Flexible Control of Downloaded Executable Content

TRENT JAEGER
IBM Thomas J. Watson Research Center
ATUL PRAKASH
University of Michigan, Ann Arbor
JOCHEN LIEDTKE
and
NAYEEM ISLAM
IBM Thomas J. Watson Research Center

We present a security architecture that enables system and application access control requirements to be enforced on applications composed from downloaded executable content. Downloaded executable content consists of messages downloaded from remote hosts that contain executables that run, upon receipt, on the downloading principal's machine. Unless restricted, this content can perform malicious actions, including accessing its downloading principal's private data and sending messages on this principal's behalf. Current security architectures for controlling downloaded executable content (e.g., JDK 1.2) enable specification of access control requirements for content-based on its provider and identity. Since these access control requirements must cover every legal use of the class, they may include rights that are not necessary for a particular application of content. Therefore, using these systems, an application composed from downloaded executable content cannot enforce its access control requirements without the addition of application-specific security mechanisms. In this paper, we define an access control model with the following properties: (1) system administrators can define system access control requirements on applications and (2) application developers can use the same model to enforce application access control requirements without the need for ad hoc security mechanisms. This access control model uses features of role-based access control models to enable (1) specification of a single role that applies to multiple application instances; (2) selection of a content's access rights based on the content's application and role in the application; (3) consistency maintained between application state and content access rights; and (4) control of role administration. We detail a system architecture that uses this access control model to implement secure collaborative applications. Lastly, we describe an implementation of this architecture, called the Lava security architecture.

Authors' addresses: T. Jaeger, IBM Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532; email: jaegert@watson.ibm.com; A. Prakash, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 2231 EECS Bldg., Ann Arbor, MI 48105; email: aprakash@eecs.umich.edu.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; *Authentication*; *Invasive software* (e.g., viruses, worms, Trojan horse); K.6.4 [**Management of Computing and Information Systems**]: System Management—*Centralization/decentralization*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software* (e.g., viruses, worms, Trojan horse)

General Terms: Design, Management, Security

Additional Key Words and Phrases: Access control models, authorization mechanisms, authentication, collaborative systems, role-based access control

1. INTRODUCTION

Downloaded executable content, messages that contain programs that are executed upon receipt, is enabling the emergence of many new applications. Examples of executable content include Java applets [Gosling et al. 1996], Tcl scripts [Ousterhout 1994], computational e-mail messages [Borenstein 1994], and replicated messages [Knister and Prakash 1993]. In most cases, executable content is implemented using a powerful language that can display user interfaces, engage users in a dialogue, return the results to the content provider, etc. A key feature of these languages is that content messages can be automatically downloaded to a wide variety of platforms and executed without recompilation. This technology is particularly useful in some emerging distributed applications, such as electronic commerce, network information services, collaborative systems, and workflow systems, where the user may only run the application once or where application actions are generated dynamically.

As noted previously [Borenstein 1992], a major problem with executing downloaded content is that, if unchecked, it can enable attackers to gain unauthorized access to the downloading principal's system resources. Thus, content interpreters attempt to enforce control of content operations. However, early content interpreters' security models are either too rigid to build complex systems (e.g., Java-enabled Netscape, ATOMICMAIL, Safe-Tcl, and Java appletviewer [Borenstein 1992; 1994; Dean et al. 1996; Jaeger and Prakash 1994]) or require the ad hoc development of security infrastructure for each application (e.g., Telescript, Inferno, and Tcl [Dorward et al. 1996; Gallo 1996; Levy and Ousterhout 1995; CNRI 1998; White 1995]). For example, Java-enabled Netscape prevents all I/O except communication back to the source IP address. On the other hand, Tcl lets us compose systems from trusted and untrusted interpreters, but much skill is required to compose a *secure* system from these components.

Improvements have been made to these access control models, but they still lack the flexibility to compose arbitrary applications. A number of subsequent content execution systems were developed with more flexible access control models [Islam et al. 1997; Netscape 1997; Gong 1997b; Ousterhout et al. 1998]. However, these systems all assign content rights based solely on their class name and provider. The problem is that for

content to execute properly under all legal conditions, the rights assigned to content must be the union of all possible rights the content may ever need. For example, in JDK 1.2 [Gong 1997b], a thread's rights are the intersection of the rights of the classes invoked by the thread, but this intersection of all rights ever needed is not necessarily consistent with the rights needed in the application's current state. For example, a collaborative editor may want to restrict the content it downloads to only operate upon the files being explicitly shared even though the collaborator may, in general, be able to modify a large set of files. Thus, applications would have to create ad hoc security mechanisms to enforce these stricter requirements.

The focus of this paper is to develop a security architecture that enables applications to be constructed from downloaded executable content that can be restricted using system and application security policies. This security architecture defines an access control model that we show is sufficient to enforce both system and application access control requirements. This architecture supports the following tasks: (1) specifying the requirements for content to be assigned the identity of a particular principal (e.g, an application or a role within an application); (2) specifying the rights commensurate with the content's role in the application and the application's state; (3) maintaining the content's permissions relative to the application's state; and (4) enforcing these permissions throughout the content's execution. System administrators specify security policy for applications and the limits within which applications and users may specify policy refinements. Content loading is done using a secure system service, so the system's security requirements can be enforced on the content. We demonstrate the architecture by showing how security policies for a distributed collaboration example are defined and enforced. We then detail an implementation of the architecture in the Lava operating system, which is composed on a successor to the L4 μ-kernel [Liedtke 1995].

Throughout this paper, we assume a conventional protection model, where *principals* (e.g., users, groups, and services) execute processes that perform *operations* (e.g., read, write, and execute) on *objects* (e.g., files, URLs, and communication channels). The privileges of a principal in performing operations on objects are called *access rights* of the principal or the principal's *permissions* [Wobber et al. 1994]. A *protection domain* is a program and its data objects encapsulated such that data objects are only accessible to the domain or through designated entry points [Saltzer and Schroeder 1975].

The structure of this paper is as follows: In Section 2, we describe an example problem we believe is representative of distributed applications that can be implemented using downloaded executable content. In Section 3, we examine how researchers approached similar problems in the past. In Section 4, we define an access control model in which security policy for solving these problems can be formally expressed. In Section 5, we detail an architecture in which security policy for the example application can be

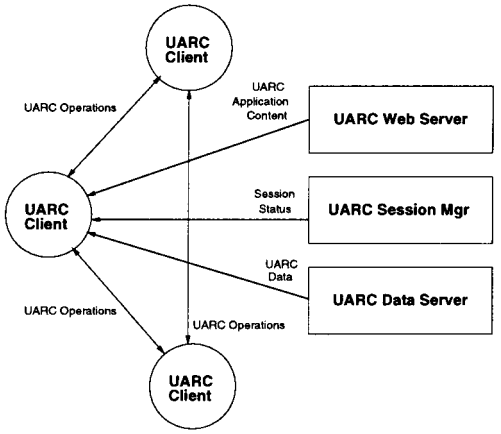

Fig. 1. A typical UARC session: The UARC servers download content and application data to UARC clients who exchange operations that modify the application data.

expressed and enforced. In Section 6, we discuss our experiences in implementing this architecture into an operating system environment that configures systems from downloaded content. In Section 7, we summarize and describe future work.

2. EXAMPLE

We use the Upper Atmospheric Research Collaboratory (UARC) [Clauer et al. 1995; Lee 1996] application as our example for identifying the security requirements for distributed applications built from downloaded executable content. UARC is a collaborative system for viewing, discussing, recording, and annotating atmospheric test data. A UARC session (i.e., an instance of an execution of the UARC application) is shown in Figure 1. First, a UARC client retrieves the UARC application from a UARC system web server. The UARC application may consist of content authored by a variety of principals. These principals may be official developers of the UARC system or outside developers whose content was found to add value to the UARC application. Once the UARC application is downloaded, it calls a UARC session manager to enter the UARC client into a collaborative session. The session manager adds the UARC client to the session in a role suitable to the collaborative group. A collaborative session is a distributed application where a set of remotely located collaborators can perform operations on a UARC session state replicated at each of the collaborators' sites. Each collaborator operation in a UARC session is implemented using downloaded executable content executed by all collaborators to keep their shared state consistent. Therefore, each UARC client may receive and execute content written by any other client. In general, any content may request that operations be performed by local services (e.g., file system and network system) or any other content on its behalf. However, these operations must

be controlled to prevent the content providers from gaining unauthorized access to the UARC client's resources.

We consider the UARC application a canonical example of a distributed application built using downloaded executable content. First, there is a main UARC application that ensures that the application performs correctly. The UARC application content performs actions on behalf of the downloading principal. Second, the UARC main application utilizes the services of third-party application content that helps it by performing specialized tasks. In general, we expect that application developers will compose their applications using their favorite supporting content. Third, UARC uses downloaded executable content to implement operations from remote principals involved in the application. Many applications may benefit by executing operations at the site of the application data or at the site of the user of the application. For example, workflow activities in a workflow system and marketing activities in an electronic marketplace could be implemented as downloaded executable content and downloaded on demand by users of those applications.

We review the specific requirements that the various types of UARC content need in order to perform their actions properly. First, the UARC application content needs to perform the following actions:

—communicate with the UARC session manager, UARC data server, and session collaborators;

—read and write atmospheric data recordings, annotations, and discussions;

—download and execute outside and collaborator content:

—access system resources, such as the CPU, disk, and screen

The extent of the resources available for the purposes specified above can be largely predetermined. Therefore, a system administrator may specify many of the permissions that are to be granted to the UARC application. However, in order for the UARC application content to execute properly, it must be informed of the location of some objects that are dependent on the individual client (i.e., the user). For example, each UARC client may choose the files used for storing their recordings, annotations, and discussions. Therefore, the derivation of UARC application permissions may depend on input from system administrators and users. However, users should be restricted in the scope of their permission management.

The security policy enforced on outside content depends on the purpose of the content. In general, the rights of content used by an application depend on the state of the application. For example, the rights of a statistical package depend on the data needed by the package. It is also conceivable that some outside content (e.g., system services) may be more trusted than the application itself, so it must be possible to load content that is permitted different rights.

The security requirements of collaborator content depend largely on the state of the UARC application. Authentication constraints, such as content freshness or content identity, must be verified to prevent attacks, like content replay or content replacement. If an attacker can replay content or get other content run with the same access rights, then the application can be invalidated. The permissions assigned to content depend on the state of the application as well. The UARC application has different types of collaborators who may be able to perform different actions. We discuss two such collaborator types: scientists and novices. Scientists use the UARC application to perform experiments, while novice collaborators use it to learn about atmospheric data analysis. At present, most actions are performed by scientists, while the novices may only view data and chat with others. The security requirements of the two groups is described below:

—**Scientists**:
  —communicate with the UARC data server;
  —read and write atmospheric data recordings, annotations, and discussions;
  —change the view of collaborator;
  —engage in a text chat.

—**Novices**:
  —communicate with the UARC data server;
  —read atmospheric data recordings, annotations, and discussions;
  —engage in a text chat.

Any participant must be able to retrieve data from the UARC data server chosen by the UARC application. The files that users have explicitly identified to the UARC application as sharable must also be accessible to the scientists. Also, the UARC security policy must control access to both system and application objects. For example, scientists may change the data being displayed, but not the state of the other collaborators' displays (e.g., location, colors, etc.). Thus, in order for the UARC application to run correctly, it must restrict its collaborators to only those objects currently being used in the collaboration. However, current operating systems do not provide application developers with security infrastructure such that they can enforce their own policies, so they must build an ad hoc security architecture, which is an arduous and error-prone task.

The problem that we address in this paper is to design a system security architecture that supports the enforcement of both system and application access control policies. Fundamentally, access control is simply controlling a principal's ability to perform operations on objects. In this sense, there is no difference between authorizing access to either system and application objects. However, the handling of principals and their security policy becomes more complex because application principal's rights may depend on a wider variety of factors and more principals are able to administer security policy. Our goal is to design an access control model and system architecture that provides enforcement for mandatory system security

policies (i.e., policies that are defined and controlled by system administrators) and permit policies to be added within those mandatory restrictions, checking that system security requirements are not violated.

We use the basic security requirements of the UARC application as motivation for the design of the access control model and system architecture. The UARC application security requirements are summarized below:

(1) Verify the source integrity that the content is designed for the purpose requested; and, for collaborator content only, the freshness of the downloaded executable content;

(2) Derive the initial content permissions on the basis of information at download time, such as its downloading principal, its provider, the application for which it is used, the application's current state, and the content's role in the application (note that the content's rights are not required to be a subset of the downloading principal's rights);

(3) Combine, within limits, permissions delegated from multiple principals including system administrators, users, applications, and collaborators;

(4) Authorize content operations on the basis of content-specific permissions (i.e., not the user's permissions);

(5) Control access to system resources, such as files, communication channels, environment variables, disk, CPU, etc. and application object;

(6) Enable a principal's permissions to evolve (via addition and removal of rights) in a limited way as application state changes to maintain least privilege;

(7) Permit delegators to revoke their delegations, if desired.

3. RELATED WORK

The enforcement of access control policies is important in a variety of contexts. In this section, we examine some key research in access control enforcement in four areas: (1) collaborative systems; (2) language systems; (3) operating systems; and (4) distributed systems. Many systems define novel access control models and/or authorization mechanisms, but these models lack the flexibility required by the UARC application, and the authorization mechanisms do not support flexible control of system and application objects. This is not intended to be an exhaustive survey (e.g., there is an enormous amount of work in operating systems security). Rather, we intend to give a sense of access control model principles and their effectiveness.

Access control in collaborative systems (i.e., groupware) is difficult because a significant amount of ad hoc sharing is necessary, and there may be different access control policies for each collaborator. Access control first became relevant to groupware researchers with the advent of active mail [Goldberg et al. 1992], a system where collaborative actions are embodied in e-mail messages that are executed by the receiver (i.e., downloading

principal). Initial work to protect receivers from malicious mails restricts content to a few public resources [Borenstein 1992; 1994]. Attempts to support more flexible access control policies makes it possible for content to obtain rights shared between the content provider and downloading principal [Jaeger and Prakash 1994] and permit user selection of shared rights [Jaeger and Prakash 1995]. Both systems depend on the users properly managing the permissions of content, but it is recognized that users may easily be spoofed. More comprehensive environments for access control that are buttressed by mandatory security policies are necessary. Gong's Enclaves system [Gong 1997a] uses a trusted "session leader" to determine the security policy for the group, and so in general the leader must understand and enforce the system security policy for each individual downloading principal. Such trust among remote principals may not be possible, although agreement is probably necessary to execute the system. Foley and Jacob define a security model in which the permissions associated with a collaborative process are determined by the activities that are executed [Foley and Jacob 1995]. We built upon this approach by restricting permissions to a system-administrator-specified policy [Jaeger et al. 1996]. However, this places too much of a burden on the system administrators because they may not really understand the semantics of application operations. We seek to develop an access control model in which system security policy can be enforced, but less-trusted principals (e.g., application developers and users) can still make security decisions within a restricted space.

