similarly, a security-aware client can rely on COM's authentication services to identify the principal associated with a response from an object in an out-of-process server, and can then determine how to handle that response.

Object deactivation: The client terminates its session(s) with the object instance. When the object instance determines that its deactivation conditions have occurred, it deactivates (or uninitializes itself). For many objects, a single deactivation condition suffices: all sessions have been terminated. However, additional conditions may apply, such as time of day for objects with large initialization overhead that are used to support specific business functions. When the server instance determines that all its object instances have been uninitialized, it deactivates (or unloads itself).

Object deletion: This is an extremely infrequent activity, representing the uninstallation of compiled code from a host. (COM is intended to facilitate updating objects, changing the implementation of an interface without changing the interface itself, and adding new functionality via new interfaces. Thus, it replaces the actions required to uninstall an old version and install a new version with an update action.) A client requests that COM delete an object class to its Inanimate Objects. The request identifies the object class, its interfaces, and its server. COM removes the object class to its Inanimate Objects.

Object update: This is an infrequent activity, representing the installation of a new version of an object class on a host. A client requests that COM update an object class in its Inanimate Objects. The request identifies the object class, its interfaces, and its server. COM updates the object class entry in its Inanimate Objects.

The following refinement of the discussion above assumes a homogeneous environment. However, we distinguish among COM instances on different hosts. We introduce an additional part of an object's life-cycle (publication) and note how the earlier description must be refined.

Object publication: This is an infrequent activity, by which an object class on one host is made accessible to other hosts. A COM instance on one host requests that an object class on that host be added to the Inanimate Objects on another host. The request identifies the object class, its interfaces, its server, and its location. The COM instance on the second host adds the object class to its Inanimate Objects.

5. JAVA RMI

In this section, we first provide a motivational perspective on Java and the Java DOC paradigm, which uses Remote Method Invocation (RMI). We next describe Java, and how distributed object computing is made possible in a Java environment using RMI, from a technical perspective, at the conceptual and architectural strata.

Java RMI provides a way to implement distributed object computing within the Java Virtual Machine architecture. The distributed objects are Java classes, or platform-local objects which an object in the platform's JVM is authorized to use. Authorization is based on the identity of an object's source. (In release 1.2, extensions to the security model plan to allow security computations also based on the signatories a class might have as well as the permissions of the classes calling the class in question.)

The Java RMI paradigm is inadequate to provide full DOC functionality. For example, Java RMI does not provide time services, which are needed for synchronization in distributed systems. The security it provides is intended primarily to protect platform-local resources from corruption or misuse. In addition, it does not address auditing or management across platforms. The modular RMI architecture allows a developer of one or more layers to provide message integrity or confidentiality, but this is not specifically addressed by Java RMI. However, Java, as an object-oriented language, can create applications designed to effectively interact with the CORBA and COM environments.

The information presented in this section is derived primarily from documents available through the Java website (Fritzinger, 1996; Gong, 1998; Sun, 1996, 1998a-e; Wallach, 1998). Additional sources include (McGraw, 1997; Microsoft, 1997b; Oaks, 1998). This section addresses Java 1.2 (Gong, 1998; Sun, 1998a-b), for which the Access Controller provides access control to platform resources using protection domains. This document integrates information from earlier Java documentation and the Java 1.2 documentation to provide a comparative description of both implementations' security models. (Due to its novelty, most Java 1.2 documentation describes only its new aspects, relying on documentation of earlier versions for the security model.)

5.1 JAVA MOTIVATION AND BACKGROUND

The motivating vision for Java is that of platform-independent plug-and-play mobile software. The basic concept is that code written in the Java language can be downloaded to any platform and run there safely (i.e., without putting platform resources at risk) without recompilation. The term "Java" refers first and foremost to an object-oriented language which can be used for programming or for interface specification. However, "Java" also encompasses an architecture (the Java Virtual Machine or JVM) that enables the safe use of foreign code, a DOC paradigm (Java RMI), and a composable software paradigm (Enterprise Java Beans) which relies heavily on RMI.

This vision is set forth in the Java language specification, architectural descriptions, related specifications, and white papers published by Sun Microsystems and the many software developers who have adopted Java. Java provides a powerful tool for distributing applications across networks. Of course, not all applications are benevolent. In order to protect client systems from hostile code, Java applications are run on the JVM which provides tools to oversee the execution of foreign code. While these protections are not foolproof, they do make attacks using Java code much more difficult. The JVM creates a safe space (the "sandbox") on a platform in which downloaded Java objects (or "applets") can run without access to platform resources. In all the Java releases to date (version 1.1.6 and before) only code which was listed as being within the host machine's CLASSPATH was exempt from this restriction. All other code was forced to live within the sandbox was restricted by the rough granularity of the SecurityManager class. However, this rough classification and handling proved too restrictive, particularly when Java is used for distributed object computing. Java 1.2 provides a more sophisticated approach to protecting platform resources at the expense of increased security administration.

5.1.1 Security Architecture Overview

Java distinguishes between an object *class* and an object, which is a running instance of a class. A class must be *loaded* into the JVM before it can be instantiated, thereby creating an object of that class and permitting its code to be run. Since most documents on RMI state that RMI is secure because it stands on top of the normal Java security, it is assumed that the security protections described below apply to classes loaded via RMI as well. The three levels of protection provided by the JVM are described below.

The first line of defense a client has against hostile code is the Java byte-code verifier. The verifier is run on all classes that are not trusted. (In Java 1.1, this would be any class that was not a member of the CLASSPATH. In Java 1.2, all classes fall into this category.) The concept behind the verifier is that the Java virtual machine has no idea what created the byte-codes it has now been asked to run. It might have been a legitimate Java compiler, but it is possible that some other application or user created the byte-codes and did not provide the same types of checks that a Java compiler would. By checking the byte-code patterns, the Verifier attempts to determine that it is not being asked to process any instructions that would have been prevented by a legitimate compiler. The Verifier does this on two levels. On the first level, it makes sure the byte code itself is formatted correctly. Secondly, the verifier uses built in algorithms to make sure that the byte codes do not violate language guidelines. For example, the Verifier is run on method calls to verify the form of the call and appropriate accesses. Should the Verifier discover bytecodes which it cannot endorse, the class would be rejected and loading would be stopped.

The next portion of Java's security architecture is the Class Loader. A class loader is associated with a Java "Name Space." There is one name space for each source (host machine and directory) of code. This means that a virtual machine may have multiple running class loaders. The separation of Java classes into distinct name spaces helps the system in several ways. First, it allows different applications to use classes with the same name without fear of conflict since the class's name space becomes part of its proper name. Secondly, since classes are separated based on their origin, policies could, theoretically, be devised which allow different security to be placed on code from different sources by associating these policies with name spaces. Thirdly, because code loaded from different sources is isolated from other code, it makes it much more difficult for foreign code to cooperate in any way. This stage of Java's security is entirely passive. A Class Loader makes no decisions as to whether or not a class will be allowed to be loaded or run, but simply segregates the classes in order to make the general running of the system more secure.

The third aspect of the Java security environment is the Java Security Manager. The Manager is designed to supervise all security sensitive activities attempted in the Virtual Machine. If running code wishes to do something like write to a file, spawn a thread, connect to another computer, or access the operating system, the Java library consults the Security Manager. (In Java 1.2, a new class called an Access Controller is called by the Security Manager and refines its granularity.) The Security Manager is configurable and can be made to reflect a given security policy. If the policy defined in the Security Manager forbids the action in question, a Security Exception is thrown and the action is prevented. Otherwise, the action is allowed to continue unhindered, although the Security Manager could include code which logged actions, altered system state, or some other activity as a result of the attempted access.

Note that these defensive aspects do not overlap. This means if a type of attack gets through the defensive layer that should catch it, the other levels will not be able to pick up the slack and the system will have been compromised. For example, if an invalid byte code pattern is missed by the Verifier, no other level of the JVM will be able to catch this oversight.

5.2 TECHNICAL PERSPECTIVE ON JAVA AT THE CONCEPTUAL STRATUM

At the highest stratum of abstraction, Java can be seen as simply the interaction of objects (an object being a running instance of a class). As part of the language specification, everything in the Java language (with the exception of a few primitives) is an object. In terms of security, we only have an object's source location (and, in 1.2, the signatures attached to that object) from which the class was loaded. The result is that the only security consideration we can make is: given the immediate source of the Java object (there is no concept of delegation or that a class might have come from yet another location before arriving at our immediate server) what actions will the object be permitted to run on the local system.

RMI takes the process one step further and allows us to call methods of foreign classes without actually downloading the class itself to the client machine. This is of obvious use when dealing with objects such as databases where the object's size would be prohibitively large. However, no new security mechanisms are put in place. This is because the goal of Java security is the protection of the local host. From this perspective, Java doesn't care whom it is protecting the host from. Thus, while RMI calls may be made by any foreign or local host who knows how to contact the server and which possesses the appropriate stub and skeleton interfaces, the security that oversee the classes of the RMI server get their security permissions based on their source (and signer in 1.2). This means that the action would be restricted by the server's security, rather than by restrictions based on the calling client. The location of the caller has no effect on the Java security calculation. As such, things such as distribution are meaningless to Java security.

