Building Systems That Flexibly Control Downloaded Executable Content

Trent Jaeger† Aviel D. Rubin‡ Atul Prakash†

Software Systems Research Lab†
EECS Department
University of Michigan
Ann Arbor, MI 48105
Emails: {jaegert|aprakash}@eecs.umich.edu

Security Research Group‡
Bellcore
445 South Street
Morristown, NJ 07960
rubin@bellcore.com

Abstract

Downloading executable content, which enables principals to run programs from remote sites, is a key technology in a number of emerging applications, including collaborative systems, electronic commerce, and web information services. However, the use of downloaded executable content also presents serious security problems because it enables remote principals to execute programs on behalf of the downloading principal. Unless downloaded executable content is properly controlled, a malicious remote principal may obtain unauthorized access to the downloading principal's resources. Current solutions either attempt to strictly limit the capabilities of downloaded content or require complete trust in the remote principal, so applications which require intermediate amounts of sharing, such as collaborative applications, cannot be constructed over insecure networks. In this paper, we describe an architecture that flexibly controls the access rights of downloaded content by: (1) authenticating content sources; (2) determining content access rights based on its source and the application that it is implementing; and (3) enforcing these access rights over a wide variety of objects and for the entire computation, even if external software is used. We describe the architecture in the context of an infrastructure for supporting collaborative applications.

1 Introduction

The ability to download executable content is emerging as a key technology in a number of applications, including collaborative systems, electronic commerce, and web information services. By downloading executable content on demand, systems can be built that provide better performance and fault tolerance than existing systems. Application performance can be improved because the program (i.e., executable content) can be downloaded to the location of the data (e.g., for a query) or the location of the user (e.g., for an interface-driven task). In addition, the fault tolerance of an application can be improved by reducing the client's dependency on the liveness of a specific server.

Recent distributed systems architectures, such as mobile agent and replicated process architectures, enable processes to download and run executable content. For example, consider the mobile agent architecture (also known as computational e-mail, command script architecture, and enabled mail) in Figure 1. First, a remote principal composes an agent by specifying a program. Through some mechanism (e.g., `http` or e-mail) the agent is downloaded to another principal. The downloading principal uses an agent interpreter process running on his machine to execute the mobile agent (number 2 in the figure). This process is owned by the downloading principal, so the agent is executed with his access rights. A malicious remote principal can use these access rights to: (1) read and write the downloading principal's private objects; (2) execute applications, such as `mail` or cryptographic software, to masquerade as the downloading principal to other users; and (3) read the password file on the downloading principal's machine. In addition, a remote principal may be spoofed into integrating malicious content from an attacker to mobile agent which would then give a third party access to the information provided above [5].

Current interpreters for executing downloaded content, such as Java-enabled Netscape, Java's ap-

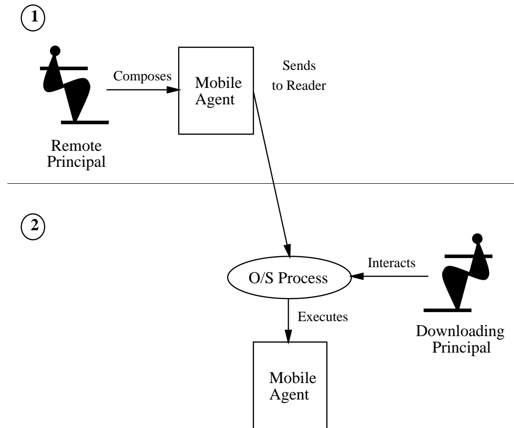

Figure 1: Mobile agent architecture

pletviewer [10], and Tcl's safe interpreter [3, 16], strictly limit the access rights of content to prevent these attacks. For example, content run using Java-enabled Netscape is prevented from performing any file system I/O and communicating with third parties (in theory anyway, see [5]). The Java appletviewer permits some access rights to be granted to content (as specified in the `~/.hotjava/properties` file). However, these rights must be shared with any remote principal whose content is downloaded. Therefore, these interpreters are not suitable for building an application where a downloading principal must grant rights to a specific remote principal.