In recent years, much activity has focused on the creation of "safe" languages and security infrastructure that enables control of content written in such languages [EC 1999; Dorward et al. 1996; Gallo 1996; Gong 1997b; Grimm and Bershad 1998; Hagimont and Ismail 1997; Hawblitzel et al. 1998; Islam et al. 19997; Jaeger et al. 1996; Ousterhout et al. 1998; CNRI 1998]. These systems define an access control model, an authorization mechanism, and, sometimes, mechanisms for maintaining permissions as the content executes. Originally, interpreters were either designed to "sandbox" untrusted programs [Sun Microsystems 1999; Netscape 1999], so like the early collaborative systems above, few applications could be built. Then, systems were developed in which access control requirements could be specified per module (within a single hardware-protected process) and enforced on intermodule operations [Ousterhout et al. 1998; CNRI 1998; White 1995]. These systems lack a comprehensive security infrastructure that uses these mechanisms, however, so application security must be developed ad hoc. Since then, several researchers have focused on creating comprehensive security architectures [Islam et al. 1997; Gong 1997b; Grimm and Bershad 1998; Hagimont and Ismail 1997; Jaeger et al. 1996], but the JDK 1.2 security architecture is the best-known and notable. There are two important features of this architecture: (1) access control policies are associated with a class and its provider and (2) the permissions of each Java thread are the intersection of the permissions of classes whose methods have been invoked by the thread (called *stack introspection*

[Wallach and Felten 1998]).[1] So that each class functions properly in any situation, the permissions assigned to it must be the union of all the rights that the class will ever need. Therefore, the UARC application cannot be guaranteed that the rights it must enforce are consistent with its access control policy. The Java system itself has another weakness: authorization requires that explicit calls to a security manager must be coded by the class developer.

A variety of innovative access control models and authorization mechanisms have been implemented for operating systems, although most are research or niche systems [Gasser and McDermott 1990; Wobber et al. 1994; Trusted Information Systems, Inc. 1994]. A comprehensive access control architecture is implemented in the Taos operating system [Lampson et al. 1992; Wobber et al. 1994]. This system's access control model enables principals to be composed from base principals, the roles they may assume, and delegations. This access control model has tremendous power, but it is difficult to create principals whose permissions depend on an application state because: (1) it is cumbersome to define roles or delegations for each state change and (2) management of access control lists becomes prohibitive with all these artificial roles and delegations [Boebert and Kain 1985]. In another approach, TMach adopted Boebert and Kain's Domain and Type Enforcement (DTE) access control model [Trusted Information Systems, Inc. 1994] to Mach. But DTE does not handle principals whose permissions change with application state because domain labels are context-sensitive. A flexible authorization mechanism is proposed in the Distributed Trusted Operating System (DTOS) variant of Mach [Minear 1995]. Here, capabilities for any principal may be cached in the kernel and may be managed by one of potentially many security servers. The kernel checks communication capabilities on IPC, and system servers must call the kernel to check system security policy. The degree to which servers control authorization is a point of contention because servers must be trusted to manage their object spaces and operation semantics, but letting servers determine whether to authorize an operation may violate the requirement for a secure monitor [Anderson 1972] that states that monitors must completely mediate all operations.

Some distributed systems also define interesting access control systems. The CRISIS system control access to web-based file systems using a combination of base rights, represented using a access control list (ACLs) and dynamically obtained rights represented using capabilities [Belani et al. 1998]. Security policy for managing the dynamic distribution of capabilities is not specified, however. Restriction of capability delegation is necessary to prevent a principal from violating the system security policy. Also, CRISIS security managers cannot enforce immediate revocation (timeouts are used). The Moses system authorizes each communication using the current security policy [Minsky and Ungureanu 1998]. Since all communication is mediated, different security policies can be supported for different

[1]Actually, a class may re-establish its full privileges using a command provided in JDK 1.2.

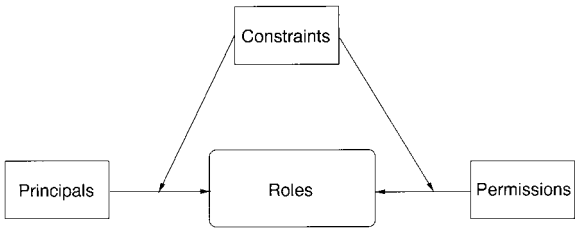

Fig. 2. Fundamental RBAC Concepts: Roles associate the principals that can assume them with the permissions assigned to the role. Constraints may limit either the assignment of principals to roles or the assignment of permissions to roles.

principals and immediate revocation is possible. Like CRISIS, Moses does not support the security policies for managing permissions.

4. REPRESENTING ACCESS RIGHTS

The fundamental concepts in our system are the access control model and the access control enforcement mechanisms. In this section, we detail the access control model. In the subsequent section, we examine the system mechanisms that enforce the policy specified using the model.

4.1 Role-Based Access Control

Our access control model is derived from role-based access control (RBAC) models [Sandhu et al. 1996]. Figure 2 shows the core relationships embodied in an RBAC model. Fundamentally, a role-based access control model permits (1) the aggregation of principals into the roles that they can assume and (2) the aggregation of permissions into the roles that are granted those permissions. This enables a principal to assume a role that is commensurate with the rights needed for a task, so a principal can be restricted to least privilege rights. In the UARC example, we want to define roles that are assigned only those privileges needed for a participant in the UARC application.

As shown in Figure 2, the assignment of principals to roles and permissions to roles may be restricted by constraints. Constraints are necessary to ensure that role specification can be restricted to obey system security policy. For example, if separation of privileges between two roles is required by the system security policy, a constraint can be used to enforce that restriction. It is also possible to prevent principals from being able to write objects that may be executed. The problem is the specification of these constraints. At present, constraint specification is not a well-developed field in RBAC. Bertino et al. [1999] define a permission constraint language for workflow systems. While the language appears fairly complete, it specifies constraints at the task-level, not the permission-level, so it cannot express the requirements stated above. In UARC, we wish to prevent all novice collaborators from obtaining any write permissions to

any objects, and we want to restrict UARC scientists to reading and writing of UARC objects only.

RBAC also eases permission management through permission aggregation and inheritance. Aggregations of objects into object groups and operations into operation groups are supported in some access control models [Boebert and Kain 1985; Jaeger et al. 1996; Karjoth 1998]. [2] We envision using object grouping to define a set of UARC-specific objects that may be available. Another important form of aggregation is defined using context-sensitive roles [Giuri and Iglio 1997; Lupu and Sloman 1997]. In a context-sensitive role, permissions may be parameterized, such that a single specification may apply in multiple contexts. This is useful to the UARC application because different sessions may require that different objects be shared. The UARC application can specify which objects are part of a session, and a context-sensitive role limits the rights of the principals in that role to the objects associated with that session.

RBAC models use a variety of role hierarchies to ease management of permissions. First, a traditional role hierarchy enables superior roles to inherit the rights of inferior roles. This hierarchy implements a strict set-subset relationship on the rights between the superior and inferior roles. Second, ARBAC97 [Sandhu et al. 1999] includes an administrative role hierarchy in which the administrative rights of principals to modify role definitions in the role hierarchy can be specified. Also, constraints can be specified on the actions that administrative roles can perform. In UARC, it is useful to permit users and UARC applications to modify the collaborator role definitions. While this relationship should be embodied by a administrative role hierarchy, in this paper we focus primarily on constraining the extent of the administrative modifications that are permitted. Third, specification of the roles that a principal may activate can be specified in a role activation hierarchy. Sandhu describes the need for role activation hierarchies to enforce separation of privilege [Sandhu et al. 1994]. In the UARC example, not only do we need to know what roles can be activated by a principal, but we need to know which role should be activated, given particular content. We refer to this mechanism as *role selection*, and we define a role selection hierarchy [Jaeger et al. 1997]. The problem is to determine the appropriate role to be assigned to a particular downloaded content, given its authenticated content description (e.g., a set of attribute-value pairs). A role selection hierarchy is defined as (1) a graph of nodes where each node maps a set of attributes and the sets of values that they may assume to form a role and (2) a search algorithm that determines how to find the matching node given a content description. We define a particular instance of a role selection hierarchy in our access control model below (see Figure 5).

[2]Despite the fact that not all of these models are considered to be explicitly RBAC models, we consider this feature fundamental to RBAC models.

4.2 Access Control Model Overview

We make four observations about the practical difficulty of building an access control model for applications composed of downloaded executable content:

—delegations from multiple untrusted principals may be necessary;

—least privilege permissions may depend on both system and application factors (e.g., application state);

—remote principals may have the knowledge to express access control requirements of their applications, but they do not know the downloading principal's system;

—it is possible to describe application access control requirements across multiple sessions.

First, we observed that the UARC application and collaborator content may be assigned permissions from multiple principals, such as system administrators, downloading principals, and the UARC application itself. While system administrators are trusted completely in our system, they may not have the application knowledge necessary to properly restrict the collaborator content. Second, we observed that UARC application and collaborator content permissions may depend on a number of factors, including the content provider, downloading principal, application state, and the content's role in the application. For example, the files that are to be granted to collaborator content depend on the downloading principal using the UARC application and the set of files that this principal is sharing. Third, although the UARC application may want to specify its collaborator content's permissions, the UARC application developers probably do not know the organization of the downloading principal's system. The names and locations of UARC recording and annotation files stored on a downloading principal's system are almost certainly not known by the UARC application developers. Lastly, at a certain level of abstraction, the security requirements of applications do not change significantly between each run. For example, each execution of the UARC application requires that only the files explicitly opened for sharing by a downloading principal (i.e., collaborator) may be shared. This knowledge should be applied to each execution of the application.

We define an access control model that employs role hierarchies, mandatory access control limits, context-sensitive roles, and declarative permission transforms to address these issues. First, role hierarchies are defined for role administration and role selection, as described in Section 4.1. For example, the role administration hierarchy specifies that system administrators may specify the UARC application role. The role selection hierarchy determines which role definition corresponds to which content principal. The administrators of a role may specify (1) the principals that may assume that role and (2) a set of mandatory access control limits on the permissions that a role may obtain, called *transform limits*. A transform limit associates

a delegator, role, and the permissions that the delegator may assign to the role. Thus, system administrators may specify transform limits, which permit users to delegate rights to read and modify their UARC files to the UARC application, for the UARC application role. However, the extent to which users can manage the permissions of the UARC application can be restricted to prevent security breaches. The goal is to ensure that process confinement is possible (e.g., separation of duty), viruses are prevented (e.g., by preventing write access to executable objects), and Trojan horses are contained (e.g., via limited rights). Using transform limits, the maximal permissions that any principal may obtain are restricted to the union of their transform limits.

Context-sensitive roles enable delegators to associate permissions with principals on the basis of the principal's context. A context-sensitive role associates parameters with a role name and permissions. Thus, the value of the parameters determines the particular permissions available to the role. We use this concept to address two of the issues above: (1) to permit delegators to choose delegations based on the current application state and (2) to permit permissions to be specified abstractly and bound to the current downloading principal's state. In the first case, the UARC application can restrict its collaborator content to the objects currently being shared. In the second case, the UARC application can specify permissions that apply to all its users because the system administrators and users can define the meaning of application specific-object groups. For example, the identity of the specific files that may be made available to the UARC application may be determined by users (within transform limits specified by system administrators).

Lastly, determining which permissions should be associated with what context depends on the operations being executed. Therefore, the access control model defines *transforms* that associate operations with permission transformations. When an operation is invoked, any transforms associated with that operation are executed. They may then add or remove the permissions as specified in the transform. For example, when the user loads a recording into the UARC application, the need to add a permission to collaborator roles is triggered, so they may properly access this recording and its annotations.

4.3 Access Control Model Definitions

We now formally define the security architecture's access control model concepts; the structure of the access control model is shown in Figure 3.

—**Identity**: A set of attribute-value pairs;

—**Type**: A relation between a type name and a set of operations that can be invoked on the type;

—**Object**: A relation between a type and a unique object identifier;

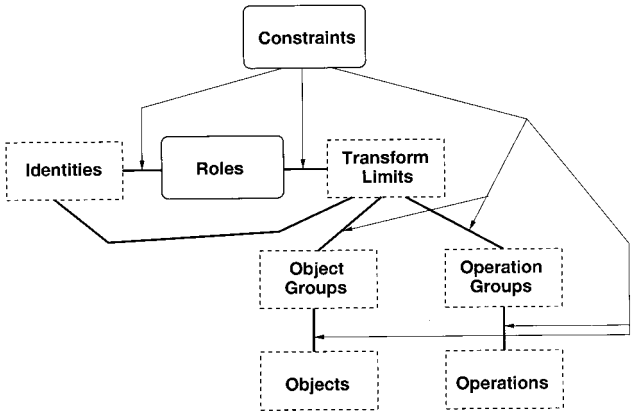

Fig. 3. Extended role-based access control model: The base RBAC model is extended. First, identities are mapped to roles, so a role may be used for multiple contexts. Second, a role is associated with transform limits (specified in terms of object groups and operation groups and the identities that may delegate rights within those limits) that determine the permissions that the role can ever obtain. Transforms are used to grant access to permissions within transform limits.

—**Object group**: A function over a name, a set of identity parameters, and a type that returns a set of objects;

—**Operation group**: A relation between a group name, a type, and a set of operations that correspond to the name and type;

—**Permissions**: A relation between a permission type (positive or negative), an object group and operation groups that describe the operations that can be performed (or precluded) on the objects in the object group;

—**Transform**: A relation between an operation and the permission changes that its execution triggers;

—**Transform limits**: A relation between a delegator identity, delegatee identity, and the permission set within which the delegator may grant rights to the delegatee;

—**Role**: A relation between an identity and its transform limits;

—**Role administration hierarchy**: A graph consisting of nodes that associate administrators with the roles that they may modify;

—**Role selection hierarchy**: A graph consisting of nodes that associate identities with the roles that they must be assigned to;

—**Principal**: A relation between an identity, a set of transforms, a set of transforms limits, and a set of its current permissions

The context of each role is determined by its identity. An identity representation must be flexible enough to express the relevant, unique execution contexts of the system. In some systems, an identity is reference

to a public key. However, more information than simply a provider's public key may be useful in defining a content's permissions. As described in the problem statement, UARC content may have different permissions based on its provider, downloading principal, application, application role, and application state. Thus, the access control model includes two sets of identity attributes:

—**Authentication**: Content provider (e.g., public key), content digest;

—**Context**: Downloading principal, application, application role, application instance identifier, download nonce.