It should, of course, be pointed out that, unlike the previous two paradigms, Java is a programming language and designed to allow programmers to construct classes and methods of their choosing. As such, it is always theoretically possible for someone to create security mechanisms which would make an RMI server sensitive to who it was being invoked by and to make other such security calculations. However, these mechanisms would be at the application level and would be specific to the Java program in question. Alone, however, Java's security is simply concerned with resource protection, and is therefore not, by nature, distributed.

5.3 TECHNICAL PERSPECTIVE ON JAVA AT THE ARCHITECTURAL STRATUM

5.3.1 Java Security Defenses

In this subsection, we refine the overview of Java security defenses to highlight their use in the RMI environment.

The Java byte-code verifier is run on all classes that are not trusted. (Before Java 1.2, this would be limited to code that was considered foreign. Classes that were loaded from the CLASSPATH were assumed trustworthy and were not verified. Version 1.2 removed the concept of trusted code in favor of the concept of Permissions, which will be discussed later. As a result the verifier would be executed on all code.) The verifier attempts to guarantee that:

- Stacks will not be overflowed or underflowed
- Types are not mislabeled (for example, it would not let an InputStream class be referenced by a Hashtable handle)
- No illegal casting is done (such as trying to treat an integer as a pointer)
- Public, private, and protected markers are honored
- Register access and stores are valid
- The Java class syntax is followed throughout

The next portion of Java's security architecture is the Class Loader. (In earlier versions, this was the "Applet Class Loader" and its use was limited to "foreign code." A special class loader, called then System Class Loader, was used to load classes from the CLASSPATH, giving them special permissions. Since all code is "foreign" as far as Java 1.2 is concerned, we assume that all code must go through this step.) A class loader is associated with a Java "Name Space". There is one name space for all the classes loaded from the CLASSPATH, and one name space for each foreign source of code (host name and directory). This means that a virtual machine may have a large number of class loader objects running at the same time. Java applications may create their own implementation of the class loaders and thus define their own name space representations, although there are security risks in doing this. Applets cannot do so and must use the default “Applet Class Loader” class. Classes from the same source are all placed in the same name space even if they were downloaded at different times for different applications.

When one class attempts to reference another class, the Class Loader does a search for that class in the following manner. First, it checks the CLASSPATH name space where the system classes live. (In version 1.2 this would only contain the Java core classes plus a select few classes added by the user. In 1.1 this would include any classes known on the local machine.) This has the added advantage of preventing an application from “spoofing” a trusted class’s name: since the local name space is checked first, any overlapping classes within the malicious application would get ignored. Secondly, if the specified class cannot be found in the CLASSPATH name space, the Class Loader checks the name space of the class making the call. This allows applications to reference all their own classes. An application may reference a class in a name space other than the two mentioned above, but only if an explicit reference to this outside name space is made (for example, by specifying the URL).

RMI has its own type of class loader, appropriately named RMIClassLoader. There must be a security manager already in place before the class loader actually does anything or a security exception is thrown. (Specifically, either RMISecurityManager or a user-defined version must be in place.)

The third aspect of the Java security environment is the Java Security Manager. The Manager is designed to supervise all security sensitive activities attempted in the Virtual Machine. If loaded code wishes to do something like write to a file, spawn a thread, connect to another computer, or access the operating system, the Java library consults the Security Manager. The Security Manager is configurable and can be made to reflect a given security policy. (Of course, applets and RMI clients are not allowed to define their own security managers.) If the Security Manager detects an access that would violate its policy it throws a SecurityException. The functions of the Security Manager class tend to be of the form “checkXXXX” where XXXX is the action the function is designed to supervise. These functions perform some calculation, the result of which indicates whether or not the procedure is allowed. The SecurityManager class itself is abstract and the programmer is expected to fill in the check functions according to the desired security policy. The class RMISecurityManager is a concrete extension of the Security Manager class provided for the convenience of the programmer, which implements a highly restrictive policy, similar to the policy built into most web browsers.

A new feature of Java 1.2 is the AccessController class and the associated Permissions classes. With these new classes, Java creates the concept of a protection domain. A protection domain is a group of classes which share the same source (host and directory) as well as certain signatures. A security policy may be assigned to oversee the actions of the classes which fall into this group. (Note: this process is not directly associated with Name Spaces since it is possible for a single Class Loader, and therefor, a single name space, to cover classes which are in different protection domains.) Whenever a class within a given protection domain wishes to perform a security sensitive action, the Permissions objects associated with that protection domain will be consulted to determine whether or not that action would be permitted. It is possible for a class to belong to more than one protection domain (i.e., to have multiple signatures). In this case, the class's total permissions would be the sum of the Permissions objects of each of its member domains. As of Java version 1.2, classes will be maintained in different domains if they have different sources or signatures, even if the permissions associated with these classes would be the same. This may result in more protection domains than strictly necessary, but work is underway to optimize this.

When a security sensitive action is attempted, the Access Controller is called to verify that the invoking class is in a protection domain where such actions are permitted. The Access Controller invokes its "check Permission" method. This method compares the action attempted with the policy of the protection domain from whence the call initiated. If access is granted on that level, the method will then check that the action is permissible within the protection domain that called the class which made the security sensitive call, and so on. This way, a class cannot assume privileges it would not normally be permitted simply by calling a class that would have those Permissions.

5.3.2 Java RMI Architecture

Java Remote Method Invocation (RMI) is an extension of the Java programming language which allows Java code reference and return objects as well as basic Java types. RMI passes and returns defined Java objects through the concept of "object serialization" which "marshals" complicated data types into a stream that can be passed along the wire and reconstructed by the recipient. Only objects which are implementations of classes that implement the Java "Serializable" interface may be passed in such a manner. Likewise, only objects which are implementations of classes that extend the Java "UnicastRemoteObject" class may have methods which are called by a remote application.

RMI operates through a three-layer architecture. These layers are as follows:

- Stub/Skeleton layer—This layer represents Stubs (client side interfaces for remote classes) and Skeletons (server side interfaces for remote classes) which serve as handles to foreign objects and perform the act of marshalling and unmarshalling data which is given and received from the Remote Reference Layer respectively.
- Remote Reference Layer—This layer is responsible for transferring the marshalled data to the Transport Layer through the abstraction of a stream-oriented connection.
- Transport Layer—The Transport Layer is charged with setting up the actual connection between the objects and managing this connection.

All these layers are distinct and modular. A user could rewrite the code for one of the layers (for example, the Transport Layer could be rewritten to use UDP as opposed to TCP) without affecting the operation of the other levels in the architecture.

5.3.3 Security for Java RMI

RMI depends on the Java environment for its security. This also means that, despite the huge extension in Java's functionality which comes from RMI, there is not a corresponding increase in Java's security mechanisms. For this reason, while Java remains able to defend the host system against hostile actions by code, there are no intrinsic protections covering distributed computing.

When Java attempts to make a security decision about some action, it looks at attributes which determine the class's policy (Security Manager in 1.1, the Permissions of the protection domain in 1.2). In version 1.2 the security checking algorithm then references the application's stack and backtracks through the stack, checking the permissions at every step of the way to make sure that neither the calling class nor any of its predecessors would be violating their security policies by making the call. The problem in terms of distributed computing is that there is no mechanism for conveying stack information between hosts. The result is that a system's security has only its local context to make decisions in. Therefore, while the built in mechanisms of the JVM remain effective in their defense of the host machine, there is no built in procedure for modifying a remote caller's access based on an identity. In other words, if a remote method is called, the security calculation for any security sensitive actions which result would be exactly the same, whether the call was made from the local machine or from across the network. The server programmer would be able to build mechanisms of their own for this and effectively add application level security to the system, but RMI and the JVM provide no protection of their own.

5.3.3.1 Remarks

This outline only describes the functioning of internal Java security. As this security's only objective is the protection of the resources of the local machine, this architecture is not well suited to the nuances of distributed security. This is not, however, to say that it is impossible to implement distributed security in Java. Java itself provides a number of classes that could be used to implement application level authorization, authentication, and encryption. Using these classes, it would be possible to implement delegation and other distributed security concepts. The only thing that would be difficult to create at the application level would be a way to distribute the local calling stack for the purposes of determining permissions. Since Java prevents programmers and users from accessing pointers through the language, the call stack is likewise inaccessible. However, if one was willing to add enough complexity to one's code, a truly distributed security implementation could be created. Since, as of this time, Java has no high level security model, these implementations could vary widely producing difficulties in interoperation among Java implementations, much less interoperability between different distributed object systems.

As of yet, there is no formal high-level security model in Java. Instead, the implementation of low-level checking is used to define the model. The result is that vendors have a great deal of leeway in terms of defining the security of their systems, and that different implementations are capable of enforcing different access policies. Wallach and Felten have formalized Java’s stack inspection, which is implemented variously by different vendors, reducing it to a finite pushdown automaton (Wallach, 1998).

6. SECURE INTEROPERABILITY STRATEGY

Basic interoperability among the various DOC paradigms, without consideration for security, is being pursued vigorously. OMG is addressing COM/CORBA interoperability (OMG, 1997d-e); the vendor community is implementing interoperability bridges between COM, CORBA, and Java RMI products. The Defense Advanced Research Projects Agency (DARPA), in conjunction with numerous organizations (particularly OMG), is sponsoring an Interoperability Clearinghouse initiative (OMG, 1998e; Weiler, 1998a-b). This builds on the Workshop on Compositional Software Architectures sponsored by DARPA, OMG, *et al.* in January 1998 (OMG et al, 1998).