Our goal is to flexibly control downloaded content, so applications which need to share resources with specific remote principals can be built. Flexible control of downloaded content means that the content's access rights can be set to any subset of the downloading principal's rights. Flexible control of downloaded content is difficult because: (1) to determine the access rights of downloaded content requires decisions at runtime and (2) to enforce these rights requires control of a wide variety of system objects. Content access rights depend on: (1) the trust in the remote principal who provided the downloaded content; (2) the access requirements of the application that the content is implementing; and (3) the state of that application. The state of the application and the access control requirements of the content are not known until runtime time, so a principal trusted to make access control decisions at runtime is necessary. However, users are typically not trusted to make such judgements. Also, executing content requires that access to a wide variety of system objects, such as files, sockets, and existing software, be controlled. In particular, control of existing software is not supported by interpreters because they are simply user processes. However, current operating systems also lack the necessary tools to effectively control external software executed by downloaded content.

In this paper, we describe an architecture for flexible control of downloaded content. This architecture enables downloading principals to execute applications that use content to share resources in a controlled manner to complete the application's goals. The architecture enables the access rights of downloaded content to be determined with little need for runtime access control specification by users. The architecture: (1) authenticates content sources; (2) determines content access rights based on its source and the application that it is implementing; and (3) enforces these access rights over a wide variety of objects and for the entire computation, even if external software is used. Cryptographic authentication identifies the source of content and also prevents some of the attacks that have plagued current interpreters, such as the DNS attack described in [5]. Also, in addition to the source, the application of the content is also used to determine the access rights of content. For example, we can remove the limitation that content can only communicate with ports at the server, by knowing the authorized remote principals in the application. We define an access control model for expressing the access rights of system objects and services for enforcing those rights. For example, the execution of external software is administered by a trusted interpreter, so we can control what software is being executed and limit the rights available to it.

Throughout the paper, we will assume a conventional protection model, where *principals* (e.g., users, groups, services, etc.) execute processes that perform *operations* (e.g., read, write, etc.) on *objects* (e.g., files, devices, etc.). The permissions of a principal to perform operations on objects are called the *access rights* of the principal. We call a set of access rights an *access control domain* or simply *domain*.

The paper is organized as follows. In Section 2, we detail the security requirements of some emerging applications. In Section 3, we define the problem of enforcing the access rights of an untrusted computation. In Section 4, we review related work. In Section 5, we present our system architecture. In Section 6, we define our access control model. In Section 7, we describe the architectural details and their implementation. In Section 8, we conclude the paper and present future work.

2 Example

At the University of Michigan, we are developing a system called the Upper Atmospheric Research Collaboratory (UARC) [6]. UARC provides its geographically-distributed users with a wide variety of applications which support collaborative analysis of atmospheric test data. For example, a test data viewer enables users located in the United States and Europe to share views of the atmospheric test data and jointly observe and annotate the views and the data. Most of the applications in UARC involve synchronous interaction among users.

Since collaborative applications have many common requirements, we are building an application-independent infrastructure called the Collaboratory Builders' Environment (CBE) [15]. The CBE provides services that are common to collaborative applications, such as replicated object management [22], multicast communication [11], and security. The goal of the security infrastructure is to provide the security services necessary to support a variety of collaborative applications.

UARC applications are to be implemented as downloaded executable content. When a user wants to use an application, the user downloads and executes the associated content on his local machine. Therefore, the user assumes the role of the downloading principal in the mobile agent architecture. In order to properly control the access rights of the user, the CBE infrastructure must be able to: (1) identify the source of the application and (2) assign that application appropriate access rights. For a UARC application, the set of UARC developers is the expected remote principal. Assigning the access rights is a difficult problem because a UARC application, such as the test data viewer, may need the rights to:

- Obtain test data from the remote UARC data server
- Communicate actions to the remote collaborators who share the view
- Read local setup files and environment variables for executing the application
- Read past analysis session files to replay them
- Store the analysis session for future replay
- Execute existing applications (e.g., editors, numerical analysis programs, etc.)

Therefore, we presume that an access control infrastructure must be able to control access to a variety of objects including file system objects, sockets, environment variables, applications, URLs, and network services.