Authentication identity attributes uniquely identify the content. Thus, the values of these attributes may be reused if the same content is loaded into another execution context. Context attributes identify a potentially unique execution context for the content. For example, the same content may be run by the same downloading principal in two separate protection domains, and this identity representation enables the specification of two different identities: one for each protection domain instance.

Objects are strongly-typed entities with unique and immutable names. First, an object's type is defined by its interface and implementation. An interface defines the operations (i.e., methods) that can be invoked on the object. One object may provide access to a set of other objects that it manages (i.e., act as an object server). For example, consider a file system. The file system object serves file objects. Logically, the *open* operation is called on the file system, but *read* and *write* operations are invoked on files. Second, we assume that objects have unique names within their name space. Each object server is responsible for the association of names with data and is trusted to maintain this association properly. For example, a file system is trusted to return, when requested. the correct data in file `foo`. Third, an object's name and name space location are immutable. Changing an object's name or moving an object to another location in the name space requires creation of a new object. This requirement and name uniqueness aids in preventing time-of-check-to-time-of-use (TOCTTOU) attacks [Bishop and Dilger 1996] because an object cannot be replaced by another object with different permissions. Lastly, object names are arranged in a tree structure. Thus, links are not provided by the basic object name space. It may be desirable to construct a more traditional name space (e.g., UNIX path names with links) on top of this; but this is beyond the scope of this paper.

The access control model supports the aggregation of objects and operations. The name space model supports the aggregation of objects into object group identifiers. Formally, an object group identifier is a tuple: $name(t, a1, a2, \ldots)$ where: (1) $name$ is the name of the object group; (2) t is the object group's type; and (3) $a1$ and $a2$ are object group parameters. Since an identity determines the role's context, identity attributes are used as object group parameters. For example, object groups associated with a

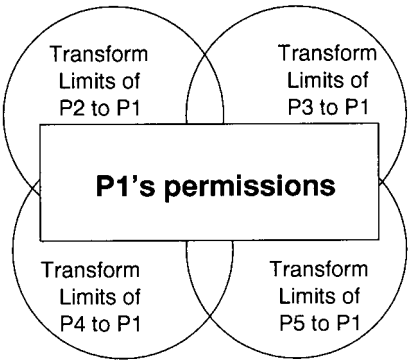

Fig. 4. Principal P1's permissions and its transform limits: The transform limits specify that principals P2-P5 may delegate limited rights to P12.

particular UARC session include the application instance identifier attribute as a parameter.

Operation aggregations simply permit a set of operations to be aggregated into a higher level operation. This enables application developers to map their objects operations to system policy-specific operations. Many authorization systems use operation aggregation, including CORBA [Object Management Group 1997] and NAPOLEAN [Thomsen et al. 1998].

Permissions define the operations that can be performed on object groups. A permission may be either positive or negative (called the *permission type*), which specifies whether the permission is, respectively, allowed or precluded. Negative permissions supersede positive ones in authorization. That is, a negative permission prevents access to an operation, regardless of the positive permissions that the principal may possess. Negative permissions are valuable for expressing exceptions to positive permissions over an object group. For example, access to all but one object in an object group may be specified by one positive and one negative permission. In general, any permission set can be represented solely by positive permissions.

The access control model includes the concept of a *transform*, which associates an operation with the permission changes that its execution triggers. Permission changes are relations between change operations, add or remove, and the permission change specifications. We define the exact structure of permission change specifications in Section 5.4. Researchers working on the J-Kernel project [Hawblitzel et al. 1998] are also concerned about the proper revocation of delegated permissions (e.g., when the server terminates). We are hopeful that revocations can also be represented as transforms. An operation corresponds to a revocation event, such that the associated transforms remove the appropriate operations.

In our access control model, a role is an association between an identity and its transform limits. Transform limits associate delegators with the permissions that they may delegate to the role. Transform limits are

therefore used to authorize transforms. A transform may add a permission only if (1) the delegator has the permission and (2) the permission is within the transform limits of the delegator for the delegatee. Also, a principal can only remove a permission that it is able to delegate. The result is that a principal's permissions are always a subset of the union of its transform limit, as shown in Figure 4.

Transform limits alone do not ensure that a system's security requirements are met. While transform limits can enforce a separation of duty requirements, the fact that separation of duty is required between two roles must also be expressed. Constraint specification is a largely open problem in RBAC (as described in Section 4.1), so this access control model does not include a constraint specification language. However, in Section 4.4, we describe how static and dynamic separation of duty constraints can be enforced using this access control model.

This access control model provides hierarchical representations for role management, role administration, and role selection. Traditional role management and administration hierarchies are used [Sandhu 1999]. Role management hierarchies are set-subset hierarchies with respect to transform limits. We discuss role administration in terms of system administrators, users, and applications. In general, system administrators limit the permissions that the others may distribute. The actual determination of the system administrator roles that correspond to different downloading principals is determined by the role administration hierarchy. This assignment is not particularly relevant in the UARC example, so we simply refer to system administrators.

As we described above, role selection is more complex in our example because a variety of identities may be assigned to content principals, and each of these may be assigned different permissions.

Fortunately, we can define a total order on the identity attributes, such that a role selection hierarchy that associates transform limits (which define a role) with identity attributes can be created. In our model, the role selection graph consists of five levels: (1) downloading principal; (2) content provider; (3) application; (4) application role; and (5) protection domain instance. Each node may refer to an attribute value, a role, and a set of child nodes. The attribute value may indicate a specific value or a group of values (with * indicating all). At each level, identity is compared to the attribute values in the role selection hierarchy and the closest matching value is selected. It is assumed that a close match at downloading a principal is more important than a close match at the lower levels, such as application and application role. We believe this is true for our UARC example, but it is a heuristic.

As an example, consider the three-level role selection hierarchy in Figure 5. This role selection hierarchy assumes a specific set of downloading principals. Given identity attribute values of `provider=UM` and `application=UARC`, the transform limit TL_U is found. Note that the role selection hierarchy enables system administrators to limit the content

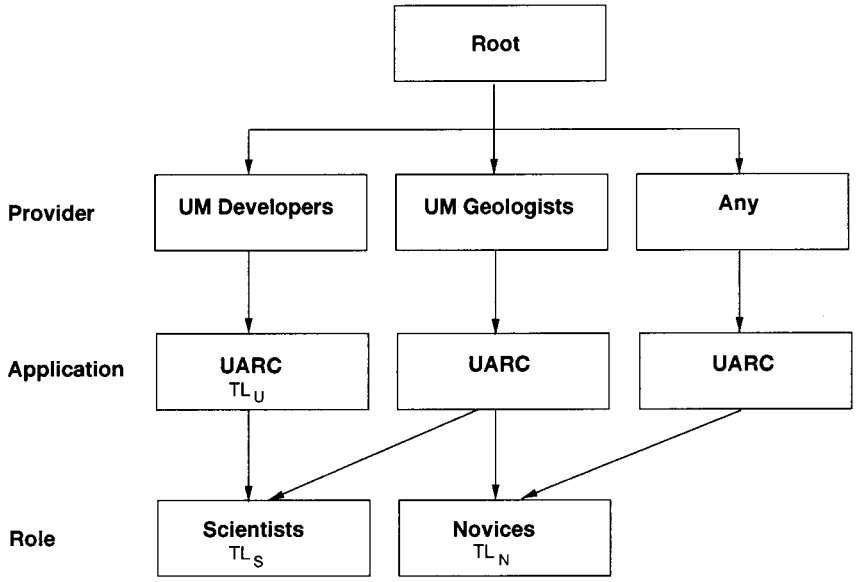

Fig. 5. Role selection hierarchy: Defines transform limits for UARC application content and content executing in the scientist and novice roles. Note that only the University of Michigan can provide UARC application content, and only the UM geologists can assume the scientist role in the specified policy graph.

providers that can assume specific roles. For example, the UARC application can define its transform limits for UARC role content (scientist and novice). However, since the system administrators build the role selection hierarchy, they can set the links that associate UARC collaborators with those roles. So `UM geologists` can assume the role of scientists (transform limits of TL_S), but others can only be assigned to a novice role (transform limits of TL_N).

4.4 Enforcing Separation of Duty

We give a brief overview of how separation-of-duty security requirements can be enforced using this access control model. For static separation of duty, roles that obey the requirements of a static separation of duty can be defined. A role defines a set of transform limits, and the rights that the principal may have at any time are a subset of these limits. Since, due to the separation of duty, transform limits have no shared rights (the roles are in static separation of duty), the principals share no rights. The role selection hierarchy defines which roles correspond to what content identities. Therefore, in order to ensure that two content identities are in static separation of duty, the roles they are assigned must obey static separation of duty.

A dynamic separation of duty is enforced as follows: Consider the Chinese Wall security policy [Brewer and Nash 1989]. This policy specifies

that a principal may have only one set of permissions out of many. If the principal chooses an object in one set, then access to the objects in the other sets is no longer permitted. Links from a specific node in the role selection hierarchy may be defined to be in dynamic separation of duty. Therefore, once one link is taken, the others are invalidated. Since the nodes connected by links are in separation of duty and only one of the nodes may be selected, the Chinese Wall policy may be enforced.

5. ARCHITECTURE

The goal of this architecture is to provide services for users to compose and execute applications from downloaded executable content such that the system's and application's security requirements can be enforced. The architecture is designed to support the access control model defined in the previous section. It provides services to determine the content's identity, determine the appropriate role for content, manage content permissions within the role's transform limits, and authorize content operations using these permissions.

In the design of an architecture for controlling downloaded executable content, we make the following assumptions. First, we assume that content has access to a well-defined set of commands only, to perform system I/O or other operations that need to be controlled. Second, the system's trusted computing base (TCB) provides address space separation among processes, a means for identifying processes, and cryptographic services for authenticating remote messages. Without trust in the TCB, it is not possible to build trusted applications that run on it. A secure booting mechanism, such as provided in Trusted Mach [Trusted Information Systems, Inc. 1994], is designed to ensure that the proper TCB is booted when the system is started. However, other means sufficient for establishing trust in the TCB may be possible.

The trust model of our system is defined from the user's perspective. First, users trust a process we'll call the *client* to make requests to download content on their behalf. The client trusts our *loader interface* to implement this request in such a way that the user's security requirements can be satisfied when the downloaded content is initiated. The loader interface trusts a set of principals designated as *certifying authorities*. The clients trust *object servers* to handle the objects they manage properly. For example, a file system is expected to perform file operations correctly. The clients trust *content providers* with the rights they are granted. However, a content provider may try to obtain unauthorized access rights using its legitimate rights and all means available to an attacker (e.g., read and modify network traffic).

The architecture is shown in Figure 6. It provides an interface for any downloading principals (clients or content itself) to download executable content, called the *loader interface* (LI). The LI is supported by four services: (1) an authentication service; (2) a derivation service; (3) a

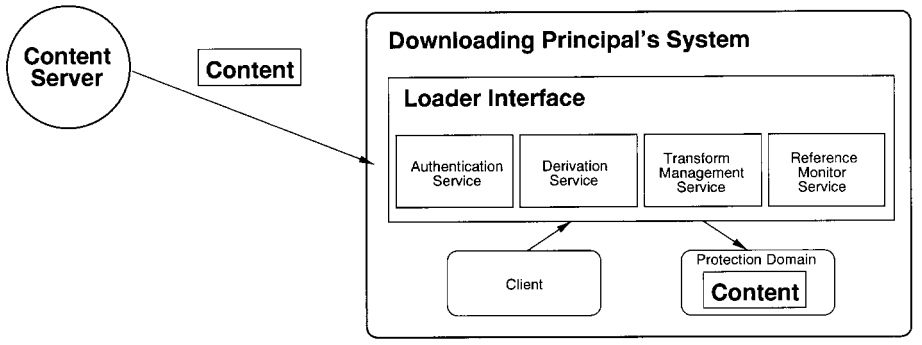

Fig. 6. System architecture: Downloading principal requests that the *loader interface* (LI) load content from the content server. The LI retrieves content, authenticates it, derives its permissions, and loads it into a protection domain in which its permissions can be managed and enforced.

transform management service; and (4) a reference monitor service (RMS). The interaction between the architecture components is shown in Figure 7. The downloading principal requests that content be loaded by calling LI. The LI uses an *authentication service* (AS) that provides the cryptographic operations necessary to determine the authenticated identity of the content principal. LI uses the *derivation service* (DS) to determine the role for that content principal and obtain the content principal's transform limits. Management of permissions during the content's execution is controlled by the *transform management service* (TMS). The TMS authorizes the execution of transforms with respect to the content principal's transform limits. LI uses the *reference monitor service* (RMS) to authorize content operations using that principal's permissions. In the following sections, we describe the design of each of these services and how these designs support the UARC application.

5.1 Loader Interface

The loader interface (LI) provides all principals with an interface to load executable content, such that the system's and application's security requirements can be enforced during its execution. In our example, clients first use the LI to download the UARC application content. Next, the UARC application uses the LI to retrieve outside content and load any collaborator content sent to the UARC application.

The LI API shown below enables the downloading principal to examine the status of each step in the load, so that they may make adjustments should a step be rejected by the LI. For example, if the LI will only grant the content a subset of the permissions required (determined by *derive*), then the downloading principal may load another content that provides similar functionality instead. Also, "all-in-one" methods *retrieve and load* and *set and load* are provided for simple default loading.