Just as security is not currently a primary concern in the implementation of a DOC product, it is not a primary concern for the implementation of interoperability bridges. We see several challenges to secure interoperability among distributed object computing systems. We believe that the development of sound theoretical foundations is essential to meeting these challenges.

6.1 CHALLENGES

Challenges to securing distributed object systems fall into two areas: secure operability and secure interoperability. The emphasis in this report is on potentials for, and barriers to, secure interoperability. However, systems and security domains cannot interoperate securely if the security of one or more domains or systems is poorly understood, misadministered, or broken. Secure operability of DOC systems raises significant challenges in the areas of policy definition, security administration, and intrusion/anomaly detection.

Existing and emerging DOC products, like many OS and network products, force the security administrator to express security policy in terms of product capabilities. The administrator must therefore express organizational policies and objectives in terms of the constructs and attributes the product supports. Organizational policy-makers cannot check that the policy as administered implements, or is even consistent with, their decisions. They must rely on the administrator's understanding of both their decisions and the product capabilities, constructs, and attributes. Difficulties in expressing security policies in a product-neutral way constitute a barrier to interoperability as well: it is hard to compare two implementations to determine whether they enforce the same policy, or whether one enforces a more stringent policy than another.

Vendors as well as administrators recognize the desirability of integrating the security administration for DOC middleware with that of the underlying OS and network. (This is one of the advantages COM offers in a Windows environment.) One challenge is to do this in a way that enables third-party administration tools. A more significant challenge is to do this without forcing the use of specific OS and network products.

Intrusion detection is a significant concern, particularly in distributed environments which depend on a vulnerable communications infrastructure.

Security interoperability has several levels, with corresponding different degrees of technical challenge:

- Interoperability between security domains, in the context of
- A single product (e.g., the same ORB),
- A single enabling product used by different DOC products (e.g., different ORBs that both rely on a specific X.509 implementation),
- A single technology used by different DOC products (e.g., different ORBs that both rely on a X.509),
- Interoperability between different instances of the same DOC paradigm (e.g., ORBs from different vendors) that do not share the same technology, and
- Interoperability between instances of different DOC paradigms (e.g., COM and CORBA).

6.2 STRATEGIES

We do not expect the industry to converge on a single DOC paradigm (DISA, 1998; Marcus, 1998; Tallman, 1998). Differences in approaches to distributed object computing arise not only from attempts to gain market dominance, but also and more fundamentally from differences in computing goals and philosophies. We therefore believe that the challenges identified above must be met in the context of multiple DOC paradigms, and of interoperability among DOC systems. Java and CORBA can interoperate smoothly; a Java ORB can run within the Java Virtual Machine, and a Java-to-IDL mapping has been defined. The primary concern is for interoperability between CORBA and COM.

We see several basic strategies for fostering secure interoperability among DOC paradigms. The “reference implementation” or “proof-of-concept” strategy involves developing (or funding the development of) prototypes, worked examples, or technology demonstrations. We believe that many issues would be highlighted by prototyping

- Secure interoperability bridges between CORBA security policy domains. This could be prototyped in the context of a single secure ORB.
- Services that translate security attributes (represented as credentials, tokens, or stack frames). Such services might be implemented as plug-ins to existing or in-development COM/CORBA bridges. See (Hartman, 1998) for a discussion of some of the issues to be addressed in secure COM/CORBA bridging.
- Security management tools for Java RMI. (Microsoft’s Internet Explorer 4.0 provides system administrators with tools for defining "trust-based" security zones, but this is Microsoft-specific (Microsoft, 1997b).)

Some security-related proof-of-concept activities are already underway, such as the ORB Gateway developed under DARPA funding (TIS, 1998). The potential effectiveness of this strategy is currently limited by the unavailability or immaturity of secure DOC products. We believe that, once enough products are available, attempts to develop translation services will be very implementation-specific, or will founder on a lack of theoretical foundations. Fundamental incompatibilities among security policies will be indistinguishable from those that are artifacts of implementation strategies.

The “specification” or “criteria” strategy is to ensure that specifications, standards, or criteria for DOC systems (and interoperability bridges) include requirements for security services (and translation functionality). Under this strategy, activities include analysis of existing and emerging specifications to ensure that they adequately reflect security needs. Another activity is development of a Common Criteria Protection Profiles and Security Targets for secure DOC products. Due to the tight linkage between COM and Windows NT, we believe that Microsoft should be encouraged to develop a Security Target for its implementation. We believe that Protection Profiles could be developed for a secure ORB, a secure ORB gateway, and a secure COM/CORBA bridge.

We see a strong need for a “theoretical foundations” strategy. The goal of this strategy is to provide conceptual foundations for distributed object security and tools for integrating sound theoretical foundations into product design and development. By providing clear statements of policy, requirements, and traceability of requirements to services, this strategy can improve the assurance of DOC products.

Goals for this strategy can be restated in terms of the characterization framework presented in Section 2. At the conceptual stratum, goals are to describe the behavior of a distributed object computing system, using a minimal set of terms and relationships among those terms, and to characterize security concerns as requirements on system behavior as expressed in this abstract and minimalist form. At the architectural stratum, goals are to refine the theories and models of the conceptual level to incorporate concepts and constraints that arise from architectural considerations; to restate security concerns as security policy objectives and describe those objectives in terms of security services and enforcement mechanisms; and thus to develop a more complex model of secure system behavior, which can be recognized as a realization of a more abstract model from the conceptual stratum. Also at the architectural stratum, a CORBA-specific goal is to make the semantics more complete and precise. At the implementation stratum, the goal is to gain better assurance that as-implemented interfaces correctly meet the specifications (or, preferably, correspond to the disambiguated representation of the specifications).

One activity involves development of a distributed object theory, with a security refinement of that theory. This would enable the currently informal security models underlying different DOC paradigms and products to be expressed as models of that theory. Differences among security models could then be exposed and clearly articulated, providing a foundation for analyzing interoperability issues in an implementation-independent way. Some object theories exist (see, e.g., (Abadi, 1996)), but are tied to object-oriented programming and focused on denotational semantics. Such theories are more complex than necessary to represent the different DOC paradigms, and are difficult to apply to COM since it is not a fully object-oriented paradigm. It is desirable that the distributed object theory be consistent with the Reference Model for Open Distributed Processing (ISO, 1996a). The theory presented in Section 7 constitutes a first step toward this activity.

Another activity involves development of concepts and tools for expressing security policy in an implementation-neutral (ideally, a paradigm-neutral) way, and for translating such expressions into commands at the security administrator interfaces to different DOC products. The Open Systems Security Framework developed by the International Standards Organization (ISO) provides a basis for this activity, particularly for the expression of access control policies (ISO, 1996b). So does the CORBA authorization model, which can serve as a general framework within which to discuss authorization models and policies for distributed object systems.

In this regard, the characterization of delegation policies is of particular concern. Delegation policies must be realistic - that is, capable of representing enterprise policies. They must also be realizable. That is, they must rely on attributes that can be represented within a DOC system, and they must not impose heavy requirements on communications bandwidth and on credentials processing. (Williams et al, 1996) provides a CORBA-specific approach, which could serve as a starting point for a more paradigm-neutral characterization.

An activity specific to CORBA involves characterizing the security interfaces more clearly and concisely. The CORBA Security Specification includes informal semantics and descriptions of intended behavior, but this information is dispersed and incomplete as written. This activity would therefore aid the development of secure ORBs or replaceable security services. One approach involves annotating the CORBAsec interface specifications with information from explanatory sections, and then defining a behavioral specification of each interface. This would identify ways in which specifications are ambiguous or overloaded, important since different interpretations of specifications result in barriers to interoperability. This activity could also involve translating the specifications into a formal or semi-formal representation that can be mapped to the model.

Theoretical foundations for intrusion detection are sparse and focus on patterns of network use. Distributed object computing systems face the risk that patterns of malicious activity that involve multiple objects, and particularly those that involve multiple ORBs or COM Libraries, will go undetected, just as networked systems face a risk that patterns of malicious activity spread across multiple hosts will go undetected. A fundamental question is whether theories, strategies, and mechanisms developed for network intrusion detection can be adapted to the DOC environment. At first glance, this seems plausible: Objects, like network nodes, are largely autonomous and mutually unaware. The responsibility for auditing is local. The challenges of consolidating audit data from different vendors' DOC middleware and from different network host OSs are similar.

7. THEORETICAL FOUNDATIONS

From the theoretical perspective, we have found an absence of rigorous foundations. Models of distributed objects computing systems are informal and generally incomplete. Models are presented in terms biased toward the goals of the DOC paradigm or the technologies that are expected to be used in realizing the models. Architectures are described with pictures or diagrams with informal and deliberately incomplete semantics. Similarly, interfaces are specified with incomplete semantics.

From a theoretical perspective, we believe that a rigorous re-expression of the underlying models is fundamental to analysis and comparison, and that increased rigor is needed in the characterization and description of interfaces. In Section 6, we proposed a strategy to improve the theoretical view of DOC security. In this section, we provide early insight into ongoing work on theoretical foundations.

At the conceptual stratum, a theory is needed to describe the behavior of a distributed object computing system, using a minimal set of terms and relationships among those terms. Security concerns can be expressed as requirements on system behavior as expressed in this abstract and minimalist form. We have developed an initial DOC theory to show how this might be accomplished. This abstraction is presented in Sections 7.1.