Since the CBE supports collaborative applications, we must prevent collaborator actions from causing security problems. In the CBE, applications are supported by a replicated process architecture where each collaborator has: (1) a process on his local machine that is executing the application; (2) the application itself; (3) shared state which is to be consistently maintained by the application; and (4) private data that is unique to each collaborator. When a principal performs an action on the shared state, this action is multicast to each collaborator and executed at each site (i.e., if the principal is permitted to perform that action). Therefore, a replicated process architecture enables multiple principals to execute actions in a single process. Security problems arise because different principals have different rights in the collaboration. For example, some principals administer the collaboration, others participate by modifying the shared application state, others can make limited changes to the shared state, and some can only view the shared state. Since each collaborator has a copy of the application, some users may run a modified version of the application which could try to perform, either accidentally or maliciously, unauthorized actions. Also, attackers on the Internet may attempt to disrupt the collaboration by sending malicious messages, deleting messages, replaying old messages, and modifying messages in transit.

3 Problem Definition

The problem of flexibly controlling UARC applications that are supported by the CBE is to: (1) authenticate the source of the content; (2) determine the least privilege access rights for the content given its source; and (3) enforce those access rights throughout the execution of the content.

Authentication involves: (1) identifying the source of content; (2) verifying the integrity of content; and (3) ensuring that the content meets its freshness requirements. Authenticating the source of applications differs from authenticating the source of collaborator content. An application may be composed of components from several sources, so each of the sources needs to be identified to determine the appropriate access rights. Also, freshness is not a factor as long as the appropriate version of the application is obtained and its integrity is verified. Collaborator content is analogous to a message in a distributed computation,

so only one source is typically responsible for the content. However, the freshness of collaborator content must be verified because a replay of this content may cause the shared state of the collaborators to diverge.

The access rights of content depend on: (1) the trust of the downloading principal in the authenticated remote principals responsible for the content and (2) the purpose of the content. If a downloading principal trusts a remote principal with an access right, then the downloading principal can grant the remote principal's content that right. Clearly, this set of rights may be more than any particular content requires, and, in addition, there may be special cases where a downloading principal is willing to grant temporary access to a private file to complete some transaction. Therefore, information about the access requirements of content and the downloading principal's willingness to grant additional rights are needed to determine the rights actually required by content. Unfortunately, this information is not known until runtime and is typically difficult for end users to provide without being vulnerable to spoofing attacks.

The access rights assigned to content should be enforced for the content and any programs controlled by the content. Obviously, the access requests made directly by content must be controlled. In addition, any existing programs or network services that can be controlled by the content (i.e., can be provided input by the content directly or indirectly) must also be restricted to the same or fewer rights. In general, access to the following types of objects needs to be controlled:

- **Application Objects**: A process can read and write objects in its memory
- **File System**: A process can perform read, write, and execute operations
- **Applications**: A process can execute another application by creating a process for that application
- **Processes**: A process can communicate with another process via pipes or the file system
- **Environment**: A process can execute its environment's scripts and read and write its environment variables
- **Network Services**: A process can communicate with a service on the network using a socket
- **Universal Resource Locators (URLs)**: A process can download and read system objects by specifying a URL

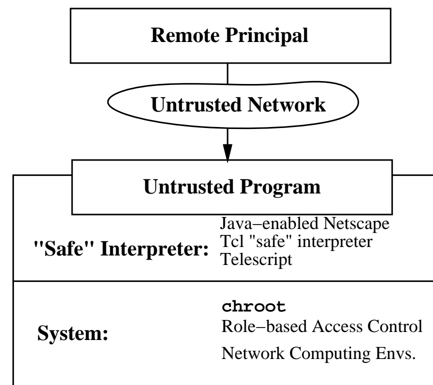

Figure 2: The current architecture for systems that utilize downloaded executable content

Operations on application objects must be authorized to ensure that the principal is permitted to modify the object. Access to file system objects should be limited to prevent unauthorized access to private objects. Also, access to objects that previously are available to trusted programs, such as environment variables and remote communication channels, must now also be controlled. These objects can be used by content to attack the downloading principal. For example, an attack on alpha version Java involves opening socket connections to an unauthorized third party to send information shared between the two legitimate principals [5].

4 Related Work

The current architecture for executing downloaded content is shown in Figure 2. In this architecture, executable content is downloaded to a principal who executes the program using one of a variety of "safe" interpreters, such as Java-enabled Netscape [10], Java's appletviewer, and Tcl's safe interpreter [3, 16]. To prevent attacks, these interpreters strictly limit the access rights of content. By default, all these interpreters only grant content (remote content in the case of Java) the right to communicate to only ports at the IP address of the content. The addition of read and write access rights to files is possible in the appletviewer by specifying those rights in the `~/.hotjava/properties` file. Unfortunately, the rights apply to all remote content. Also, these interpreters lack authentication, so all content must be assumed to have been downloaded from highly untrusted sources.