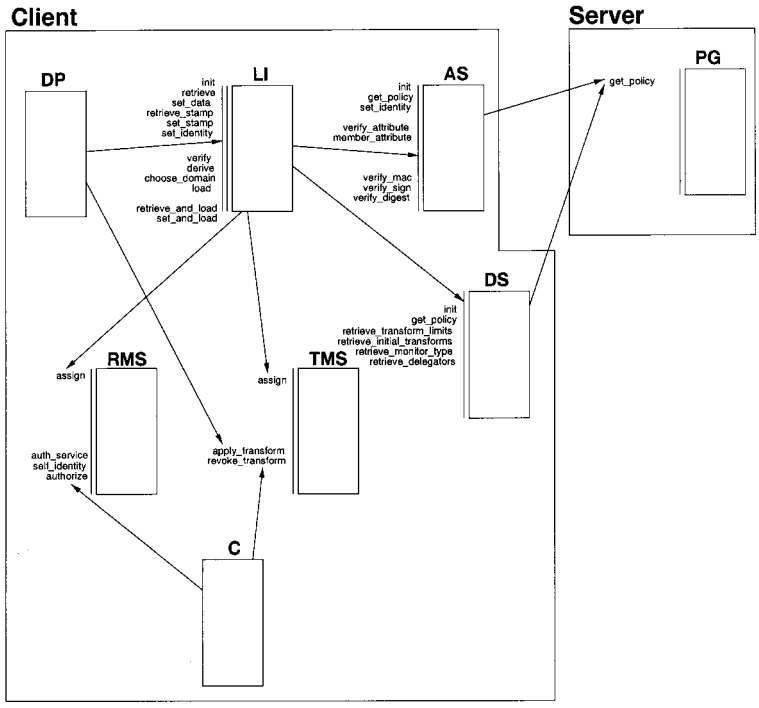

Fig. 7. System architecture APIs: The loader interface (LI) enables the downloading principal (DP) to load content (C). The authentication service (AS) and derivation service (DS) use the server's policy database (PG) to retrieve authentication and derivation policies, respectively. The reference monitor service (RMS) and transform management service (TMS) control execution of loaded content.

```
int init (in ulong comp_gid);
int retrieve (in ulong loadid, in string location_hint);
int set_data(in ulong loadid, in string data);
int retrieve_stamp(in ulong loadid, in string location_hint);
int set_stamp(in ulong loadid, in string stamp);
int set_identity(in ulong loadid, in string identity);
ulong verify(in ulong id);
ulong derive(in ulong id);
int choose_domain(in ulong loadid, in ulong domain, in permis-
sions required_perms);
ulong load(in ulong loadid, out ulong domain);
ulong retrieve_and_load(in string location_hint, in string
stamp_location_hint, in string identity, in int option, in int
modify_perms_p, out ulong intf);
ulong retrieve_and_load(in string location_hint, in string
stamp_location_hint);
ulong set_and_load(in string data, in string stamp, in string
identity, in int option, int modify_perms_p, out ulong intf);
```

The first method, *init* provides the downloading principal with a handle, called a *loadid*. The loadid is a reference to a *load request* object managed by the LI. A load request stores the status of a secure load. The LI requires

that it can obtain the identity of the downloading principal securely (e.g., by the interpreter providing a thread identifier). Only the LI has direct access to the load request data. The loader request data is as follows:

—The downloading principal;

—the content data;

—a content stamp (see below);

—a requested content identity;

—the target domain (e.g., process);

—a content principal (maps the identity to permissions and transform limits).

Content may be downloaded from the network or file system (via *retrieve*) or uploaded from memory (via *set data*). In the case of a *retrieve*, the downloading principal must locate and retrieve the content from a server using some global object retrieval mechanism. We leverage existing work in this area, such as using URLs to reference World-Wide Web (WWW) objects. Using *set data*, the downloading principal provides the content directly.

The content provider, or other sufficiently trusted principal, may create a signed description of the content, called a *content stamp*. A content stamp's description specifies the identity of the content's principal and includes information necessary for the AS (see Section 5.2) to authenticate that identity (e.g., including public key certificates). Content stamps are retrieved or uploaded to the LI using *retrieve stamp* and *set stamp*, respectively.

The method *set identity* enables the downloading principal to annotate the identity required for the content. The LI derives an identity from the information in the content stamp. *Set identity* enables the downloading principal to provide additional requirements for the identity. For example, the downloading principal may wish to assign the content to a specific role. Values for any subset of the identity fields (see Section 4) may be specified. If the stamp provider and the downloading principal specify contradictory identity requirements, then the content cannot be authenticated. The identity is used by the AS to retrieve authentication policy to verify that the content can assume that identity.

The *verify* method uses the content stamp to verify that the content can be executed according to the specified identity (using the AS). The result of *verify* is that an identity is assigned to the content (in its newly created principal).

The *derive* method determines the content principal's role and transform limits. There may already be a principal associated with this identity, so the method first determines if permissions need to be derived. If not, the LI calls the DS interface (see Section 5.3). The RMS and TMS renforce and manage, respectively, the permissions assigned to the principal.

The LI may load content into a new or existing protection domain (e.g., process). For example, outside content that implements trusted libraries (e.g., libc) may be loaded with UARC application content. The *choose domain* method enables the downloading principal to load the content into an existing protection domain. Content can be loaded into a protection domain if the following security requirements are satisfied:

—The principal associated with the target protection domain is the downloading principal;

—If content is to be loaded into an existing protection domain, the intersection of the rights of the content principal and the existing domain does not result in the loss of rights required by the existing domain;

—Neither downloading the principal or the content principal gains any unauthorized rights due to sharing the domain;

—Neither downloading the principal or the content principal gains unauthorized access to secrets stored within the other's portion of the domain.

First, downloading principals may only load content into their own protection domains. Loading content into another principal's protection domain can be used to attack that other principal, so it is not permitted. The second requirement states that the existing principal may specify that certain required rights be preserved before the content load is permitted (the *required perms* argument of the *choose domain*). The third requirement is enforced because when content is to be loaded into an existing protection domain, the downloading and content principals are merged, such that the permissions and transform limits of the two principals are intersected. Thus, no unauthorized permissions are made available to the merged content or the downloading principal, but the content may delegate rights to others if it still possesses them. Lastly, content with secret information may not be merged with other content, unless the other content is trusted. Obviously, all information within the process's address space will be accessible to both content programs, so secrets cannot reliably be kept from colocated content. Whether content has secrets that need to be protected is specified in the content's stamp.

The *load* method loads the content into the specified protection domain. This method returns a reference (e.g., entry capability), so that the downloading principal may call the newly loaded content.

For UARC, the loader interface is used in the following manner to load UARC application and collaborator content.

—**UARC application**
  —`id = init()`: obtain the load identifier `id`
  —`retrieve(id, UARC-URL)`: retrieve the UARC content from the UARC web server
  —`retrieve_stamp(id, UARC-stamp-URL)`: retrieve the UARC stamp from the `UARC-stamp-URL`

—`set_identity(id, "application=UARC")`: set identity context field application to UARC
—`verify(id)`: verify the content against its authentication requirements
—`derive(id)`: determine the role and derive the transform limits for the UARC application
—`new_domain = choose_domain(id, null, true)`: create a new domain for the content
—`load(id, new_domain)`: load the UARC application content into the new domain

—**Collaborator content**
—`id = init()`: obtain the load identifier `id`
—`set_data(id, collaborator content)`: upload the content provided by the collaborator
—`set_stamp(id, stamp)`: upload the content stamp provided by the collaborator
—`set_identity(id, "application=UARC; role=collab; appl_inst =session")`: state the identity context fields to be for the UARC application, the collaborator's role, and the UARC session
—`verify(id)`: verify the content against its authentication requirements
—`derive(id)`: determine the role and derive the transform limits for the UARC collaborator content, if necessary
—`collab_domain = choose_domain(id, null, true)`: create a new domain for the collaborator content
—`load(id, collab_domain)`: load the UARC collaborator content into its new domain

The client principal uses the LI API to request that the UARC application content and stamp be retrieved from the UARC web server. The identity's application attribute for the content is set to UARC by the downloading principal to ensure that the content is intended for the UARC application. After authentication and derivation, the content is loaded into a new domain. Note that the default operation *retrieve and load* could have been used to implement this download.

The download for outside content (e.g., the statistical analysis package) is not shown, but proceeds using essentially the same calls, except that the UARC application may load the content into its own domain. The UARC application content could check the transform limits resulting from the *derive* operation and determine whether colocation of this content is possible without modifying the content's required access rights (third argument to *choose domain*).

UARC collaborator content is sent to the UARC application content by collaborators. Therefore, the UARC application uses *set data* and *set stamp* to initiate the load. In this case, UARC sets the following identity attributes: (1) application is set to UARC; (2) the application role is set to the collaborator's role; and (3) the application instance is set to the UARC

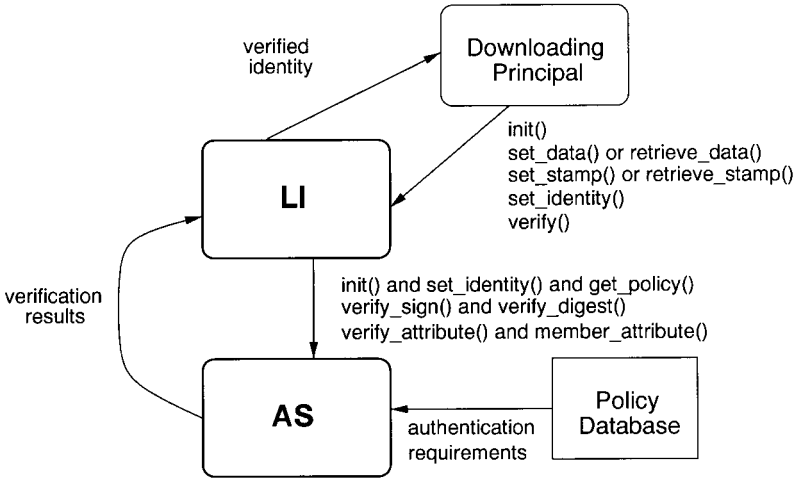

Fig. 8. Authentication service protocol: The content stamp (set by *set stamp* or *retrieve stamp*) and identity requirements specified by the downloading principal (using *set identity*) are combined into a proposed identity. The *verify* command triggers the LI to initiate authentication of the proposed identity with the AS. LI uses the AS to retrieve authentication policy and verify that the content and content stamp are sufficient for the content to assume the proposed identity.

application instance identifier (e.g., process id). These are in addition to the downloading principal and content provider. If this collaborator has downloaded content before, the content identity maps it to active authentication requirements and an active principal (with a current set of rights). The collaborator content is loaded into a new domain for the collaborator.

5.2 Authentication Service

The authentication service's (AS) goal is to prove that downloaded content meets the criteria necessary for it to assume a specific identity. As shown in Figure 8, the AS uses a proposed content identity derived from the content stamp and the downloading principal's inputs set using commands *set/retrieve stamp* and *set identity*, respectively, to retrieve authentication requirements from the authentication policy database. This policy specifies the tests that must be met before the content can be assigned to a principal with that identity.

We show how the AS works by describing how it is used to verify the authenticity of UARC application, outside, and collaborator content. UARC application and outside content have the same authentication requirements, listed below.

(1) must have a valid signature from a trusted principal or the content provider for the content and its stamp;

(2) must be unmodified;

(3) must be authored by a principal authorized to write UARC application or outside content;

(4) must be claimed to be the requested content (UARC application or the specific outside content); and

(5) must be an acceptable version.

For collaborator content to be executed, the download server must ensure that it

(1) must have a valid signature from a trusted principal or content provider for the content and its description;

(2) must be unmodified,

(3) must be fresh (i.e., not a replay);

(4) must be for the UARC application;

(5) must be authored by a principal trusted to assume the requested UARC role;

(6) must be authored by an active collaborator in this session;

(7) must be for a UARC role; and

(8) must be for the particular UARC session (i.e., application instance).

Note that some of the requirements for the collaborator content depend on the state of the UARC application. For example, collaborator content must be signed by an active UARC collaborator. The collaborator content also has one extra authentication requirement: that it be fresh (i.e., not a replay of a prior message). Freshness is an important requirement because an unauthorized replay may cause the divergence of a collaborator state. If the collaborators believe they are looking at the same application state, but are not, incorrect results may be generated.

The LI employs the following approach in authenticating a content identity using the AS. First, it composes a proposed content identity from the input of the content provider and downloading principal. The proposed identity is the union of the identity fields specified by the two principals. This approach enables a content provider to propose an identity for its content (in the content stamp) and for the downloading principal to annotate this identity on the basis of the expected use of the content (using *set identity*). The proposed identity is used to retrieve the system's authentication requirements. The role selection hierarchy is used to store authentication requirements using the proposed identity. The LI then uses the AS to verify that the content satisfies each of the requirements before the content may assume that identity. The LI can verify that the content is authorized by the system to assume the proposed identity.

The AS uses a *content stamp* to obtain the content provider's proposed identity for the content. The structure of a content stamp is shown in Table I. In general, a content stamp is a list of attribute-value pairs that describe

Table I. Example Content Attributes and Values for a Content Stamp

Categories	*Attributes*	*Examples*
Authentication	Signature	DSA signature [NIST 1994]
	MAC	MMH hash [Halevi and Krawczyk 1997]
	Digest	SHA-1 digest [NIST 1995]
	Nonce	Sequence number
Descriptive	Signer	Rating service
	Provider	IBM
	Name	UARC
	Role	Active
	Application ID	UARC ID
	Version Range	1.0
Application	Permissions	Requested initial permissions
	Transforms	Load grants RW to Active
	Secrets	Yes/No
	Role Specs	Roles and their transform limits
	Role Key Distribution	SSL 3.0 [Freier et al. 1996]

the different aspects of content: (1) its authentication data; (2) its descriptive data; and (3) its application-specific security information, to be interpreted by the LI. Authentication attributes such as *signature*, *digest*, *MAC*, and *nonce* enable the verification of the source, integrity, and freshness of the UARC content. For example, the signature enables verification of the source and integrity of the content stamp itself. Descriptive attributes such as *signer*, *manufacturer*, *type*, *name*, *role*, *application ID*, and *version range* describe the source of the content and its intended use. For example, UARC collaborator content should indicate that it is for the UARC application (name), that it implements a specific role within the UARC application (role), and that it is for a specific application instance. Application attributes provide application-specific information to the LI. The content stamp may include a request for a set of permissions to be granted upon download to enable it to execute (*permissions*) as well as the transforms that are to be associated with operations (*transforms*). The content stamp may also be used to specify roles for the application, including their transform limits (*role permissions*). We also expect applications to specify their key distribution requirements for application roles (*role key distribution*), but this discussion is beyond the scope of this paper.