All distributed object systems make assumptions about trust that impact their security. These systems also assume that all trust decisions are made outside the system and are encoded into the design of the system. In order to compare different distributed object systems, we need to consider how they manage trust, and how trust relationships can be transferred between systems. In subsection 7.2, we begin to address static trust relationships.

The CORBA authorization model is very abstract. It attempts to describe a wide range of concrete authorization models. It can thus serve as a general framework within which to discuss authorization models and policies for distributed object systems. In subsection 7.3, we present an initial set of formalisms, based on CORBA. We believe that these formalisms will provide a paradigm-neutral basis for characterizing and comparing authorization models, and for describing delegation policies unambiguously.

7.1 TEMPLATES FOR DOC SECURITY

In this subsection we summarize research presented in (Thayer, 1998). That paper proposes a template for describing distributed object computing security. In the following paragraphs, we describe our approach to developing a theory of objects. We then present the basic theory.

As noted in the Introduction, there are many possible meanings for the term object. There is also great variation in the degree to which system entities, such as address spaces, processes, communication channels, databases, and file systems are expressed in an object-oriented manner. In establishing the DOC description template, we will assume that all these system entities can be described in some object-oriented way, even if we do not explicitly say how this description is carried out. This is often trivial though perhaps unintuitive, since practically anything can be viewed as an object.

7.1.1 Theories of Objects

There are several ways of developing a theory of Objects. One choice is to focus in on the syntactic aspects of objects and develop a formal calculus for objects (Abadi, 1996). Our choice is to focus almost exclusively on semantic aspects of objects. We do this for two reasons:

- By concentrating on semantic issues, we can much more readily regard objects as real-world entities, interconnected in much the same way as we regard other components of a computer system. On the other hand, the syntactic approach is of more interest to programming language designers.
- Our main concern is middleware, which deals with components designed and built using many different languages and programming models. Focusing in on one specific calculus, would likely cause us to emphasize those languages and programming styles which are closest to our chosen calculus.

In the course of our exposition, we will formulate various theories of objects: For our purposes a *theory* consists of

1. A set of names intended to denote the primitive sets of the theory. We will refer to these sets as sorts. The sort declarations may also state inclusion relations between the basic sets.
2. A set of names intended to denote mappings between the sorts. It is a characteristic of our model that mappings are partial. We will refer to these as *combinators*.
3. Some axioms constraining the sets and mappings.

There are two relationships between theories we need to consider: *Theory Extension* and *Theory Refinement*. Theory extension is a completely general concept: T2 extends T1 means that all sort names, constant names, function names and assumptions which occur in T1 also occur in T2. Theory refinement is a relation between two theories T1 and T2 that each extend a basic theory T.

7.1.2 The Basic Theory

An *Object Theory* is a theory which extends the Basic Object Theory defined below. An Object Theory T consists of

- A sort `obj` of objects.
- A sort `oper` of operations.
- A sort `st` of global states.

- An interpretation mapping from OBJECT to T which preserves the sorts `err` and `val`.

Operations can be applied to objects yielding a value and possibly changing the global state. In our formal model we will express this by providing two combinators for applying operations to objects:

- `app` for the value returned by applying an operation.
- `app_dt` for the global state change resulting by applying an operation.

The symbol < is to be read "is a subsort of." It is intended to denote the subset relation.

We define the Basic Object Theory (denoted OBJECT) to include mathematical sorts such as integers **Z**, reals **R** and floats, arithmetic functions on these sorts and basic set constructors such as disjoint union and cartesian product. In addition, OBJECT contains the following sorts and mappings:

Sorts:

```
val
err < val
oper
st
obj < val
```

In the following specification, `val*` denotes the set of finite sequences of `val.`

Mappings:

```
app:        st, oper, obj, val* -> val
app_dt:     st, oper, obj, val* -> st
obj_st:     st, obj             -> val
exists_p:   st, obj             -> bool
```

At the highest level of system description, the global state is completely *localizable*, that is:

- The global state is completely determined by the states of all the objects, and
- The result of an operation on an object depends only on the state of that object.

The latter condition is reasonable only in a very high-level description of a system, since one expects that operations on an object have side-effects on other objects as well.

Two refinements of the Basic Object Theory are defined to enable a more detailed representation of system behavior. The first refinement (Client Theory) introduces the concepts of *invocations* and *clients*. The second (Thread Theory) introduces the concept of a *current object*, which enables the representation of processing threads.

7.1.3 Security

The Basic Object Theory can be further refined to represent security considerations. Security concerns can be expressed in terms of *Principals* and *Resources*. A principal is an entity to which privilege (and hence accountability) is granted. Principals can be atomic or composite. Examples of atomic principals include names such as e-mail addresses or employee numbers, keys such as RSA key pairs, and groups such as corporate divisions. Examples of compound principals include roles such as system designer, and delegation principals such as task leader; compound principals are constructed from other principals by operators.

Principals can be entities within the system, such as a client or entities outside the system such as persons or corporations. We assume a principal outside the system interacts with the system—at least for purposes of authentication -- through a system entity in the sort `init`. Motivated by CORBA terminology, we will call this entity a *Sponsor*. In the CORBA architecture, once a Sponsor is succesfully authenticated, the system may create a `client` for it which can initiate system activity. Though the relationship between a Principal and its corresponding sponsor is not part of the Object Model, this relationship must be carefully considered in any system design. At the level of abstraction we are considering, we can only say that we assume that a Principal's sponsor *speaks for* the Principal in the sense of (Burrows et al, 1989).

Four categories of atomic principals are relevant to reasoning about distributed objects:

- Code sponsors (whether people or organizations) that endorse and sponsor programs. The programs are instantiated as objects and executed as threads (see below). Programs, together with supplemental information, are signed by code sponsors.
- Object sponsors (whether people or organizations) that endorse and sponsor objects. Objects are instantiations of program code and contain encapsulated data and behavior. Object sponsors are accountable for the resources consumed by objects (such as memory) and are responsible for setting policies that govern the activation and deactivation of objects.
- Thread initiators that invoke threads to act on their behalf. A thread is a sequence of method invocations on objects.
- Places where program code is stored, objects are manipulated, and threads are executed. Each place consists of a storage and execution environment.

For most purposes “resource”can be identified with “object.” Resources include databases, files and communication channels.

In (Thayer, 1998), the Thread Theory is extended to incorporate these concepts. Extensions explicitly represent authentication, i.e., the determination of whether a client speaks for a principal; trust; and authorization, i.e., the determination whether to allow an invocation.

7.2 TRUST MANAGEMENT

All distributed object systems make assumptions about trust that impact their security. For instance, CORBASec assumes that the ORB is trusted; if the ORB is tampered with, then the security mechanisms provided by the ORB may be rendered ineffective. These systems also assume that all trust decisions are made outside the system and are encoded into the design of the system. For instance, CORBASec assumes that all code installed within a domain is trusted equally; it does not address the subtleties that arise with mobile code where all code is not trusted equally. In order to compare different distributed object systems, we need to consider how they manage trust, and how trust relationships can be transferred between systems. In this subsection, we shall primarily address static trust relationships. Dynamic trust is an exciting and important new area of research, and we will address the challenges it poses in future work.

Two important concepts in trust management are *principals* and *trust*. Below, we elaborate on these concepts within the context of distributed object computing.

7.2.1 Principals

A *principal* is an entity to which privilege (and hence accountability) is granted. Examples of atomic principals include names such as E-mail addresses or employee numbers, keys such as RSA key pairs, and groups such as corporate divisions. Examples of compound principals include roles such as system designer, and delegation principals such as task leader; compound principals are constructed from other principals by operators.

Four categories of atomic principals are relevant to reasoning about distributed objects:

1. *Code sponsors* (whether people or organizations) that endorse and sponsor programs. The programs are instantiated as objects and executed as threads (see below). Programs, together with supplemental information, are signed by code sponsors.

2. *Object sponsors* (whether people or organizations) that endorse and sponsor objects. Objects are instantiations of program code and contain encapsulated data and behavior. Object sponsors are accountable for the resources consumed by objects (such as memory) and are responsible for setting policies that govern the activation and deactivation of objects.

3. *Thread initiators* that invoke threads to act on their behalf. A thread is a sequence of method invocations on objects.

4. *Places* where program code is stored, objects are manipulated, and threads are executed. Each place consists of a storage and execution environment.

Each atomic principal may have its own public/private key pair. An implementation also requires certification authorities; these play a standard role and are not discussed further in this paper.

Compound principals can be built from the above atomic principals using operators such as delegation and roles. Programs, objects, and threads are not allowed to have keys since they are handled by places that may be untrustworthy. Rather, they are associated with compound principals. The association depends on the delegation policies being used and on the trust relationships between code sponsors, object sponsors, thread initiators, and places.

7.2.2 Trust

We define trust to be a ternary relation that relates two principals and an action. This definition of trust is based on a series of articles in (Gambetta, 1988):

Definition: *P trusts Q on actions A* if P expects Q to perform the actions A where:

- P's choice of action depends on Q's actions; and
- P has not yet monitored (and may not have the capacity to ever monitor) Q's actions.

Trust is meaningful only if P may be disappointed by the actions of Q, and if such disappointment would cause P to regret his choice of action. Note that *trust* depends on Q's underlying disposition or motivation to perform certain actions; in contrast, *confidence* depends on Q's ability to perform the actions. Thus we show confidence in our doctor's ability to cure us of our ailments whereas we trust that our doctor will use those abilities when treating us.