To grant rights to Tcl or Java content per remote principal currently requires the use of an in-

terpreter for trusted content or that custom applications be built. Interpreters for executing trusted content run content with the downloading principal's access rights, so access to objects not normally available to another principal, such as private files, are granted. We do not believe that content should be granted rights that another user would not normally have. On the other hand, custom applications require their own authentication, access control specification, and authorization infrastructure. Therefore, ad hoc security services need to be constructed which is an arduous and error-prone.

The Telescript engine [34] is another mobile agent interpreter. The Telescript engine differs from the interpreters described above in its use of *credentials* for authentication and *permits* for authorization. Credentials are cryptographic representations of the identity of the principal responsible for the content. Permits list the access rights of content. A permit can contain rights to another principal's (e.g., the downloading principal) resources. When content is downloaded, the downloading principal can deny rights that the content's permit grant, but this decision is ad hoc. Also, like other interpreters, the Telescript engine cannot control the execution of external software.

Operating systems can control the access rights of external software it executes, but current operating systems are not designed to flexibly restrict a principal's rights. For example, we showed in [12] that current file systems, such as Unix [24] and AFS [27], only provide limited mechanisms for a principal to dynamically restrict the access rights of one of his processes. Additionally, in Unix-based systems, the command `chroot` is available to limit the execution scope of a process to a file system subtree. However, `chroot` is cumbersome to use because of the need to transfer files to the restricted file system subtree, and, even worse, it cannot control remote communication by content. In fact, no current operating controls remote communication. Control of remote communication is typically provided by firewalls, but firewalls do not control access rights on a per process basis.

Recent research has yielded systems which provide support for defining limited access control domains, but it is not possible to generate a new domain at runtime. Role-based access control (RBAC) [1, 9, 33, 35] models permit a user to execute processes using different principals, called *roles*, which are associated with different access control domains. Thus, two processes run by the same user can have different access rights. However, to create these access control domains, most of these models require changes to the ACLs of all effected objects, so creating roles dynamically is prohibitively expensive. The Domain Type Enforcement (DTE) RBAC model uses the file system hierarchy to express domains more concisely. Therefore, we believe it is possible, from a performance perspective, to dynamically generate limited domains using DTE, but users and their processes are prevented from generating domains because DTE is used as a mandatory access control model. Other RBAC models permit evolution of access rights using rules [4, 17]. However, the rules, like the domains, must be specified in advance, but the rights of content depend on the goals of the content's application which are large in number and are often not known until runtime.

In [30], the DTE RBAC model is applied to controlling content. A domain that includes the objects needed by the browser to run and a "scratchpad," public directory are defined. While this permits an extension of current interpreter access domains, it still lacks the flexibility and per-content access control we desire. Also, control of remote communication is still lacking (however, IPC is controlled by DTE).

Another mechanism for granting limited rights to a process is delegation. Delegation is advantageous in that cryptographic credentials are used to represent the rights being delegated, but flexibility, standardization, and trust are problematic. Some systems, such as Taos [35], delegate rights via roles, so they suffer from the same flexibility limitations as RBAC. Kerberos version 5 [14, 29] provides a field for storing access control domains in the delegated credentials, so flexible delegation of rights over a variety of objects is possible. However, access rights must be transferred to servers in a language that the server can understand and enforce. Also, a trusted mechanism for transferring the rights to the server must be used. For example, a service controlled by the downloaded content cannot be given the responsibility to control file system objects.

5 Architecture

In the design of an architecture for controlling downloaded executable content, we make the following assumptions. First, we assume the existence of a public key infrastructure that be used to securely obtain the public key of any principal. Thus, any principal can verify the source and integrity of a message signed with a private key. Next, we assume that we can identify any I/O commands in the content language. This is necessary to control access to system objects. Next, we assume that the operating system has an unmodified trusted computing base, protects

process domains, and provides authentication of principals. This ensures that system software, such as cryptographic software, can be trusted, processes can only interact in controllable ways, and authentication of services is possible. Without trust in the operating system, it is not possible to build trusted applications that run on that operating system. A secure operating system, such as Trusted Mach [32], satisfies these requirements. Lastly, for applications involving three or more principals, we assume the existence of a secure group membership protocol, such as that described by Reiter [23], to ensure that all valid correctly-behaving principals share the same view of the application's group.