The AS exports the following API to the LI so that it may authenticate the proposed content identity. Other processes may also use this API, but their results do not affect the state of any LI load request.

```
ulong init(in ulong prinid, in ulong dp);
ulong set_identity(in ulong auth_id, in identity_ref identity);
string get_policy(in ulong auth_id, in ulong dp);
int verify_sign(in ulong auth_id, in byte_array data, in
byte_array signature, in string signer, in ulong algorithm);
int verify_digest(in ulong auth_id, in byte_array data, in
byte_array digest, in ulong algorithm);
```

```
int verify_mac(in ulong auth_id, in byte_array data, in byte
_array mac, in ulong algorithm);
int verify_attribute(in ulong auth_id, in string attr, in
string expected, in ulong required);
int member_attribute(in ulong auth_id, in string attr, in
string expecteds, in int ct, in ulong required);
```

Principals use the API in the following manner. First, *init* initiates an authentication session and returns an *auth id* reference for subsequent operations; *set identity* fixes the content identity to be authenticated. The identity is used by the *get policy* method to retrieve the system's authentication policy for the identity from the policy database.

The remaining methods perform the authentication tasks. The API enables verification of authentication requirements for cryptographic data and identity attributes specified in the content stamp. The AS is supported by a cryptographic API (e.g., CDSA [Open Group 1997]) that provides support for certificate, trust, and key management in addition to signature, message authentication code (MAC), and digest verification. *Verify attribute* enables the AS to verify that an attribute has an expected value. *Member attribute* enables the AS to verify that an attribute value is a member of a specified set. Since some attribute values may be optional, these methods enable the policy to specify whether a missing value is acceptable. By setting the *required* argument to true, the content stamp must specify an expected value. Otherwise, if the content stamp specifies a value, the value must be an expected value.

The policy evaluated for the UARC application is shown below:

```
verify_sign(stamp, signature, "UARC-developers", "DSA")
verify_digest(content, content_digest, "SHA-1")
verify_attribute("author", "UM-developers", TRUE)
verify_attribute("name", "UARC", TRUE)
verify_attribute("version", version_number, TRUE)
```

These tests are sufficient to verify the authentication requirements of the UARC application and outside content, as stated at the beginning of this section. We first verify that the signature of the stamp is valid. The signer field in the stamp is used to specify the signer. The AS stores certificates associated with principals (using the crypto API), so that it can retrieve the appropriate public key. Next, the integrity of the downloaded content is verified by computing its digest and checking that it is the same as the content digest in the stamp. Then, we verify that the content stamp fields—author, name, and version—correspond to the expected values provided by the policy.

Content identity for the collaborator content is derived from both the content stamp and the identity requirements set by the downloading principal. The content stamp specifies the collaborator, content digest, nonce, UARC application, collaborator's UARC role, and UARC application identifier. Each identity field specified in the content stamp is used automatically in the identity. In addition, the UARC application acting on behalf of the user (the downloading principal) may also specify the values

of identity fields using the *set identity* method offered by the LI API. In this case, the downloading principal specifies that the collaborator content belongs to a specific role, has a specified nonce (sequence number), and is for a specific application instance. The AS verifies that any identity fields set by the content stamp and downloading principal are consistent. If the downloading principal and content stamp specify contradictory identity values, then the content authentication fails. The AS then retrieves the following authentication requirements from this derived identity.

```
verify_sign(signature, signer, "DSA")
verify_digest(content, digest, "SHA-1")
verify_attribute("name", "UARC", TRUE)
member_attribute("author", role_members(identity.appl_id,
identity.role), count, TRUE)
verify_attribute("role", identity.role, FALSE)
verify_attribute("application ID", identity.appl_id, TRUE)
verify_attribute("nonce", identity.nonce, TRUE)
```

In addition to verifying the integrity and application, as described above, we must verify the content's author, role, application, and freshness. The author is verified by checking that the collaborator can assume the role specified in the identity and is active (i.e., belongs to the *role members* of the application instance). It is not required that the content stamp specify a role for the content, but if one is specified it must be the same as the identity's role. The collaborator must specify an application instance to prevent content from being run in the wrong application. Freshness is verified by comparing the nonce value in the content stamp to the expected nonce specified by the UARC application. Sequence numbers are used to make the nonces predictable, so that verification is possible.

5.3 Derivation Service

Using the access control model defined in Section 4, principals are associated with transform limits within which they may obtain permissions via transforms. The derivation service (DS) provides a mechanism for deriving a principal's transform limits. Also, the DS retrieves and executes initialization transforms which may be specified by off-line principals, such as system administrators and users.

In UARC, there are two types of permission derivations to consider: (1) UARC application and (2) UARC content (both outside and collaborator). First, system administrators must be able to limit the downloading principals' rights that may be delegated to the UARC application, as described in Section 2. To summarize here, the UARC application should be limited to the following permissions:

—read access to UARC recordings of the downloading principal;

—write access to UARC annotations of the downloading principal;

—execute access to supporting applications;

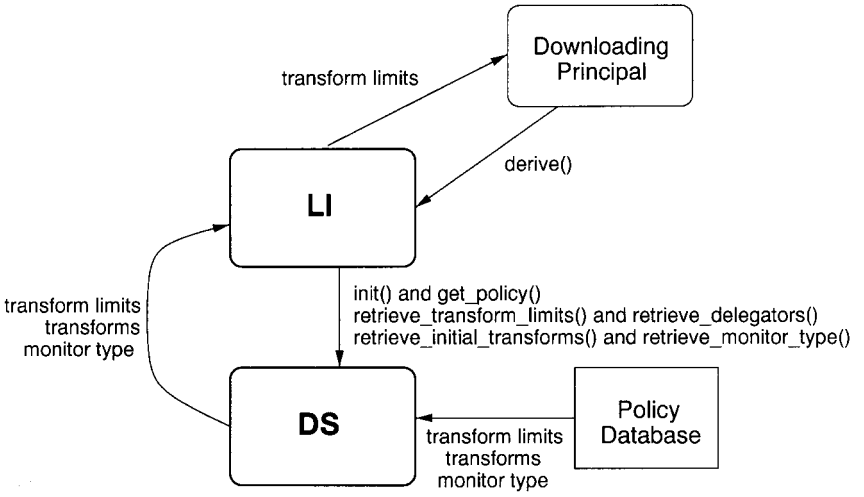

Fig. 9. Derivation service protocol: The loader interface (LI) calls the derivation service (DS) upon a downloading principal's *derive* request. The DS uses content identity to retrieve the content's transform limits and any initial permission delegations. Permissions may be delegated to the content principal within the transform limits.

—communication access to UARC managers, UARC data servers, and UARC participants;

—read access to UARC web information.

System administrators must limit the rights that can be delegated to the UARC application in order to protect the system from compromise, but it is likely that both the users and the UARC application itself know more about which rights are really needed. For example, downloading principals know which recordings they are willing to grant to the UARC application. However, in general, system administrators cannot trust users and applications to manage system permissions properly. Using our system, administrators can specify transform limits for downloading principals and the UARC application within which they may manage the delegation of permissions to the UARC application.

Second, the UARC application must be able to control the permissions made available to a UARC collaborator and outside content in order to ensure its proper execution. For example, recall the two collaborator roles, scientist and novice. Scientists can upload recordings and annotations for replay, trigger saves of recordings and annotations, and manage the display of recording data. Novices can only view the collaboration and chat with the scientists and other novices. The UARC application can make any of its permissions available to these roles. However, for the application to execute safely and correctly, only the permissions actively in use should be made available. For example, scientists should only be able to access recordings and annotations made available by some scientist. Using the

access control model, the UARC application can define context-sensitive roles, so the rights of the application principals can be kept consistent with the state of the application. As the application state changes, transforms activate the permissions implied by that state (as described in Section 5.4).

DS defines the following API to retrieve transform limits for specific content.

```
ulong init(in ulong prinid);
ulong get_policy(in ulong derivation_id);
permission_array retrieve_transform_limits(in ulong derivation_id);
identity_ref_array retrieve_delegators(in ulong derivation_id);
transform_array int retrieve_transforms(in ulong derivation_id);
int retrieve_monitor_type(in ulong derivation_id);
```

The protocol for deriving transform limits is shown in Figure 9. First, the LI uses *init* to create a derivation session for the principal. Next, the LI uses the content principal's identity to retrieve the security policy for the content from the role selection hierarchy using the *get policy*. The policy specifies the identity's transform limits, initialization transforms (i.e., those from off-line principals such as the system administrator and user), and monitoring mechanism.[3] The four *retrieve* methods enable the downloading principal to view the derivation policy. For example, the potential delegators for which transform limits are defined may be retrieved using *retrieve delegators*.

We now examine how this mechanism works in the context of our UARC example. First, the UARC application's identity is used to retrieve its role specification using the *get policy*. The identity of the UARC application content consists of a downloading principal, content provider (UM developers), and its application identity (UARC). As described above, roles for application content are defined by system administrators. The UARC application role consists of two entries: (1) the transform limits for the downloading principal and the UARC application itself, defined by the system administrators and (2) initialization transforms specified by the downloading principal and system administrator. The transform limits, shown below, define the set of permissions within which system administrators, downloading principals, and the UARC application may delegate permissions to the UARC application.[4]

—Transform Limits

—Role: Downloading principal=(dp); provider=UM; application=UARC

—Definer: System administrators

---

[3]We only describe one authorization mechanism in Section 5, but the architecture can support an arbitrary number of authorization mechanisms.

[4]Permissions are defined using the following fields: positive or negative right, server, interface, operations, object, and optional operation limits. Also, parameters in the context-sensitive roles are compressed for readability from `downloading principal=dp`, `provider=UM`, and `application=UARC` to `dp`, `UM`, and `UARC`, respectively. Lowercase values for principal attributes (e.g., dp) are variables and uppercase values (e.g., UM) are literals.

—Delegator: System administrators
  —{+, *name_server*, *name_space*, [*open*], *system*}

—Delegator: Downloading principal=(dp)
  —{+, *file_server*, *file_server*, [*open*], *system_file_server*}
  —{+, *file_server*, *file*, [*read* | *write*], *writeable_files*(*dp*, *UM*, *UARC*)}
  —{+, *file_server*, *file*, [*read*], *readonly_files*(*dp*, *UM*, *UARC*)}
  —{+, *file_server*, *file*, [*exec*], *num_analysis*}

—Delegator: Downloading principal=(dp); provider=UM; application=UARC
  —{+, *net_server*, *net_server*, [*open*], *system_net_server*}
  —{+, *net_server*, *achannel*, [*connect*], *session_mgr*(*dp*, *UM*, *UARC*)}
  —{+, *net_server*, *achannel*, [*connect*], *data_server*(*dp*, *UM*, *UARC*), *max_bytes*}
  —{+, *http_serv*, *url*, [*get*], `http://www.uarc.org/notices.html`}
  —{+, *net_server*, *achannel*, [*connect*], *collaborators*(*dp*, *UM*, *UARC*)}

Transform limits associate a definer, delegator, and delegatee with the permissions that the delegator may delegate to the delegatee. The definer specifies object groups and the operations that delegators are allowed to grant on them. In the case of UARC, system administrators define the object groups in the transform limits. Only they may change the membership of these object groups. Although we do not show them here, the downloading principals and UARC application may also define transform limits for this role, as long as the permissions specified are a subset of their limits. For example, a specific downloading principal may want to define a new object group that further restricts that downloading principal's objects, which may be made accessible to the UARC application.

These transform limits are specified using context-sensitive permissions. The actual transform limits depend on the downloading principal and the application in the content's identity. For example, a variety of applications may have session managers and data servers, but only those associated with the UARC application are accessible. Also, the UARC collaborator principals (i.e., scientists and novices) are limited to those approved by the system administrator for collaboration with the downloading principal. The mapping of objects to object groups must , however, still be maintained by the system administrators, so the main advantage of using context-sensitive permissions is to keep the role graphs simpler. More advantages are seen for application roles described below.

Given these transform limits, the system administrators, the UARC application, and the downloading principal may delegate rights to the UARC application. Since these principals may not be active, they need another way to get their transforms executed. We enable such offline principals to specify initialization transforms which are to be executed by

the transform management service (TMS, see Section 5.4) when the content is started. In this example, we expect that both system administrators and downloading principals will delegate their full transform limits at initialization time.

The UARC application can delegate permissions to itself using its own transforms, as described in Section 5.4.

The UARC application content specifies the roles for its collaborator content and any outside content it controls. The UARC application is restricted to defining roles whose transform limits are a subset of its own permissions (i.e., union of the transform limits in its role). The UARC application defines transform limits for itself and the system administrators. System administrators always delegate access to the name server. Note that the object groups defined in these transform limits are managed by the UARC application. The UARC application content specifies the following transform limits for the *scientist* role:

—Transform limits

—Role: downloading principal=(dp); provider=(cp); application=UARC; role=scientist; appl id=(session)

—Definer: downloading principal=(dp); provider=UM; application=UARC

—Delegator: System administrators
  —{+, *name_server*, *name_space*, [*open*], *system*}

—Delegator: downloading principal=(dp); provider=UM; application=UARC
  —{+, *net_server*, *net_server*, [*open*], *system_net_server*}
  —{+, *file_server*, *file_server*, [*open*], *system_file_server*}
  —{+, *net_server*, *achannel*, [*connect*], *uarc_data_servers*(*dp*, *UM*, *UARC*)}
  —{+, *file_server*, *file*, [*read* | *write*], *uarc_annotations*(*dp*, *UM*, *UARC*)}
  —{+, *file_server*, *file*, [*read*], *uarc_replay*(*dp*, *UM*, *UARC*)}
  —{+, *file_server*, *file*, [*exec*], *num_analysis*(*dp*, *UM*, *UARC*)}
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *uarc*, [*open*], *sessions*}
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *replay*, [*start* | *annotate* | *stop*], *replays*}
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *data_display*, [*show_data*], *displays*}
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *chat*, [*read* | *write*], *chats*}

For the UARC role of *novice* the following transform limits are defined:

—Transform limits

—Role: downloading principal=(dp); provider=(cp); application=UARC; role=novice; appl id=(session)

—Definer: downloading principal=(dp); provider=UM; application=UARC

—Delegator: System administrators
  —{+, *name_server*, *name_space*, [*open*], *system*}

—Delegator: downloading principal=(dp); provider=UM; application=UARC
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *uarc*, [*open*], *sessions*}
  —{+, *uarc*(*dp*, *UM*, *UARC*, *session*), *chat*, [*read* | *write*], *chats*}

Recall that the membership of remote principals in the scientists or novices groups is determined by the system administrators in the construction of the role selection hierarchy. Given a combination of downloading principal, content provider, and application, the system administrators determine the legal application roles by specifying links to application role nodes in the role selection graph. Thus, the UARC application defines transform limits for its application roles, but the system administrators determine the precise assignment of principals to the role.