Note that trust is not the same as risk. *Risk* is defined as the likelihood of the occurrence of an action A. Several techniques exist for handling risk. Risk mitigation techniques seek to reduce risk by eliminating primary causes of the risk. Risk transfer techniques (such as insurance) seek to transfer risk from one principal to another. Trust is a means for grounding risk when there is no capacity to determine whether the action A did indeed occur.

Most distributed object systems make some static assumptions about trust. For instance, a system may assume that all places within a security domain are trusted equally, or that all code installed in a domain is trusted (with the trust decision being made out-of-band).

Distributed object systems also include trust management mechanisms that enable them to make dynamic trust decisions. For instance, a set of name-key binding certificates enables a principal to bind a key to a name with a level of trust that is determined by the certificate set.

We represent trust as a 5-dimensional relation. The 5-tuple (P,Q,A,T,D) asserts that principal P trusts principal Q on actions A for the time interval T; D is a boolean flag that specifies whether Q can delegate P’s trust in Q to another principal. Note that this is the same representation as the 5-tuple defined in the Simple Public Key Infrastructure (SPKI) Internet Draft (Ellison et al, 1997a-b); however, in SPKI, 5-tuples represent authorization relations whereas our 5-tuples represent trust relations.

Trust management mechanisms address issues related to representing principals, certificates that assign attributes to principals, 5-tuples that capture trust relationships between principals, delegating principals and 5-tuples, and validating and reasoning with certificates and 5-tuples. Currently, we are carefully studying the kinds of trust relationships that appear in various distributed object and mobile code systems.

7.3 AUTHORIZATION MODEL

The basic computational mechanism in the CORBA object model is a *method invocation*: An *initiator* i can invoke a *method* m on an *object* o. The CORBA authorization model is defined in terms of an access decision predicate which determines whether a method invocation is permitted.

The CORBA authorization model is instantiated as follows:

- Define three sets: principals, targets, and actions;
- Associate security attributes with each principal, target, and action;
- Define an access decision predicate F on triples of security attribute sets; that is, define F to be a boolean-valued function that maps each triple of security attribute sets to either true or false;
- Define three functions that map initiators to principals, objects to targets, and method names and arguments to actions.

7.3.1 Authorization Types

The CORBA authorization model is defined in terms of eight sets: initiators, objects, method calls, principals, targets, actions, privilege attributes, and control attributes. The CORBA object model and Interface Definition Language specifications provide definitions for objects and method calls. CORBA does not define the other sets; implementations can adopt their own definitions of these sets. Table 5 contains examples of what these sets may contain.

Table 5. Example elements of sets in authorization model

Set	Example elements
Initiator	User
Object	Object reference (specified by CORBA)
Method	Method name and arguments (specified by CORBA)
Principal	Access identity, role, group

Target	Object, security domain
Action	Operation, right
Privilege Attributes	Access identity, role, group, capability, security clearance label
Control Attributes	ACL, object classification

7.3.2 Authorization Functions

The CORBA authorization model is defined in terms of seven functions over the above sets. Again, CORBA does not define these functions; implementations can adopt their own definitions of these functions. Table 6 contains the types of these functions.

Table 6. Types of functions in authorization model

Function	Type
Auth	Initiator $\rightarrow$ Principal
Targ	Object $\rightarrow$ Target
Act	Method $\rightarrow$ Action
PA	Principal $\rightarrow$ PrivilegeAttributes
CA	Target $\times$ Action $\rightarrow$ ControlAttributes
F	PrivilegeAttributes $\times$ ControlAttributes $\times$ ActionAndContextInfo $\rightarrow$ Boolean
Authorization	Initiator $\times$ Object $\times$ Method $\rightarrow$ Boolean

7.3.3 Authorization Decision

Definition: Let i Initiator, o Object, and m MethodCall. Then the authorization function is defined as follows:

Authorization(<i,o,m>) = F(PA(Auth(i)), CA(Targ(o),Act(m)))

To determine if an initiator i is permitted to invoke method m on object o, the CORBA authorization model:

1. Maps the initiator, object, and method call to a principal, target and action

2. Computes the privilege attributes of the principal and the control attributes of the target and action

3. Applies the access decision function to the privilege and control attributes

The operation is permitted if and only if the result of the access decision function is true. That is, an initiator i is permitted to invoke method m on object o if and only if Authorization(<i,o,m>) is true.

7.3.4 Delegation of Privilege Attributes

The CORBA security specification describes a range of mappings from initiators to principals and principals to privilege attributes.

We define an initiator to be a sequence of object invocations that begins with a user u:

<u, <o_1, m_1, a_1*>, … , <o_1, m_n a_n*> >

We define two functions:

UserAuth : User → Principal \\
ObjectAuth : Object → Principal

Auth : Initiator → Principal \\
Auth (<u, (o_1,m_1,a_1*), …, (o_1,m_n,a_n*) >)
= <UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_n) >

PA(<UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_n) >)
= PA(UserAuth(u))
(simple delegation)

PA(<UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_n) >)
= PA(UserAuth(u))   PA(ObjectAuth(o_n))
(composite delegation)

PA(<UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_n) >)
= PA(ObjectAuth(o_n))   PA(<UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_{n-1})>)
(combined delegation)

PA(<UserAuth(u), ObjectAuth(o_1), … , ObjectAuth(o_n) >) =
<PA(ObjectAuth(o_n)), PA(<UserAuth(u), ObjectAuth(o_1), … ,
ObjectAuth(o_{n-1})>)>
(traced delegation)

7.3.5 Discussion

The CORBA authorization model is very abstract. It attempts to describe a wide range of concrete authorization models. The model does not standardize or even suggest an access control policy, but rather serves as a general framework within which to discuss authorization models and policies for distributed object systems.

8. REFERENCES

(Abadi, 1996)	Martin Abadi and Luca Cardelli, *A Theory of Objects*, New York NY, Springer-Verlag, 1996
(Appelbaum et al, 1996)	Rob Appelbaum, Marshall Kline, and Mike Girou, The CORBA FAQ—Frequently Asked Questions, 1996-97; http://www.cerfnet.com/~mpcline/corba-faq/
(Burrows et al, 1989)	Burrows, Michael, Martín Abadi, and Roger Needham, A logic of authentication, in *Proceedings of the Royal Society*, Series A, 426(1871):233-271, December 1989. Also appeared as SRC Research Report 39 and, in a shortened form, in *ACM Transactions on Computer Systems* 8, 1 (February 1990), 18-36.
(Chandry et al, 1998)	Chandry, K. Mani, Joseph Kiniry, Adam Rifkin, and Daniel Zimmerman, "A Framework for Structured Distributed Object Computing," CS 256-80, California Institute of Technology Infospheres Project, 1997; http://www.infospheres.caltech.edu/
(Chapin, 1997a)	Chapin, Susan, *Overview of Distributed Systems Infrastructure Environments: COM/DCOM/ActiveX, CORBA, and DCE (Draft)*, The MITRE Corporation, April 1997
(Chapin, 1997b)	Chapin, Susan, *Security Features of COM, DCOM, and ActiveX (Draft)*, The MITRE Corporation, June 1997
(Chappell, 1996)	Chappell, David, *Understanding ActiveX and OLE*, 1996, Microsoft Press, Redmond, WA
(Chung et al, 1997)	Chung, P. Emerald, Yennun Huang, and Shalini Yajnik, Bell Laboratories, Lucent Technologies, Deron Liang, Joanne C. Shih, Chung-Yih Wang, Institute of Information Science, Academia Sinica, Republic of China, Taiwan, Yi-Min Wang, AT&T Labs, Research, Florham Park, New Jersey, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer," 3 September 1997; http://akpublic.research.att.com/~ymwang/papers/HTML/DCOMnCORBA/S.html
(Csurgay, 1998)	Csurgay, Peter, "Platform Evaluation for Distributed Object Computing, Project report, Plug and Play Architecture ITEM/NTNU," Distributed Systems Group, Department of Telematics, Norwegian University of Science and Technology, 4 February 1998; http://www.item.ntnu.no/~csurgay/ppp/Platform5.html
(Curtis, 1997)	Curtis, David, "Java, RMI, and CORBA," The Object Management Group, 1997; http://www.omg.org/news/wpjava.htm
(DISA, 1998)	Defense Information Systems Agency (DISA), "DII COE Distributed Applications Series: Recommendations for Using DCE, DCOM, and CORBA Middleware," DISA Joint Interoperability and Engineering Organization (JIEO) Center for Computer Systems Engineering (DISA/JIEO/JEXF), 13 April 1998; http://spider.osfl.disa.mil/dii/atd/atd_page.html

(DSTC, 1998) Distributed Systems Technology Centre, RM-ODP Home Page, http://www.dstc.edu.au/AU/research_news/odp/ref_model/ref_model.html

(Ellison et al, 1997a) Ellison, C.M., B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen, "SPKI Certificate Theory, Internet Draft," November 1997

(Ellison et al, 1997b) Ellison, C.M., B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen, "Simple Public Key Certificate, Internet Draft," November 1997

(Fritzinger, 1996) Fritzinger, J. Steven and Marianne Mueller, "Java Security," Sun Microsystems, 1996, http://www.javasoft.com

(Fukayama, 1996) Fukuyama, Francis, *Trust*, Free Press, New York, 1996

(Gambetta, 1988) Gambetta, Diego, ed., *Trust: Making and Breaking Cooperative Relations*, Basil Blackwell Publishers, 1988