The following types of principals are involved in a downloaded content computation:

- **Downloading Principal**: The principal who downloads and executes the content
- **Remote Principals**: The principals responsible for the content
- **Application Developers**: Remote principals who provide content that implements applications that execute downloaded content from others
- **System Administrator**: A trusted principal who understands the structure of the downloading principal's system

The downloading principal trusts the system administrator and may have some trust in the remote principals (including the application developer). Thus, the downloading principal wants to grant only a subset of his access rights to a remote principal's content, and the system administrator is trusted to help the downloading principal define this subset. Note that the downloading principal may have different levels of trust (i.e., define different access control domains) in each remote principal. We assume that remote principals are trusted keep secrets from other remote principals with less privilege (e.g., private keys).

An architecture for flexible control of downloaded executable content is shown in Figure 3. This architecture consists of four levels: (1) local system services for access control; (2) a trusted, application-independent interpreter; (3) an application-specific interpreter (optional); and (4) an interpreter for executing downloaded content. The local system services provide operating systems services for controlling operations external to the interpreters. Also, system-specific information is made available for local system services (e.g., the file system structure).

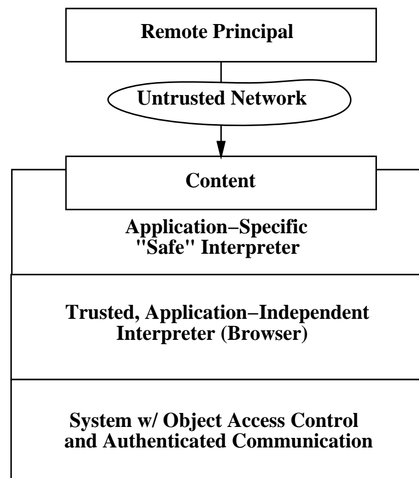

Figure 3: An architecture for flexible control of downloaded executable content: (a) system services control access outside the interpreter; (b) trusted, application-independent interpreters (*browsers*) control access to system objects by the less-trusted interpreters; (c) application-specific interpreters control access to application objects and determine the access rights of content within their domain; (d) downloaded content implements the actions of remote principals.

Trusted, application-independent interpreters (which we will refer to as *browsers* from here on) control access to system objects by the application-specific interpreters and downloaded content. An application-specific interpreter controls access to application objects and specifies the access rights of downloaded content within its limited domain.

The process for executing content using this architecture is shown in Figure 4. The remote principal sends a content message to the downloading principal which provides the content, the content type, and any authentication/encryption information. The browser receives this message and verifies the identity of the remote principal, the integrity of the content and content type, and the freshness of the message. If the verification succeeds, the browser determines which interpreter to execute the content. If the type refers to a known application-specific interpreter and the remote principal has the rights to execute content in that interpreter, then the content is run in that interpreter. The access rights available to the content are an intersection of: (1) the rights the downloading principal grants to the remote principal to execute the application; (2) the rights that the downloading principal grants to the application developer for the application; and (3) the rights that the application grants

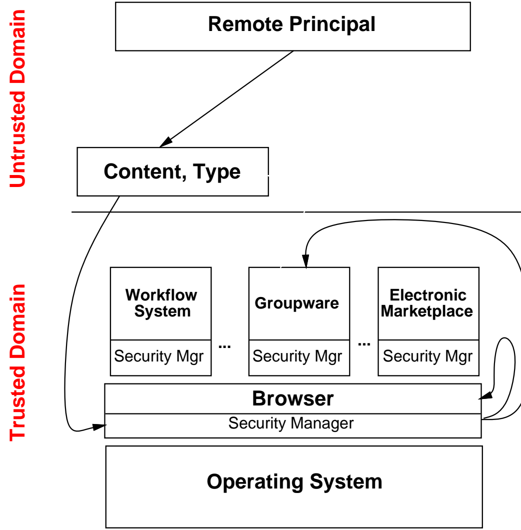

Figure 4: Content is downloaded to application-independent interpreter (called the *browser*) which authenticates the remote principal and finds the appropriate interpreter to execute the content. If the remote principal is authorized to execute content in that interpreter, then the content is executed.

to the remote principal. In addition, the application-specific interpreter has some leeway in transforming the access rights of content within its access control domain. This may result in either rights becoming available or unavailable from content, but the browser always enforces the intersection described above.