Scientists may be given the ability to start data analysis by loading an existing recording or using data from the UARC data server. Also, scientists may be able to perform numerical analyses and store recordings and annotations for later use. The parameterized object groups specified indicate that the objects to which access may be granted depends on the downloading principal, application, and content provider. The other permissions describe rights to objects in the particular UARC session, so further restriction based on context is not required. As specified, scientists can join a session, start, stop, and annotate a replay in that session, choose the data to display in the session, and engage in chat. UARC novices (e.g., high school students) may only join a session and engage in chat. They can, of course, see the session data being displayed, but may not specify the data to be displayed.

5.4 Transform Management Service

The transform management services (TMSs) manage the evolution of one or more principal's permissions. Since permissions are modified solely by transforms, TMSs need only to authorize transforms to completely manage the evolution of their principal's permissions. In addition, TMSs define delegation semantics for the architecture. Below, we list the design issues for TMSs:

—Should a transform be applied before or after the operation executes?

—What happens when a second delegator delegates a permission already possessed by the delegatee?

—What happens when a delegator revokes a permission delegated multiple times?

—Which principals are capable of revoking permissions they did not delegate?

—If a right is revoked from the delegator, how does this affect the delegatees of this right?

—If a transform delegates a set of rights, how does revocation of one of these rights affect the delegation of the entire set?

We first define transforms and then develop the answers to these questions.

We define transforms to be a tuple consisting of the following fields: (1) triggering operations and (2) transform operations. The triggering and transform operations define the association between a delegator and its delegatees. When a delegator executes a triggering operation, the associated transform operations delegate rights to or revoke rights from the delegatees. Because context-sensitive roles are used to define principals, delegations may be specified either by permissions or the addition of objects to object groups. As an example of the latter, when the user loads a UARC recording, that recording file can be added to the *replay files* of the current application instance. Management of permissions by managing object groups is useful for applications because as objects become accessible to applications, they can be added to the appropriate object group. Only those objects that are in use may then be accessible to application content (e.g., collaborator content).

A transform operation is defined as a tuple: (1) operation; (2) change; and (3) execution flag. The *operation* determines whether the change is an addition or a removal. The *change* specifies either a permission or an object and object group. This specifies the change in permissions performed by the transform. The *execution flag* determines whether the transform is applied before or after the operation itself is executed. The execution flag can also express that all copies of a permission are to be revoked (i.e., a server may revoke all permissions to an object it serves, as described below).

Given these requirements and the transform structure, each TMS works as follows (see Figure 10). Upon a transform, the TMS authorizes the transform against the transform limits for the delegator and delegatee. If authorized, the TMS adds the permissions to the delegatee with an association to the delegator (either before or after the operation, depending on the execution flag). Thus, by default, a delegatee may obtain multiple copies of a permission from multiple delegators; a delegator may only revoke the permissions that it delegated. The UARC application is simple in that it is the delegator of almost all the permissions in this example, but it seems clear that multiple principals may delegate and revoke rights independently. TMS supports separate delegations and revocations.

In some cases, however, a more privileged principal may be able to revoke permissions delegated by others. We define two cases: (1) object server revocation and (2) administrator revocation. First, it makes sense that the UARC application is able to revoke permissions to its own objects, regardless of the previous delegators. Therefore, we permit object servers to revoke all permissions to their own objects regardless of the identity of the original delegators. Second, the definers of transform limits may revoke

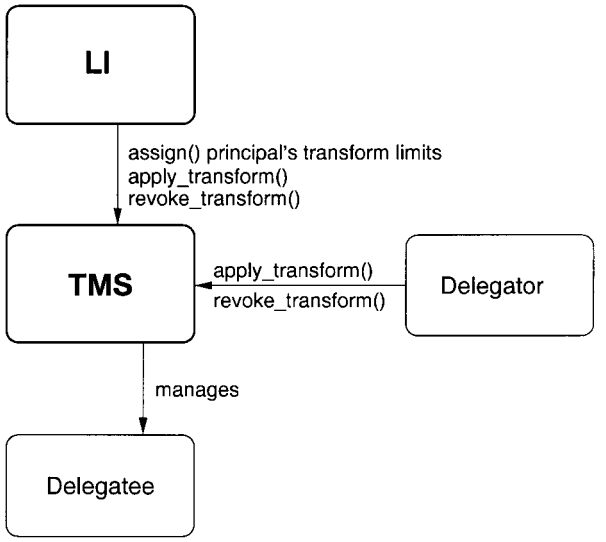

Fig. 10. Management service protocol: When a delegator executes an operation, the transforms associated with that operation are triggered and *apply transforms* on the delegatee's transform management service (TMS) is invoked. If a transform is authorized, its operations are applied such that the delegator either adds or removes permissions for the delegatee.

permissions on demand, regardless of the actual delegator, by revising the transform limits.

Since a principal may not delegate a right that it does not possess, a delegation must be revoked when the right is revoked from the delegator. In order to enforce this requirement, the TMS must be able find the permissions that it delegated and invalidate them. TMS maintains the permissions that its principals delegated in a *delegated permission set*. An entry in a delegated permission set contains a permission and the principals to whom it was delegated. When a permission is revoked from a principal, the TMS checks its delegated permission set to see if it has further delegated this permission. If so, all delegated permissions are invalidated by calling the delegatees' TMSs. We assume that such revocation is infrequent, so a highly optimized implementation is not necessary.

Since the revocation of rights need not mirror the delegation of rights in all circumstances in UARC, we do not require that the revocation of one right in a transform result in the revocation of all rights in that transform. We leave such management to the discretion of the delegators and servers. For example, the UARC application itself may delegate itself all the rights in its transform limits in one transform. If the system administrator removes a permission, the remaining rights should still be available.

The TMS API:

```
ulong assign(in ulong prinid, in ulong procid);
ulong apply_transform(in ulong prinid, in string transform);
ulong revoke_transform(in ulong prinid, in ulong transformid);
```

```
Op: UARC.start_scientists()

    {add, Perm(+,uarc(dp,cp,appl,session),recording,replays,[read|write]), after}
    {add, Perm(+,file_server,file,uarc_replays(dp,cp,appl,session),[read]), after}
    {add, Perm(+,file_server,file,uarc_annotations(dp,cp,appl,session),[read]), after}
    {add, Perm(+,uarc(dp,cp,appl,session),data_display,displays,[show_data]), after}
    {add, Perm(+,uarc(dp,cp,appl,session),chat,chats,[read|write]), after}
```

Fig. 11. Transform for object groups: When scientist content is started, it is granted access to specified object groups. Membership in these groups may be null initially, and the groups must be within the content principal's transform limits for the transform to be authorized.

Assign permits LI to assign a TMS to a specific principal and process. The principal identifier *prinid* determines the principal's permissions and transform limits.

Apply transform authorizes and executes transforms on behalf of the delegator. The delegator must be authorized to perform the specified transform. A transform to grant or revoke a permission is authorized if the permission is (1) possessed by the delegator/revoker and (2) within the transform limits of the delegator/revoker for the delegatee. If a permission is delegated, the TMS adds an active permission to the delegatee's permission set. TMSs and RMSs use the same authorization mechanism, shown in Figure 15 and detailed in Section 5.5

The *apply transform* operation returns an identifier for the applied transform. This enables explicit revocation of transforms, using *revoke transform*. This operation enables a specific transform to be revoked. In addition, a transform may specify revocations independent of transforms.

We now examine how TMSs are used in the UARC example. At initialization time, the UARC application is delegated its full set of rights by the initialization transforms retrieved in the previous section. These grant the UARC application full access to the permissions in its transform limits.

When collaborator content is loaded, the UARC application defines session-specific permissions for this content using a transform. An example of such a transform is shown in Figure 11. In this case, the *start scientist* operation provides scientist content with access to a set of object groups for the session. Initially, each object group contains no members, so this transform will be authorized.

Subsequent operations by the UARC application will update the membership of these object groups. The transform in Figure 12 states that when the UARC application executes the operation *user start display*, the specified recording and annotation files, as well as the replay data object, will be added to their respective object groups. If authorized, this transform enables principals to operate on these objects as specified by their rights to them. In this operation, a user selects a recording file and an annotation file for replay (called files r_file and a_file, respectively). As specified by

```
Op: recording x = Recordings.user_start_replay(r_file, a_file)

   {add, Object(r_file,uarc_replays(dp,cp,appl,session), before}
   {add, Object(a_file,uarc_annotations(dp,cp,appl,session), before}
   {add, Object(x,replays, after}
```

Fig. 12. Transform delegation: Upon the `user start replay` operation, delegate access to a recording being replayed (x), its file (r_file), and its associated annotation file (a_file).

```
Op: Recordings.user_stop_replay(x, r_file, a_file)

   {remove, Object(r_file,uarc_replays(dp,appl,session), after}
   {remove, Object(a_file,uarc_annotations(dp,appl,session), after}
   {remove, Object(x,replays, after}
```

Fig. 13. Transform revocation: Upon `user stop replay` operation, remove access from a recording being replayed (x), its file (r_file), and its associated annotation file (a_file).

the transform, these files are to be added to the object groups $uarc\_replays(dp, cp, appl, session)$ and $uarc\_annotations(dp, cp, appl, session)$ prior to the operation's execution. These transform operations must be authorized prior to the operation's execution. *user start replay* returns a reference to a replay object, and the transform specifies that the replay object is added to the *replays* object group for the session ($uarc(dp, cp, appl, session)$) after the operation returns.

Closure of a recording by the user should result in the removal of the rights to the recording. For example, the transform shown in Figure 13 associates the *user stop replay* with a removal of the replay, recording file, and annotation file permissions. The UARC application has access to these permissions and is permitted to delegate them, so it can execute this transform.

5.5 Reference Monitor Service

Reference monitor services (RMS) authorize a principal's operations using their permissions. When a request to perform an operation on an object is made by a principal, its RMS compares that request to its permissions to determine if a permission grants the request. For subsequent invocations of the same operation, the system servers may provide their clients with capabilities [Dennis and Van Horn 1966]. A capability associates a specific object with the holder's rights for performing operations on that object. RMSs also support conversion of permissions into legal capabilities, ensuring that these capabilities are unforgeable, unmodifiable, and revocable.

The UARC application performs operations on behalf of the downloading principal that the RMS must authorize, such as the *user start replay* command of the previous section. In this command, the user specifies the name of two files to be opened, and it is the job of the RMS to determine

—whether the file server is accessible to the UARC application;

—whether the specified files may be opened; and

—whether subsequent file operations on the file data are permitted.

First, some security policies may restrict communication between processes, so it is possible that a process may not even be permitted to send a request to a file server. Second, the RMS must authorize opening the specified files. This task is nontrivial because files are specified by name and permissions are specified by object identifier to prevent time-of-check-to-time-of-use (TOCTTOU) attacks [Bishop and Dilger 1996]. So the RMSs and server must engage in a protocol that enables each RMS to properly enforce the UARC application's permissions. Lastly, the file server returns a capability to the client which its RMS must intercept and authorize. The RMS must authorize the creation of a capability for the UARC application using its permissions, since to enforce system security policy the RMS must be able to prevent a server from granting a right to a client. Also, the RMSs maintain the UARC application's capability set, so that it is not possible for capabilities to be forged or modified, and capabilities may be immediately revoked when the associated permissions are invalidated.

A key issue in the design of the RMSs is the definition of the trust model between the RMSs and servers. The RMSs can control communication between processes (e.g., block IPCs to unauthorized servers), but since the servers determine the semantics of any operation, the RMSs trust that the semantics of the operations defined by the servers are implemented by the server. Servers define the mapping between data and object identifiers, so the RMSs also trust the servers to make this association correctly. Otherwise, the RMSs cannot effectively implement access control on server operations unless no communication is permitted.

An effective authorization mechanism must (1) mediate all operations; (2) protect itself from tampering; and (3) be simple enough to enable validation [Anderson 1972]. The RMS is designed to satisfy these three criteria. The RMS protocol is shown in Figure 14. First, the LI assigns an RMS to a principal. An RMS stores both the capabilities and permissions of the principals it manages. An RMS intercepts all of its principals' interdomain operation requests, and it authorizes whether the principals can perform the operations specified in the requests.

Authorization requirements for an operation are determined by the operation's *signature*. Each interface defines the signatures (i.e., their arguments, return values, and types) of its operations using a component interface definition language (IDL). There are three potential authorization requirements: (1) the authorization to perform the operation; (2) the authorization to transfer the capabilities in the operation; (3) the authori-

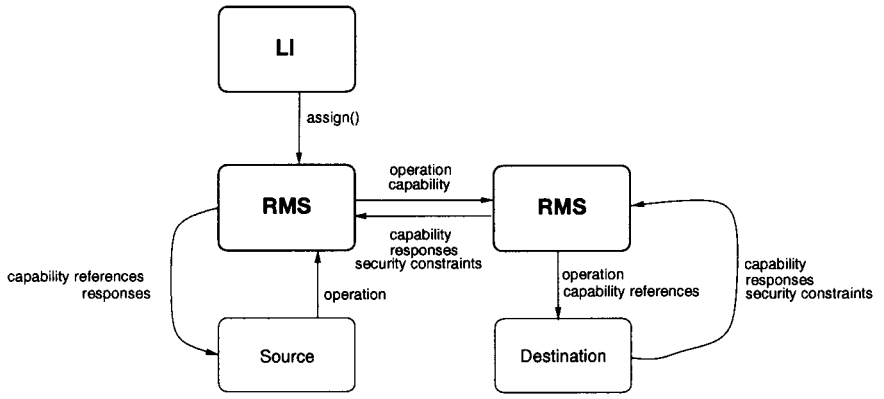

Fig. 14. Reference monitor service protocol: When a source performs an operation, its assigned reference monitor service (RMS) authorizes the operation using the source principal's capabilities. If authorized, the operation's destination RMS authorizes the delegation of any capabilities in the operation. The destination performs the operation and returns the result to the source. This result may include capabilities whose transfer is authorized by the source's RMS. References to the capabilities are returned to the source, rather than the capabilities themselves, to prevent forgery and modification.

zation of the response; and (4) the authorization to transfer the capabilities in the operation's response. First, an operation is permitted by an RMS if the client process has a valid capability to perform the requested operation on the server and object (first argument in the operation). Next, the signature may indicate that an operation passes capabilities to the server. These must be authorized by the server's RMS. The server's RMS permits responses to operations to the client. Finally, the operation's signature may indicate that the return value is a capability. These are authorized by the client's RMS. The RMS caches the capabilities delegated to its principals and provides capability references for them. Since processes only hold capability references, the capability cannot be forged or modified, and immediate revocation is possible.