(Goswell, 1995) Crispin Goswell, The Microsoft Corporation, *The COM Programmer's Cookbook*, 13 September 1995; http://premium.microsoft.com/msdn/library/techart/msdn_com_co.htm

(Gong, 1998) Gong, Li, "Java Security Architecture (JDK 1.2)," Draft Version 0.9, Sun Microsystems, 10 June 1998; http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html

(Gonsalves, 1998) Antone Gonsalves, "COM+: Talk, no action – Microsoft's new Visual Studio 6.0 won't include new object model," *PC Week Online*, 25 May 1998, http://www.zdnet.com/pcweek/news/0525/25com.html

(Grimes, 1997) Richard Grimes, *Professional DCOM Programming*, Wrox Press Ltd., Chicago IL, 1997

(Hartman, 1998) Bret Hartman, "Secure Interoperability of DCOM and CORBA?," *Distributed Object Computing* Security Column, 4 June 1997

(InfoWorld, 1998) Dana Gardner, Bob Trott, and Cara Cunningham, "Microsoft has no plans to bring COM+ to platforms other than NT 5.0," *InfoWorld Electric*, 17 October, 1998, http://www.infoworld.com

(Inprise, 1998) Inprise Corporation website, 1998, http://www.inprise.com

(IONA, 1997) IONA Technologies, "OrbixSecurity White Paper," Version 1.2, 1997, http://iona.com

(ISO, 1996a) ISO/IEC, Reference Model for Open Distributed Processing (RM-ODP), ISO/IEC 10746-1 through 4 (Information technology -- Basic reference model of Open Distributed Processing -- Part 1: **Overview**; Information technology -- Open Systems Interconnection -- Data Management and Open Distributed Processing -- Basic reference model of open distributed processing: **Descriptive model;** Information technology -- Open Systems Interconnection -- Data Management and Open Distributed Processing -- Basic reference model of open distributed processing: **Prescriptive model**; Information technology -- Basic reference model of

	Open Distributed Processing -- Part 4: **Architectural Semantics**), 1996; Parts 2 and 3 at http://www.iso.ch:8000/RM-ODP/; other parts available at (DSTC, 1998)
(ISO, 1996b)	International Standards Organization (ISO) and International Electrotechnical Commission (IEC) Joint Technical Committee (JTC) 1/SC 21, Information Technology - Open Systems Interconnection - Security Frameworks for Open Systems - Part 1: Overview; Part 2: Authentication; Part 3: Access control; Part 4: Non-repudiation; Part 5: Confidentiality; Part 6: Integrity; and Part 7: Security audit and alarms, ISO/IEC 10181-1 through 7, 26 July 1996
(ISO, 1996c)	ISO/IEC, Information Technology - Security Techniques - Key Management: Introduction, ISO/IEC 11770-1, 1996
(Johnson, 1991)	Johnson, Brad Curtis, "A Distributed Computing Environment (DCE) Framework," Document DEV-DCE-TP6-1, Open Software Foundation (now The Open Group), 10 June 1991, Cambridge, MA; http://camb.opengroup.org/dce/info/papers/dev-dce-tp6-1.ps
(Khare, 1998)	Rohit Khare and Adam Rifkin, "Trust Management on the World Wide Web," *First Monday*, Vol. 3, No. 6, 1 June 1998, http://www.firstmonday.dk/issues/issue3_6/khare/index.html
(Long, 1998)	Long, Sally, DCE Program Manager, The Open Group, "The Open Group and The Securities Industry Middleware Council Announce Security Solution for Wall Street," 4 June 1998, Cambridge MA, http://www.camb.opengroup.org/dce/tech/pki/PressRel.html
(Marcus, 1998)	Bob Marcus, General Motors, "Design Alternatives and Decisions," position paper in preparation for OOPSLA (Object-Oriented Programming Systems, Languages, and Applications) 1998 Mid-Year Workshops, http://www.acm.org/sigplan/oopsla/oopsla98/midyear/contrib/design.html
(McGraw, 1997)	Gary McGraw and Edward Felten, *Java Security: Hostile Applets, Holes, and Antidotes*, Wiley Computer Publishing, New York NY, 1997
(Microsoft, 1995)	The Microsoft Corporation, "The Component Object Model Specification," Version 0.9, 24 October 1995; http://premium.microsoft.com/msdn/library/specs/Tech1/D1/S1D137.htm
(Microsoft, 1997a)	The Microsoft Corporation, "The Distributed Component Object Model Protocol (DCOM 1.0);" http://premium.microsoft.com/msdn/library/specs/dcom/distributedcomponentobjectmodelprotocoldcom10.htm
(Microsoft, 1997b)	The Microsoft Corporation, "Trust-Based Security for Java," April 1997, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm/trust_based_security.htm
(Microsoft, 1998a)	The Microsoft Software Developer Network (MSDN) website (registration required): http://premium.microsoft.com/msdn/library/

(Microsoft, 1998b)	The Microsoft Corporation, The Microsoft Transaction Server Start Page; http://premium.microsoft.com/msdn/library/tools/transerv/F1/D21/
(Moeller, 1998)	Michael Moeller, "COM+ services add up," *PC Week Online*, 13 April 1998, http://www.zdnet.com/pcweek/news/0413/13com.html
(Mowbray, 1995)	Mowbray, Thomas J. and Ron Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects*, John Wiley and Sons, 1995
(Mowbray et al., 1997)	Mowbray, Thomas, William A. Ruh, Richard Soley, and William Ruth, *Inside Corba : Distributed Object Standards and Applications*, Addison-Wesley, 1997
(Oaks, 1998)	Oaks, Scott, *Java Security*, O'Reilly & Associates, Sebastopol, CA, 1998
(OMG, 1997a)	The Object Management Group, "A Discussion of the Object Management Architecture," January 1997, Framingham, MA; http://www.omg.org/library/omaindx.htm
(OMG, 1997b)	The Object Management Group, 1997 CORBA Buyer's Guide, 1997; http://www.omg.org/news/cbg/corbadir.pdf
(OMG, 1997c)	The Object Management Group, Proceedings of the 1997 Workshop on Distributed Object Computing Security (DOCsec '97), May 1997, Framingham, MA; http://www.omg.org/docsec/1997
(OMG, 1997d)	The Object Management Group, "COM-CORBA Interworking," 1 September 1997; ftp://ftp.omg.org/pub/docs/orbos/97-09-07.pdf
(OMG, 1997e)	The Object Management Group, "COM-CORBA Interworking Part B," 1 September 1997; ftp://ftp.omg.org/pub/docs/orbos/97-09-06.pdf
(OMG, 1998a)	The Object Management Group (OMG) website: http://www.omg.org
(OMG, 1998b)	The Object Management Group, CORBA Security Specification (Security Service Revision 1.2, Chapter 15 of CORBAservices: Common Object Services Specification), 14 January 1998
(OMG, 1998c)	The Object Management Group, Proceedings of the 1998 Workshop on Distributed Object Computing Security (DOCsec '98), May 1998, Framingham, MA; http://www.omg.org/docsec/1998
(OMG, 1998d)	The Object Management Group, "The Object Management Architecture Overview," undated, Framingham, MA; http://www.omg.org/about/omaov.htm
(OMG, 1998e)	The Object Management Group, Interoperability Clearinghouse, undated, Framingham, MA; http://www.omg.org/library/ic.htm
(OMG et al, 1998)	The Object Management Group, DARPA, Microelectronics and Computer Technology Corporation (MCC), and Object Services and Consulting, Inc., Workshop on Compositional

Software Architectures, Monterey, CA, 6-8 January 1998; http://www.objs.com/workshops/ws9801/index.html

(PeerLogic, 1998) PeerLogic, Inc., "DAIS Security: A PeerLogic White Paper," August 1998, http://www.peerlogic.com

(Petersen, 1998) Scot Petersen, "For Microsoft, COM + MTS = Component Services Architecture," *PC Week Online*, 23 March 1998, http://www.zdnet.com/pcweek/news/0323/23mcom.html

(Robinson, 1997) Steve Robinson and Alex Krassel, "COMponents," 8 August 1997; http://premium.microsoft.com/msdn/library/techart/msdn_components.htm

(SPKI, 1997) Carl Ellison, Intel Corporation, "Simple Public Key Infrastructure (SPKI) Requirements Internet Draft," 24 October 1998, http://search.ietf.org/internet-drafts/draft-ietf-spki-cert-req-02.txt

(Sun, 1996) Sun Microsystems, Remote Method Invocation Specification, 1996 and 1997; http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmi-arch.doc.htm1

(Sun, 1997a) Sun Microsystems, Java-Based Distributed Computing: RMI and IIOP in Java, 26 June 1997; http://java.sun.com/pr/1997/june/statement970626-01.html

(Sun, 1997b) Sun Microsystems, RMI and IIOP in Java - FAQ, 26 June 1997; http://java.sun.com/pr/1997/june/statement970626-01.faq.html

(Sun, 1998a) Sun Microsystems Java Technology website, http://www.javasoft.com

(Sun, 1998b) Sun Microsystems Java RMI website, http://java.sun.com/products/jdk/rmi/index.html

(Sun, 1998c) Sun Microsystems, Java Remote Method Invocation - Distributed Computing for Java, 26 May 1998; http://www.javasoft.com/marketing/collateral/javarmi.html