Content is executed as shown in Figure 5. After the browser authenticates the content and sends the content to be run in the appropriate application-specific interpreter as described above, the content is assigned to content interpreter based on the identity provided by authentication. This content interpreter has access rights consistent with the identity of the remote principal. Any controlled operation run by the content is authorized by the appropriate interpreter and if authorization is granted then the operation is executed. Operations are executed in the interpreter trusted with the operation. For example, file open is restricted to the browser. Also, the result returned to the content interpreter must not enable the content interpreter to gain additional rights. A read-only file handle is returned to the content interpreter in this case.

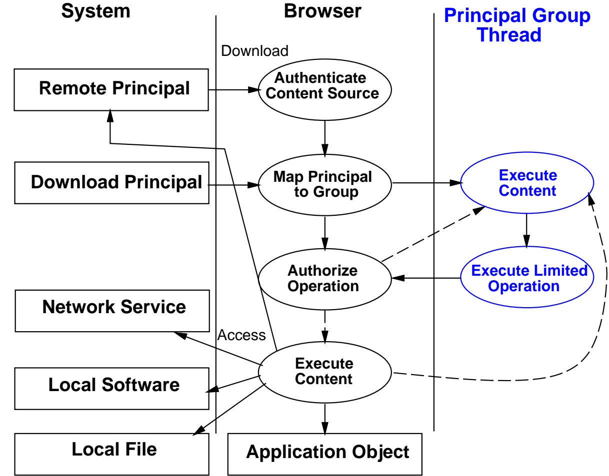

Figure 5: **Content Execution Protocol**: (1) Remote principal downloads content to browser; (2) Browser assigns content to principal group in application-specific interpreter (not shown); (3) content executes a limited operation which cause authorization in the browser; (4) if authorized the browser access system objects on behalf of content.

6 Access Control Model

Access control is the goal of this system, so we define an access control model for expressing the access control requirements of remote principals. This access control model uses the following concepts:

- **Definition 1**: A *principal group* is a set of principals with the same access rights.
- **Definition 2**: An *object group* is a set of objects which are grouped for expressing common access control requirements.
- **Definition 3**: *Access rights* of a principal group are the defined by two types of specifications:
  - **Domain Rights**: A tuple, $\{p_group, allowed_ops, object_group\}$, which describes a set of operations which the principal group (p_group) can perform on an $object_group$.
  - **Exceptions**: A tuple, $\{p_group, precluded_ops, objectgroup'\}$, which describes a set of operations which the principal group (p_group) is precluded from performing on an $object_group'$.
- **Definition 4**: A *class operation* is a set of operations that a principal group can perform on

objects belonging to that class given authorization via object group rights.

- **Definition 5**: A *transform* moves an object from one object group to another.

The relationship between these concepts in the access control model is shown in Figure 6. Individual principals are aggregated into a *principal group* if they all have the same access rights. Objects are also aggregated into groups called *object groups* for expression of a common access control requirement. The *access rights* of a principal group are described by its domain rights and exceptions. *Domain rights* describe the rights permitted to the principal group, and *exceptions* describe rights which are precluded from the principal group. This permits access rights to be defined concisely for a large set of objects while permitting some objects in the group to override those rights. Thus, fewer object groups should be necessary. Note that exceptions take precedence in the authorization mechanism, so an operation is granted only if a domain right grants the operation and no exception exists that may preclude the operation.

The expression of object groups is fairly straightforward except in the case file system objects. Normally, object groups are simply unique names that refer to a set of objects. For example, suppose we permit content from any member of the UARC scientists group to communicate with any other member of the same group. We would express this rights as a domain right $\{scientists, communicate, scientists_certs\}$. Definition of the UARC scientists group is simply a listing of their public key certificate identifiers (a more general specification is proposed in PolicyMaker [2]). File system objects are different because these objects already have access rights. We would like to use these existing rights to express a subset of the downloading principal's rights. We describe a set of file system objects by a path and a sharing type. A *path* indicates the domain in the file system hierarchy in which object group resides. For example, a path of $\sim$ indicates all objects in the downloading principal's home directory or any of its descendant directories. A *sharing type* indicates objects that are accessible to the same classes of principals. Examples of sharing types are:

- **All**: all objects that the downloading principal can perform the operations allowed (in the domain rights).