Servers may also specify further security constraints to be enforced by the RMS. For example, a server can update the quantity of a resource used by the process, and the RMS can authorize whether this exceeds usage requirements. If the usage limit is exceeded, the result is not returned and the capability is invalidated.

A capability consists of the following fields: (1) server; (2) interface; (3) object identifier; and (4) rights. The server field identifies a unique server process identifier. Therefore, the capabilities are bound to a specific process. A process may include a set of components, each of which may define multiple interfaces. The interfaces determine the type of the object upon which the operation is invoked. The object identifier and rights are the traditional fields in a capability. The rights may be extended to enforce limited use operations, as described below.

```
authorize(ulong:client, ulong:server, ulong:type, ulong:object, ulong:ops):
   Set found_ops to NULL ;
   do for each p in positive permissions
      if p.type ≠ type then continue fi ;
      if p.comp_inst ≠ comp_inst then continue fi ;
      if p.server ≠ server then continue fi ;
      if p.object = object
         then Add p.ops to found_ops
      fi ;
   od ;

   if ops ⊈ found_ops then return FALSE fi ;

   do for each p in negative permissions
      if p.type ≠ type then continue fi ;
      if p.comp_inst ≠ comp_inst then continue fi ;
      if p.server ≠ server then continue fi ;
      if p.object = object and p.ops ∩ found_ops then return FALSE fi ;
   od ;
   return TRUE ;
```

Fig. 15. Authorization mechanism for authorizing transforms and capability delegations.

Our architecture permits creation of multiple RMSs that may control the operations of one or more content processes (i.e., domains). The use of multiple RMSs enables a different security policy to be enforced on different protection domains. Therefore, each of the policies described above can be assigned and enforced. Each RMS provides the following API:

```
int assign(in ulong prinid, in ulong procid);
ulong self_identity();
ulong auth_service(in ulong prinid);
int authorize(in ulong client, in ulong server, in ulong type,
in ulong object, in ulong ops);
```

The *assign* command assigns an RMS to a content process and principal. Only the LI may use this command to transfer control of the content to an RMS. The RMS uses the principal identifier *prinid* object to find the principal and its permissions for the content process. The RMS both reads and updates the principal's capabilities.

A process may obtain its own identity using the *self identity* command. Knowing one's identity is useful for determining which names need to be resolved into capabilities. For example, the UARC collaborator content uses its identity to determine the UARC session to which it belongs.

The RMS authorization mechanism is shown in Figure 15. To authorize the delegation of a capability, the RMS must find (1) permission in the principal's permission set that grants use of the operation on the object and type in the server and (2) no negative permission in the principal's permission set precluding the operation on the object and type in the server.

Typically, operations are authorized transparently to the client and server. However, a principal may manually verify whether a particular

operation is permitted. Principals may retrieve the RMS for a specified principal using the *auth service* function. Principals may then request that the content's RMS authorize its operation using the *authorize* command.

We now examine how the RMS supports the UARC application. When the UARC application is initiated, it has a set of transform limits within which permissions may be delegated, and an initial set of permissions delegated by the downloading principal to access, respectively, UARC files and remote principals. However, in order to use these permissions, the UARC application must be able to identify the file and network server processes. A system name server that converts logical names to capabilitiesis is defined. The UARC application, however, cannot even access the name server without a capability. System administrators not only grant the UARC application permission to access the name server, but the UARC application must be initiated with a capability to the name server.

The name server capability is provided by the LI using a content initialization operation, *init component*, which all content must define for this purpose. Since RMS knows the signature of this operation, it knows that a name server capability is passed to the content process. The RMS permits the content principal to obtain the capability if it is permitted to resolve names using the specified name server. Thus, RMS can be used to restrict the name servers to which a content process can communicate. The RMS stores the capability and provides the UARC application with a reference to the capability, which it uses in operation requests to the name server.

Using the name server capability reference, the UARC application requests capabilities for the file and network servers. First, it obtains capability references to perform *open* operations on these servers. Using these references, the UARC application can then open specific files and communication channels. For example, it can specify the identity of the recording and annotation files to be opened when the downloading principal executes the *user start replay* operation. When a file is opened, a capability for accessing the file is returned by the file server. RMS recognizes capability delegation using the operation's signature, and authorizes delegation using the content principal's permissions. RMS then returns a capability reference to the files to the UARC application, so the UARC application may only perform authorized operations on the files.

In addition, the server may use RMS to help it enforce its security requirements. In particular, a server may collaborate with the RMS to enforce limited use of its operations. In Section 5, the UARC application is restricted to download no more than max_bytes from the UARC data server. Capabilities may be extended to include operation limits. In this case, the rights field in the capability refers to a sequence of structures of the form: (1) operations bit map; (2) current value; and (3) limit. That is, each operation may have a limit associated with it. The current value indicates how much of the limit has been used. Since it is difficult for the RMS to determine how to increment the current value on an arbitrary

operation, it must depend on the server to provide such information. Therefore, limit restrictions cannot be enforced without the server's cooperation. Servers may provide a capability in addition to the return data, including an updated limits field. The RMS can then authorize the updated limit against its security requirements. If the limit is exceeded, the resultant value is not returned, but an error is returned instead.

The UARC application receives content it needs to execute from collaborators. Once the collaborator content is loaded, it is also granted a capability reference to the name server. It uses this reference to obtain capabilities to servers, including the UARC application instance in which the content is loaded. Multiple UARC application instances may be running in one system, so the collaborator content must determine its instance before it can ask for a capability. The application identifier is part of the collaborator content principal's identity, and the collaborator content can use the RMSs *self identity* operation to retrieve its identity. It can then construct a description of the UARC application instance sufficient to retrieve its capability from the name server. Note that it cannot obtain a capability to another UARC application instance because it only has permissions to access its own instance.

When a principal's permission is revoked, any capabilities associated with that permission must also be revoked. The simplest way to address a revocation is to invalidate all the principal's capabilities. When a reference to an invalid capability is used in an operation, the capability may be reauthorized by the RMS. One of the bits in the capability's rights field is reserved for signaling whether the capability is valid (i.e., the *valid bit*). Permission revocation ensures that any further delegations of this, now invalid, capability are revoked.

6. IMPLEMENTATION

We implemented the described security architecture in IBM's Lava operating system environment. Lava enables composition of component-based operating system on a small nucleus (about 12K of code) upon which system services and applications can be configured dynamically. Lava prototypes run on Intel Pentium, Pentium Pro, and Pentium II machines.

6.1 Implementation Model

A Lava architecture to support UARC clients is shown in Figure 16. Lava architecture consists of a nucleus, loader interface, reference monitors, name server, and components. The nucleus, loader interface, and reference monitors comprise the TCB of the system. In addition, the system's security depends on the integrity of the security policy. For such reasons, the system policy is stored on a secure server. Updates to the policy are limited by a role administration hierarchy. In general, system administrators manage all roles except those internal to an application. Only the application principals can manage those roles.

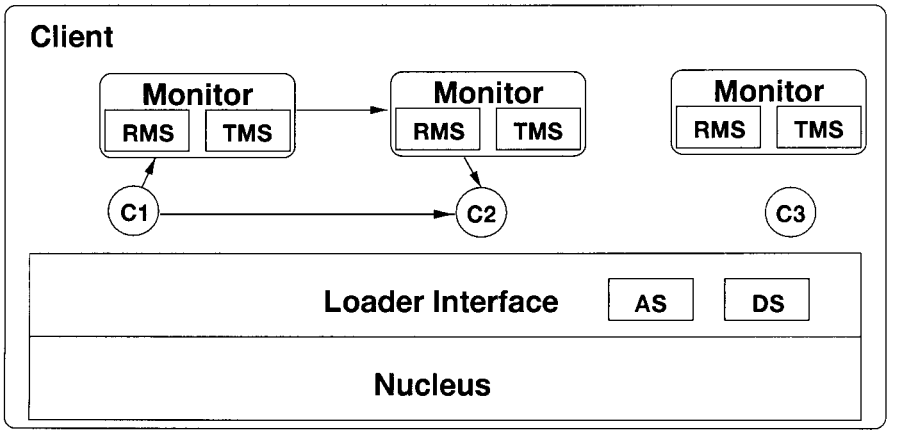

Fig. 16. Lava system architecture: Loader interface includes authentication and derivation services of the architecture. It creates tasks and assigns each task to its own monitor. A monitor implements both the transform management and reference monitor services.

The nucleus provides fundamental operating system functionality: tasks (e.g., processes), threads, address spaces, interprocess communication (IPC), flexible memory management, and interrupts. The nucleus also provides an IPC redirection model (called *Clans & Chiefs* [Liedtke 1992]) in which a monitor can be assigned to a task, and the nucleus automatically redirects all intertask IPCs to or from the controlled task to the monitor. This mechanism's semantics determine that all the tasks belonging to a monitor can send IPCs freely; however, in order to control communication between each pair of tasks, each individual task must be assigned a monitor. The nucleus identifies the sender of any IPC.

The loader interface (LI) implements the services described in Section 5.1. It also includes the authentication service (AS) and derivation service (DS) components. The loader interface handles requests from the downloading principal to load content into new or existing tasks. As described in Section 5, this composite LI authenticates the content principal (according to the AS interface), derives its transform limits (according to the DS interface), and loads the content such that a reference monitor can effectively enforce those permissions.

A reference monitor includes the transform management service (TMS) and reference monitor service (RMS) components (see Sections 5.4 and 5.5, respectively). The loader interface loads reference monitors and tasks in such a way that the nucleus automatically redirects any intertask IPC to the monitors. For example, when C1 invokes an operation on C2, it is implemented as an IPC that is automatically redirected to C1's and C2's monitors prior to C2 receiving the operation (shown in Figure 16). A monitor has access to capabilities, permissions, and transform limits, and intercepts its task's IPCs so it can authorize operations on strongly-typed objects, prevent capability forgery, control permission and capability delegation, and provide immediate revocation.

A component is a set of interfaces and implementations. The set of interfaces defines the set of types of objects that a component may serve. Note that a component may be either a client or server, depending on the situation. All components are trusted to behave according to the server trust model defined in Section 5.5. An interface defines a set of methods that may be invoked on objects with that interface. Components are loaded into tasks. Multiple components (even the same one multiple times) may be loaded into a single task.

In Lava, tasks are multithreaded, so IPCs are sent to threads not tasks. Thus, method invocation requires the following information: task, thread, component instance (since there may be multiple instances of the same component in the same task), interface, object, and method. Capabilities (see Section 5.5) determine which methods can be invoked by a principal. So Lava capabilities have the following fields: task and thread (server), component instance and interface (interface), object, and method (operation).

When the Lava client is booted (we assume a secure boot mechanism [Wobber et al. 1994]), the nucleus starts the loader interface which, in turn, starts the security services (e.g., AS and DS), basic system services (e.g., name server and device drivers), and an initial task for the potential downloading principals (e.g., login). The loader interface uses password authentication to verify a downloading principal. However, smart card authentication is preferred, so the downloading principal does not have to trust the client with its secrets.

A secure machine stores the system administrators' security policy for their user community. Security policy specifies authentication requirements, transform limits, and initialization transforms in role hierarchies. Access control lists are used to protect the policy from unauthorized modification. Any client system is trusted to retrieve entries from the database.

The downloading principal task can then request that other content be loaded, such as other systems support (e.g., device drivers) or applications, such as the UARC application, by using the loader interface. When a component is downloaded, its security policy is retrieved from the server for authentication and permission derivation.

Tasks like verifying signatures (including the management of public key certificates) and digests are performed using the cryptographic services of the IBM KeyWorks Toolkit. This toolkit implements the Open Group's Common Data Security Architecture (CDSA) [Open Group 1997] cryptographic API. It provides comprehensive services for protection of secrets, trust management, and a variety of cryptographic algorithms and protocols.

The derivation service is not performance-critical, so it is implemented much as described in Section 5.3. The derived transform limits are then uploaded to the appropriate monitor. Applications maintain there own transforms and request that monitors execute them as they desire. Of course, the monitors verify that all transforms are legal. The delegator's

monitor verifies that the delegator has the specified permissions; the delegatee's monitor verifies that the permissions being delegated are within the transform limit for the delegator-delegatee pair.

The key part of the implementation is how individual operations are handled by the reference monitors. Reference monitors must authorize operations and delegation of capabilities. We start with the specification of an operation. In Lava, tasks define protection domains, so monitors are designed to authorize intertask operations and responses. An intertask operation specifies (1) the destination task and thread; (2) the component instance; (3) the interface; (4) the operation identifier; and (5) the operation arguments. Because the signature of a response is different from the signature of a request, the monitor must be able to determine both the operation and the whether it is a request or response. The operation can be uniquely determined by the interface and operation identifier. Whether the operation is a request or response is determined by the value of the component instance. No component instance is required on a response, so the value is null.

In order to minimize authorization overhead, the monitors must have an efficient mechanism to retrieve permissions and capabilities. A monitor must perform the following operations:

(1) Given a capability reference, determine if the associated capability permits the specified operation;

(2) Given a capability delegation, determine if the principal already has the capability;

(3) Given a capability delegation, determine if the principal has a permission that permits it to use the capability;

(4) Given a reduction in permissions, invalidate the effected capabilities.

We expect case 1 and 2 to be the most frequent, so we optimize for them. First, the capability references created by monitors given to the client directly, identify the location of the capability in the monitor. Thus, monitors can retrieve capabilities from references directly. Second, monitors must determine whether they need to authorize the transfer of a capability. If the destination has the specified capability already, then authorization is not necessary. A problem is that if different processes have different references to the same capability, then a reference conversion step (reference for process A is converted to a reference to the same capability in process B) is required for the transfer. To eliminate this step, monitors use the same capability reference for capabilities to the same object. This reference is determined by the server's monitor to ensure that it is the same for all clients. The reference number is a combination of component instance and reference, so it is unique over all components. Therefore, a two-step process is needed to retrieve a capability: (1) find the component's capabilities; and (2) find the capability at the specific reference.

In the third case, authorization using permissions is not as performance-critical because it is only done to gain a capability to an object, and capabilities are used on all subsequent accesses. Permissions are simply hashed by the server, component instance, interface, and object identifier. They must be accessed to authorize the delegation of a capability or execution of a transform. Recall that the access control model supports hierarchical object identifier spaces, so multiple permissions may need to be examined to authorize an operation. The authorization mechanism first finds the permission for the object identifier specified, then for the parent object identifier, and so on. Of course, all ancestors of an object identifier must be checked for a negative permission because any negative permission precludes the operation.