(Sun, 1998d) Sun Microsystems, Java Distributed Computing Technology Further Enabled by Support for IIOP, 8 July 1998; http://java.sun.com/pr/1998/07/pr980708.html

(Sun, 1998e) Sun Microsystems, Java Remote Method Invocation Specification, JDK 1.2 Beta 4, July 1998; http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

(Tallman, 1998) Tallman, Owen and J. Bradford Kain, "COM versus CORBA: A Decision Framework," Quoin, Inc., Cambridge, MA, 1998; http://www.quoininc.com/quoininc/COMCORBA.html

(Thompson et al, 1997) Thompson, Craig, Object Services and Consulting, Inc., Ted Linden and Bob Filman, Microelectronics and Computer Technology Corporation (MCC), "Thoughts on OMA-NG: The Next Generation Object Management Architecture," 1997; ftp://ftp.omg.org/pub/docs/ormsc/97-09-01.html

(TIS, 1998) Trusted Information Systems (TIS), "Advanced Research and Security Engineering: CORBA Security - ORB Gateway," TIS/Network Associates, 1998; http://www.tis.com/research/applied/arse_corba.html

(TOG, 1992)	The Open Group, "Security in a Distributed Computing Environment (DCE)," Document OSF-O-WP11-1090-3, Open Software Foundation (now The Open Group), January 1992, Cambridge, MA; http://www.camb.opengroup.org/dce/info/papers/osf-o-wp11-1090-3.ps
(TOG, 1996)	The Open Group, "Distributed Computing Environment (DCE) Overview," Document TOG-DCE-PD-1296, 1996, Cambridge, MA; http://www.camb.opengroup.org/dce/info/papers/tog-dce-pd-1296.htm
(TOG, 1997a)	The Open Group, "The Open Group Architectural Framework - Version 3," Cambridge MA, 1997, http://www.rdg.opengroup.org/public/arch/
(TOG, 1997b)	The Open Group, "Common Security: CDSA (Common Data Security Architecture) and CSSM (Common System Security Manager)," Document C707, ISBN: 1-85912-194-2, 1997; http://www.opengroup.org/onlinepubs/9629299/toc.htm
(TOG, 1997c)	The Open Group, "CAE Specification: DCE 1.1: Authentication and Security Services," Document C311, August 1997, Cambridge, MA; http://www.opengroup.org/onlinepubs/9668899/toc.htm
(TOG, 1998)	The Open Group, Open Group Distributed Computing Support Titles, 1998, Cambridge, MA; http://www.opengroup.org/public/pubs/catalog/dc.htm and http://www.opengroup.org/pubs/catalog/web.htm
(Wallach, 1998)	Dan S. Wallach and Edward W. Felten, "Understanding Java Stack Inspection" *in Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 3-6 May 1998, IEEE Computer Society Press.
(Wallnau et al, 1997)	Kurt Wallnau, Nelson Weiderman, and Linda Northrop, "Distributed Object Technology with CORBA and Java: Key Concepts and Implications," CMU/SEI-TR-97-04 and ESC-TR-97-004, Carnegie-Mellon University Software Engineering Institute, June 1997; http://www.sei.cmu.edu/publications/documents/97.reports/97tr004/97tr004title.htm
(Weiderman et al, 1997)	Nelson Weiderman, Linda Northrop, Dennis Smith, Scott Tilley, Kurt Wallnau, "Implications of Distributed Object Technology for Re-engineering," CMU/SEI-TR-97-05 and ESC-TR-97-005, Carnegie-Mellon University Software Engineering Institute, June 1997; http://www.sei.cmu.edu/publications/documents/97.reports/97tr005/97tr005abstract.html
(Weiler, 1998a)	Weiler, John A., OMG Liaison Officer, The OBJECTive Technology Group, "The Interoperability Clearinghouse Initiative: Component-Based Architecture Modeling for a Distributed Enterprise," 26 May 1998; http://www.theotg.com/archives/presentations/theotg/index.htm

(Weiler, 1998b)	Weiler, John A., OMG Liaison Officer, The OBJECTive Technology Group, "From Architectures to Reality: Making the promise of "Plug and Play" work - An Interoperability Clearinghouse White Paper: Using Distributed Component Architecture Modeling (DCAM) Technology," presented at the Interoperability Clearinghouse initiative kick off meeting at the OMG Technical Committee meeting, 8 June 1998, ftp://ftp.omg.org/pub/docs/omg/98-07-01.{rtf, ps, doc, txt} or http://www.theotg.com/archives/whitepapers/index.html
(Williams et al, 1996)	James G. Williams, Don Faatz, Bill Herndon, and Dale Johnson, The MITRE Corporation, and Mark Wales, NSA/R23, *CORBA Threat Mitigation Model*, 16 October 1996, http://www.omg.org:80/docs/security/98-06-01.doc
(Williams, 1994)	Sara Williams and Charlie Kindel, The Microsoft Corporation, "The Component Object Model: A Technical Overview," October 1994; http://premium.microsoft.com/msdn/library/techart/msdn_comppr.htm

9. APPENDIX

This appendix provides a concise presentation of information about security-relevant objects, interfaces, and attributes to facilitate the development of interoperability bridges. This information is dispersed throughout the CORBASec, COM, and Java specifications and documentation.

9.1 CORBA SECURITY SERVICES

Security services provide access control (more precisely, control on the invocation of an object interface via a specific object reference by an entity with a set of credentials), accountability (audit, non-repudiation), authentication, and secure associations between objects.

Security services are provided via interfaces to security-related objects. Security-related objects manage the security-relevant information indicated above (e.g., required rights, privileges, audit requirements) and perform security processing. The CORBASec Specification does not specify the processing associated with invocations of security interfaces, but that processing is described or suggested by the explanations accompanying each interface specification. Implementors of security-related objects may extend the definitions of the specified interfaces, e.g., to provide optional parameters.

Table 7 summarizes the security objects, their interfaces, and the operations on those interfaces. An asterisk identifies an operation required for Level 1 compliance; an (l) indicates a location-constrained object; two asterisks identify security replaceable interfaces.

Table 7. CORBA Security Objects, Interfaces, and Operations

Security Object	**Interfaces**	**Operations**
Principal Authenticator (l)	PrincipalAuthenticator	authenticate, continue_authentication
Current	Current	get_attributes*, set_credentials, get_credentials, received_credentials, own_credentials, received_security_features, get_policy, required_rights_object, principal_authenticator, access_decision, audit_decision, create_qop_policy, create_mechanism_policy, create_invoc_creds_policy
	RequiredRights	get_required_rights, set_required_rights
Credentials (l)	Credentials	copy, destroy, set_security_features, get_security_features, set_privileges, get_attributes,

		is_valid, refresh
Target Application Object	Policy	get_policy, get_domain_managers, set_policy_override
Application Access Decision (l)	AccessDecision	access_allowed
Audit Decision (l)	AuditDecision	audit_needed, audit_channel
Audit Channel (l)	AuditChannel	audit_write, audit_channel_id
Non-repudiation Credentials	NRCredentials	set_NR_features, get_NR_features, generate_token, verify_evidence, get_token_details, form_complete_evidence
Vault (l)	**VaultInterface	init_security_context, accept_security_context, get_supported_mechs
Security Context (l)	**SecurityContext	Received_credentials, security_features, continue_security_context, protect_message, reclaim_message, is_valid, refresh
Secure Invocation Policy Objects: SecClientSecureInvocation, SecTargetSecureInvocation	SecurityAdmin::SecureInvocation Policy	set_association_options, get_association_options
Delegation Policy: SecDelegation	SecurityAdmin::DelegationPolicy	set_delegation_mode, get_delegation_mode
Access Policy Objects: SecClientInvocationAccess, SecTargetInvocationAccess	SecurityAdmin::AccessPolicy RequiredRights (object that is an attribute of the Current object)	get_effective_rights; additional operations as defined by policy provider get_required_rights, set_required_rights
Application Access Policy: SecApplicationAccess	SecurityAdmin::AccessPolicy (no standard interfaces specified)	
Invocation Audit Policy Objects: SecClientInvocationAudit, SecTargetInvocationAccess	SecurityAdmin::AuditPolicy	set_audit_selectors, clear_audit_selectors, get_audit_selectors, set_audit_channel
Application Audit Policy uses the Audit Decision and Audit Channel objects	AuditDecision AuditChannel	audit_needed, audit_channel audit_write, audit_channel_id
Non-repudiation Policy	NRService::NRPolicy	get_NR_policy_info, set_NR_policy_info
Domain Manager Object: Domain Access Policy	SecurityAdmin::DomainAccessP olicy	grant_rights, revoke_rights, replace_rights, get_rights

Security-related objects can be categorized according to the types of other objects to which their interfaces are available (OMG, 1998b, p. 15-81). *Application-visible* security-related objects are the Principal Authenticator, Current, Credentials, Application Access Decision, Audit Decision, Audit Channel, and Non-repudiation Credentials objects. *Administrator-visible* objects are the Secure Invocation Policies, Delegation Policy, Access Policies, Invocation Audit Policy, Application Audit Policy, and Domain Manager objects. Finally, the Credentials, Secure Invocation, Vault, Security Context, Access Control, Access Decision, Audit Decision, and Audit Channel objects are visible to implementors of ORBs and security services.

Table 8 identifies parameters input to security-relevant interfaces.

Table 8. Parameters Used by Security-Relevant CORBASec Interfaces

Parameter	**Operations**
security_name	authenticate
auth_data	authenticate
privileges	authenticate
creds	authenticate, continue_authentication, set_credentials, create_invoc_creds_policy, audit_write
auth_specific_data	authenticate, continue_authentication
continuation_data	authenticate, continue_authentication
response_data	continue_authentication
direction	set_security_features, get_security_features, set_association_options, get_association_options
security_features	set_security_features
force_commit	set_privileges
requested_privileges	set_privileges
actual_privileges	set_privileges
attributes	get_attributes
expiry_time	is_valid
policy_type	get_policy
policies	set_policy_override
set_add	set_policy_override
cred_type	set_credentials, get_credentials
del	set_credentials
obj_ref	get_security_names
qop	create_qop_policy, protect_message, reclaim_message
mechanisms	create_mechanisms_policy
event_type	audit_needed, audit_write
value_list	audit_needed
time	audit_write
descriptors	audit_write
event_specific_data	audit_write
cred_list	access_allowed
target	access_allowed, init_security_context
operation_name	access_allowed, get_required_rights, set_required_rights
target_interface_name	access_allowed
obj	get_required_rights
interface_name	get_required_rights, set_required_rights
rights	get_required_rights, set_required_rights, grant_rights, revoke_rights
rights_combinator	get_required_rights set_required_rights, replace_rights
attrib_list	get_effective_rights
rights_family	get_effective_rights, grant_rights, revoke_rights, replace_rights, get_rights

priv_attr	grant_rights, revoke_rights, replace_rights, get_rights
del_state	grant_rights, revoke_rights, replace_rights, get_rights
object_type	set_audit_selectors, clear_audit_selectors, replace_audit_selectors, get_audit_selectors, set_association_options, get_association_options, set_delegation_mode, get_delegation_mode
events	set_audit_selectors, clear_audit_selectors, replace_audit_selectors, get_audit_selectors
selectors	set_audit_selectors, replace_audit_selectors
audit_channel_id	set_audit_channel
requires_supports	set_association_options, get_association_options
options	set_association_options
mode	set_delegation_mode
creds_list	init_security_context, accept_security_context
target_security_name	init_security_context
delegation_mode	init_security_context
association_options	init_security_context
mechanism	init_security_context
mech_data	init_security_context
chan_bindings	init_security_context, accept_security_context
security_token	init_security_context
security_context	init_security_context, accept_security_context
in_token	accept_security_context, continue_security_context
out_token	accept_security_context, continue_security_context
message	protect_message, reclaim_message
text_buffer	protect_message, reclaim_message
token	protect_message, reclaim_message

Table 9 identifies parameters specific to non-repudiation.

Table 9. Parameters Used by CORBASec Interfaces Supporting Non-Repudiation

Parameter	Operations
requested_features	set_NR_features
actual_features	set_NR_features
input_buffer	generate_token
generate_evidence_type	generate_token
include_data_in_token	generate_token
generate_request	generate_token
request_features	generate_token
input_buffer_complete	generate_token
nr_token	generate_token
evidence_check	generate_token, verify_evidence
input_token_buffer	verify_evidence
form_complete_evidence	verify_evidence
token_buffer_complete	verify_evidence, get_token_details
output_token	verify_evidence
data_included_in_token	verify_evidence, get_token_details
evidence_is_complete	verify_evidence
trusted_time_used	verify_evidence, form_complete_evidence
complete_evidence_before	verify_evidence, form_complete_evidence

complete_evidence_after	verify_evidence, form_complete_evidence
token_buffer	get_token_details
token_generator_name	get_token_details
policy_features	get_token_details
evidence_type	get_token_details
evidence_generation_time	get_token_details
evidence_valid_start_time	get_token_details
evidence_validity_duration	get_token_details
request_included_in_token	get_token_details
request_features	get_token_details
input_token	form_complete_evidence
output_token	form_complete_evidence
nr_policy_id	get_NR_policy_info
policy_version	get_NR_policy_info
policy_effective_time	get_NR_policy_info
policy_expiry_time	get_NR_policy_info
supported_evidence_types	get_NR_policy_info
supported_mechanisms	get_NR_policy_info
requested_mechanisms	set_NR_policy_info
actual_mechanisms	set_NR_policy_info

9.2 COM SECURITY SERVICES

Table 10 identifies the specified interfaces for enforcing access control policies. This information is derived from (Microsoft, 1995; Grimes, 1997).

Table 10. COM Access Control Policy Enforcement Interfaces

Type	**Interface**	**Function/Constant**
Activation Security	IActivationSecurity	GetSecurityDescriptor
Call Security	General API	RPC_C_AUTHN, RPC_C_IMP, CoInitializeSecurity, CoQueryAuthenticationServices, CoRegisterAuthenticationService
Call Security	IClientSecurity	QueryBlanket, SetBlanket, CopyProxy
Call Security	Client APIs	CoQueryProxyAuthenticationInfo, CoSetProxyAuthenticationInfo, CoCopyProxy
Call Security	IServerSecurity	QueryBlanket, ImpersonateClient, RevertToSelf
Call Security	Server APIs	CoGetCallContext, CoSetCallContext, CoQueryClientAuthenticationInfo, CoImpersonateClient, CoRevertToSelf

Table 11 identifies security-relevant constructs and their security-relevant attributes. To avoid ambiguity, we distinguish between the multiple meanings of such terms as object and server. Terms that do not appear in the COM Specification and supporting documentation are indicated with an asterisk (*).

Table 11. Security-Related Constructs and Attributes in COM

Common Usage Term	Construct	Attribute
Client	Client	principal, privileges (represented as an access token), process (the process in whose process space it is running)
	Process	threading, host (the host on which it is running)
Object	Object class	Interfaces, permissions, host (the host on which it is stored)
	Object instance	interface pointers, server instance
Server	Server type	object classes, security-awareness (yes/no), flavor (in-process or out-of-process)
	Server instance	principal, privileges (represented as an access token)
COM	COM instance*	Inanimate Objects* (set of object classes, implemented via the Registry in NT 4.0), Animate Objects* (set of interface pointers, implemented via the Running Objects Table)
	Request	client, interface
	Session*	client, object instance

9.3 JAVA

Table 12 summarizes security classes in Java 1.2. These classes all belong to the java.security package. The following classes belong to the java.security.cert package and support public key certificates:

- Certificate,
- Revoked Certificate,
- X509Certificate,
- X509CRL, and
- X509Extension.

The following classes belong to the java.security.spec package and support key generation, import, and export:

- DSAParameterSpec,
- DSAPrivateKeySpec,
- DSAPublicKeySpec,
- EncodedKeySpec,
- KeySpec,
- PKCS8EncodedKeySpec, and
- X509EncodedKeySpec.

The following classes belong to the javax.crypto.spec package:

- DESKeySpec,
- DESParameterSpec,
- DESedeKeySpec,
- DHGenParameterSpec,
- DHParameterSpec,
- DHPrivateKeySpec,

- DHPublicKeySpec,
- PBEKeySpec,
- PBEParameterSpec,
- RSAPrivateKeyCrtSpec,
- RSAPrivateKeySpec, and
- RSAPublicKeySpec.

The following classes belong to the javax.crypto package:

- Cipher,
- NullCipher,
- CipherInputStream,
- CipherOutputStream,
- CipherSpi,
- KeyAgreement,
- KeyAgreementSpi,
- KeyGenerator,
- KeyGeneratorSpi,
- SealedObject,
- SecretKey,
- SecretKeyFactory, and
- SecretKeyFactorySpi.

Table 13 summarizes other Java 1.2 classes that are security-relevant. This information is derived from (Gong, 1998; Oaks, 1998).

Table 12. Java Security Classes and Supported Security Functionality

Security Functionality	**Java Classes and [Interfaces]**
Authentication and Security Association	Identity [Principal], IdentityScope
Authorization and Access Control	AccessControlContext, AccessController, Permission, its extensions (AllPermission, BasicPermission, SecurityPermission, UnresolvedPermission), and sets of permissions (PermissionCollection and its extension Permissions), GuardedObject [Guard], Policy
Platform Protection	CodeSource, SecureClassLoader, ProtectionDomain
Encryption, Key Management, and Crypto Applications	
Algorithms	AlgorithmParameters, AlgorithmParameterGenerator, AlgorithmParametersSpi, AlgorithmParameterGeneratorSpi
Supporting Mechanisms	SecureRandom
Security Provider Management	Provider (for Spi classes), Security
Key Management	Key, KeyFactory, KeyFactorySpi, KeyPair, KeyPairGenerator, KeyPairGeneratorSpi, KeyStore, PublicKey, PrivateKey
Digital Signature	Signature, SignatureSpi, SignedObject, Signer

Message Digest	MessageDigest, MessageDigestSpi, DigestInputStream, DigestOutputStream

Table 13. Security-Relevant Java Classes

Purpose	Classes
Permissions	java.awt.AWTPermission, java.io.FilePermission, java.io.SerializablePermission, java.lang.RuntimePermission, java.lang.reflect.ReflectPermission, java.net.NetPermission, java.net.SocketPermission, java.util.PropertyPermission
Class Loaders	java.lang.ClassLoader, java.net.URLClassLoader, java.rmi.server.RMIClassLoader
Security Managers	java.lang.SecurityManager, java.rmi.RMISecurityManager