The fourth case occurs when a transform removes a permission from a principal. The removal of a permission may or may not affect the principal's current capability set. For example, a capability may be backed by multiple permissions delegated from multiple sources, so it is not necessary to revoke the capability when one permission is removed. But the problem is that potentially many capabilities may be affected by a revocation (e.g., if it occurs at an nonleaf object identifier). In general, all capabilities whose object identifiers begin with the identifier specified in the removed permission must be revalidated. At present, we simply scan the set of capabilities for such entries and reset their valid bit. Exploring whether a more efficient mechanism is needed is future work.

An initial performance analysis of the system has been done previously [Jaeger et al. 1998]. We summarize the results briefly here. We examined the optimal performance of Lava's security architecture capability authorization. First, we estimated the optimal expected performance using microbenchmarks of the Lava nucleus IPC, operation analysis, and capability authorization. We then measured the actual performance of an implementation of these mechanisms. The estimated optimal performance is about 4 μs, but the actual measured performance is about 9.5 μs. The difference is largely attributable to some cache and TLB misses. With further analysis, these may be reduced or eliminated, but a macroanalysis is also necessary. Note that in this performance analysis the principals' capabilities are stored in a simple array, and in the protocol described here a second array access is necessary (for the component). We are developing a more flexible redirection mechanism than Clans & Chiefs, which, in many cases, will enable a single monitor to redirect a destination for multiple processes [Jaeger et al. 1999].

7. CONCLUSIONS AND FUTURE WORK

We presented a system architecture that enables flexible access control of execution of downloaded content, using both system and application security requirements. Design and implementation of this system shows that it is feasible to construct an architecture that can support both system and application-specific access control policies. We show that a variety of

application access control requirements can be enforced, so that little, if any, ad hoc application security infrastructure needs to be built. The access control model enables privileged principals to specify mandatory access control policies. They may also specify access control domains, within which less privileged principals may perform discretionary access control. For example, system administrators can define what rights users and application developers can delegate to content. If they desire, users and application developers can further restrict these limits, even maintaining consistency between application state and the commensurate security requirements. We demonstrate the architecture by defining access control policies for a collaborative application, and believe that many distributed applications that use content will benefit from such flexibility.

We describe an implementation of our architecture in the Lava operating system environment. The Lava-based system consists of a nucleus, loader interface, reference monitors, and components. The nucleus provides basic operating system primitives (tasks, threads, IPC, etc.) and automatic IPC redirection. The loader interface derives principals (permissions, transforms, and transform limits) and loads content so that a system reference monitor can enforce the principal's access control policy. Since the nucleus automatically redirects IPCs to the reference monitor, it is able to authorize all intertask operations, authorize delegations, and revoke delegations.

Using the Kain and Landwehr [1986] capability taxonomy, we rate the system as $abdab$. The rights associated with a new capability depend on the security policy (even for newly created objects), so restricted access may be inserted in a new capability ($1 = a$). Subsequent policy changes lead to marking capabilities invalid (e.g., if the transform limits are changed, $2 = b$). Capability copying (i.e., delegation) is controlled by monitors, so rights are determined by a trusted process ($3 = d$). When a capability is provided for access, it is not changed unless it has been marked for reverification due to a transform limit change ($4 = a$). Lastly, access checking is performed using the available access rights ($5 = b$).

We are interested in examining the trade-offs between performance and policy complexity. We have some initial performance results that an authorized IPC (i.e., capability validation) takes 9.5 μs [Jaeger 1998], which is many times faster than IPC in other systems, and we believe that further optimizations are possible (the current ideal time is 4 μs). While these performance results are promising, further optimization is possible by removing one of the reference monitors from the IPC path. We have designed an IPC redirection protocol that enables flexible assignment of reference monitors to IPC paths [Jaeger et al. 1999]. Protocols for the management and enforcement of security policies need to be designed. We are also examining other domains, particularly the composition of operating systems from components, so we expect that new policy requirements will arise. The effect of the enforcement of these requirements on performance will be examined.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Ravi Sandhu for their helpful guidance in the preparation of this paper. We also thank our colleagues at the IBM Thomas J. Watson Research Center: Paul Karger, Larry Koved, Yoonho Park, Seva Panteleenko, J. R. Rao, David Safford, Jonathon Shapiro, John Tracey, Volkmar Uhlig, and Leendert van Doorn. We also thank colleagues with whom we had the pleasure of several discussions and, with some, of collaboration, including Ed Felten, Li Gong, Peter Honeyman, Avi Rubin, and Dan Wallach.

REFERENCES

ANDERSON, J. P. 1972. Computer security technology planning study. Tech. Rep. ESD-TR-73-51. James P. Anderson and Co., Fort Washington, PA.

BELANI, E., VAHDAT, A., ANDERSON, T., AND DAHLIN, M. 1998. The CRISIS wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium* (Jan.). USENIX Assoc., Berkeley, CA, 15–29.

BERTINO, E., FERRARI, E., AND ATLURI, V. 1999. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur. 1*, 2 (Feb.), 65–104.

BISHOP, M. AND DILGER, M. 1996. Checking for race conditions in file accesses. *Comput. Syst. 9*, 2, 131–152.

BOEBERT, W. E. AND KAIN, R. Y. 1985. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Conference on Computer Security*. 18–27.

BORENSTEIN, N. S. 1992. Computational mail as network infrastructure for computer-supported cooperative work. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), M. Mantel and R. Baecker, Eds. ACM Press, New York, NY, 67–74.

BORENSTEIN, N. S. 1994. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*. 389–402.

BREWER, D. F. C. AND NASH, M. J. 1989. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA). IEEE Computer Society Press, Los Alamitos, CA, 206–214.

CLAUER, R. C. E. AL. 1995. A prototype upper atmospheric collaboratory (UARC). In *Applications of Data Handling and Visualization Technique in Atmospheric Space Sciences*. 105–112.

CORP. FOR NATIONAL RESEARCH INITIATIVES, 1998. Grail home page. grail.cnri.reston.va.us/grail/

DEAN, D., FELTEN, E., AND WALLACH, D. 1996. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, May). IEEE Press, Piscataway, NJ.

DENNIS, J. B. AND VAN HORN, E. C. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (Mar.), 143–155.

DORWARD, S., PIKE, R., AND WINTERBOTTOM, P. 1996. Inferno: la commedia interattiva. inferno.bell-labs.com

ELECTRIC COMMUNITIES, 1999. Using the EC Ttrust manager to secure Java. www.communities.com/company/papers/trust/index.html

FOLEY, S. AND JACOB, J. 1991. Specifying security for CSCW systems. In *Proceedings of the Fourth IEEE Workshop on Computer Security Foundations*. IEEE Computer Society Press, Los Alamitos, CA, 136–145.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL Protocol Version 3.0. Internet Draft.

GALLO, F. S. 1996. Penguin: Java done right. *Perl J. 1*, 2, 10–12.

GASSER, M. AND MCDERMOTT, E. 1990. An architecture for practical delegation in a distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA). IEEE Computer Society Press, Los Alamitos, CA, 20–30.

GIURI, L. AND IGLIO, P. 1997. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (RBAC '97, Fairfax, VA, Nov. 6–7, 1997), C. Youman, E. Coyne, and T. Jaeger, Eds. ACM Press, New York, NY, 153–159.

GOLDBERG, Y., SAFRAN, M., AND SHAPIRO, E. 1992. Active mail—a framework for implementing groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), M. Mantel and R. Baecker, Eds. ACM Press, New York, NY, 75–83.

GONG, L. 1997. Enclaves: Enabling secure communication over the internet. *IEEE J. Sel. Areas Commun. 15*, 3 (Apr.).

GONG, L. 1997. Java security: Present and near future. *IEEE Micro 17*, 3, 14–19.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

GRIMM, R. AND BERSHAD, B. N. 1998. Providing policy-neutral and transparent access control in extensible systems. Technical Report Number UW-CSE-98-02-02. University of Washington, Seattle, WA.

HAGIMONT, D. AND ISMAIL, L. 1997. A protection scheme for mobile agents on Java. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking* (MOBICOM '97, Budapest, Hungary, Sept. 26–30, 1997), L. Pap, K. Sohraby, D. B. Johnson, and C. Rose, Eds. ACM Press, New York, NY, 215–222.

HALEVI, S. AND KRAWCZYK, H. 1997. MMH: Software message authentication in the Gbit/s rates. In *Proceedings of the Fourth Workshop on Fast Encryption*.

HAWBLITZEL, C., CHANG, C.-C., CZAJKOWSKI, G., HU, D., AND VON EICKEN, T. 1998. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Conference*. USENIX Assoc., Berkeley, CA.

ISLAM, N., ANAND, R., JAEGER, T., AND RAO, J. R. 1997. A flexible security model for using Internet content. *IEEE Softw. 14*, 5 (Sept.).

JAEGER, T., ELPHINSTONE, K., LIEDTKE, J., PANTELEENKO, V., AND PARK, Y. 1999. Flexible access control using IPC redirection. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*.

JAEGER, T., GIRAUD, F., ISLAM, N., AND LIEDTKE, J. 1997. A role-based access control model for protection domain derivation and management. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (RBAC '97, Fairfax, VA, Nov. 6–7, 1997), C. Youman, E. Coyne, and T. Jaeger, Eds. ACM Press, New York, NY, 95–106.

JAEGER, T., LIEDTKE, J., AND ISLAM, N. 1998. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium* (Jan.). USENIX Assoc., Berkeley, CA, 143–156.

JAEGER, T. AND PRAKASH, A. 1994. Support for the file system security requirements of computational E-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (Fairfax, VA, Nov. 2–4), D. Denning, R. Pyle, R. Ganesan, and R. Sandhu, Eds. ACM Press, New York, NY, 1–9.

JAEGER, T. AND PRAKASH, A. 1995. Implementation of a discretionary access control model for script-based systems. In *Proceedings of the 8th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society Press, Los Alamitos, CA, 70–84.

JAEGER, T., RUBIN, A., AND PRAKASH, A. 1996. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*. USENIX Assoc., Berkeley, CA, 131–148.

KAIN, R. Y. AND LANDWEHR, C. E. 1986. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy* (Oakland, CA, Apr. 7-9, 1986). IEEE Computer Society Press, Los Alamitos, CA, 95–100.

KARJOTH, G. 1998. Authorization in CORBA security. In *Proceedings of the Conference on ESORICS*.

KNISTER, M. AND PRAKASH, A. 1993. Issues in the design of a toolkit for supporting multiple group editors. *Comput. Syst. 6*, 2, 135–166.

LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. 1992. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 265–310.

LEE, J., PRAKASH, A., JAEGER, T., AND WU, G. 1996. Supporting multi-user multi-applet workspaces in CBE. In *Proceedings of the 6th ACM Conference on Computer-Supported Coorperative Work* (CSCW '96, Boston MA, Nov.). ACM Press, New York, NY, 344–353.

LEVY, J. Y. AND OUSTERHOUT, J. K. 1995. Safe Tcl: A toolbox for constructing electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*. USENIX Assoc., Berkeley, CA, 133–135.

LIEDTKE, J. 1992. Clans & chiefs. In *Architektur von Rechensystemen*. Springer-Verlag, Vienna, Austria.

LIEDTKE, J. 1995. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (SIGOPS '95, Copper Mountain Resort, CO, Dec. 3–6), M. B. Jones, Ed. ACM Press, New York, NY.

LUPU, E. AND SLOMAN, M. 1997. Reconciling role based management and role based access control. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (RBAC '97, Fairfax, VA, Nov. 6–7, 1997), C. Youman, E. Coyne, and T. Jaeger, Eds. ACM Press, New York, NY, 135–141.

MINEAR, S. E. 1995. Providing policy control over object operations in a Mach-based system. In *Proceedings of the 5th USENIX Security Symposium*. USENIX Assoc., Berkeley, CA.

MINSKY, N. H. AND UNGUREANU, V. 1998. Unified support for heterogenous security policies in distributed systems. In *Proceedings of the 7th USENIX Security Symposium* (Jan.). USENIX Assoc., Berkeley, CA, 131–142.

NIST, 1994. *NIST FIPS PUB 186, Digital Signature Standard*. U.S. Department of Commerce.

NIST, 1995. NIST FIPS PUB 180-1, Secure Hash Standard. National Institute of Standards and Technology, Gaithersburg, MD.

NETSCAPE CORP., 1997. Introduction to the capabilities classes. Netscape Corp.. Available from developer.netscape.com/library/

NETSCAPE CORP., 1999. The Navigator Java environment: current security issues. Netscape Corp.. Available at developer.netscape.com/docs/manuals/javasecurity.html.

OBJECT MANAGEMENT GROUP, 1997. Security service specification. In *CORBAservices: Common Object Services Specification*, Object Management Group. Available from http://www.omg.org

THE OPEN GROUP, 1997. Common security: CDSA and CSSM. Available from http://www.opengroup.org

OUSTERHOUT, J. K. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

OUSTERHOUT, J. K., LEVY, J. Y., AND WELCH, B. B. 1998. The Safe-Tcl security model. In *Proceedings of the 23rd USENIX Annual Conference*. USENIX Assoc., Berkeley, CA.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proc. IEEE 63*, 9 (Sept.), 1278–1308.

SANDHU, R. 1998. Role activation hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control* (RBAC '98, Fairfax, VA, Oct. 22–23, 1998), C. Youman and T. Jaeger, Eds. ACM Press, New York, NY, 33–40.

SANDHU, R. S., BHAMIDIPATI, V., AND MUNAWER, Q. 1999. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur. 1*, 2 (Feb.).

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. 1996. Role-based access control models. *IEEE Comput. 29*, 2, 38–47.

SUN MICROSYSTEMS, 1999. Frequently asked questions: Java security. Sun Microsystems, Inc., Mountain View, CA.

THOMSEN, D., O'BRIEN, D., AND BOGLE, J. 1998. Role based access control framework for network enterprises. In *Proceedings of the 14th Conference on Computer Security Applications*. IEEE Computer Society Press, Los Alamitos, CA.

TRUSTED INFORMATION SYSTEMS, INC., 1994. Trusted Mach System Architecture (TIS TMACH Edoc-0001-94A ed.). Trusted Information Systems, Inc..

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack introspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA.

WHITE, J. E. 1995. Telescript Language Reference Manual. Available from www.genmagic-.com.

WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. 1994. Authentication in the Taos operating system. *ACM Trans. Comput. Syst. 12*, 1 (Feb. 1994), 3–32.

Received: May 1997; revised: April 1998; accepted: October 1998