- **Intersection**: both the downloading principal and the remote principal objects can perform the operations allowed.

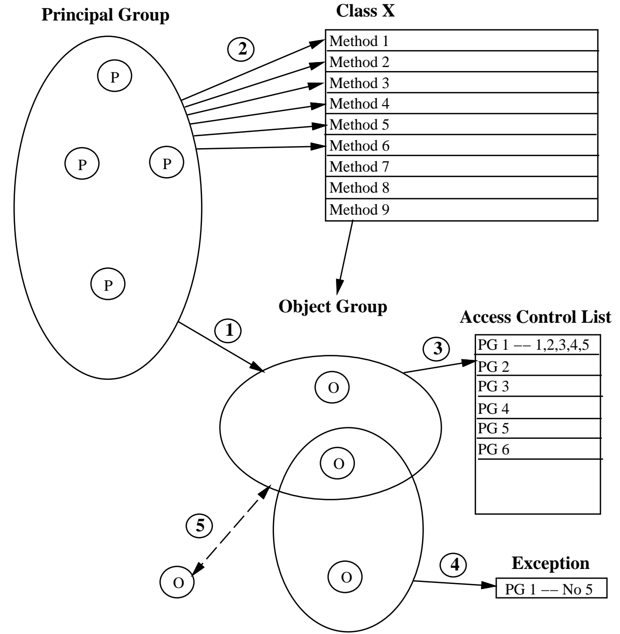

Figure 6: Access control relationships among entities: (1) **principal groups** can execute operations on objects that belong to classes (class operations); (2) principal groups can execute operations on objects in **object groups** (accessible objects); (3) **domain rights** specify a principal group's permissions to execute operations on an object group; (4) **exceptions** prevent operations from being executed on objects in an object group (even if a domain right is granted); and (5) **transforms** enable an object to join or leave a group.

- **Public**: The operations allowed for that are available to local users

- **Foreign**: The operations allowed for that are available to foreign users

- **None**: No objects

- **New**: Can only create, examine, and modify new objects in the domain

Using our access control model, the access rights for a UARC scientist using the UARC application described in the Example Section are (scientist is abbreviated as `sci`):

- $\{sci, communicate, uarc_cert\}$

- $\{sci, communicate, scientists_certs\}$

- $\{sci, r, uarc_env\}$

- $\{sci, r, \sim/.uarc : public\}$

- except: $\{sci, rw, \sim/.uarc/prefs : all\}$
- $\{sci, rw, \sim/.uarc/sessions : new\}$
- $\{sci, rx, /usr/bin/num_analysis : public\}$

UARC scientists' content can communicate with the UARC server and other UARC scientists. Also, environment variables for uarc can be used by the UARC scientists content. UARC scientists content can use downloading principal's file system to obtain public UARC setup information, except for the downloading principal's personal preferences. Also, existing, public analysis sessions in `~/.uarc/sessions` can be replayed given this specification. However, new sessions can only be saved into new files in that directory. Finally, this specification permits the content to execute a numerical analysis package.

The other access control model objects are used to help developers build their applications to manage access rights. The rights of principals to perform operations on classes of objects are defined using *class operations*. Class operations describe the set of operations that a principal group may ever perform. Class operations are strongly-typed (i.e., the types of the arguments must be well-defined). Also, casting of objects is restricted to ancestors or descendants in a class hierarchy to prevent unauthorized access by using a remotely different class's version of an operation with the same name. In addition, access control predicates on any argument in an operation can be added, which is useful for authorizing the execution of external software. Lastly, we define *mandatory class operations* as operations which can always be run (given that an object handle is available). These operations do not require authorization. All other controlled class operations (classes for which class operations are defined) require authorization.

The ability to transform access rights as the application state changes involves adding or removing objects from object groups. For example, when a downloading principal loads test data into a view, the window object should be loaded into an object group for shared windows. Thus, in the access control model, certain operations are associated with changes in object group membership, called *transforms*. The idea that operations modify the set of access rights, and that high-level specifications should be used to represent these access rights is adapted from Foley and Jacob [7]. However, our implementation specifies changes in rights rather than the complete set of rights. For example, the operation `x.load` returns a window object `y` upon load of test data `x`. In order to automatically add new windows to the shared window object group, developers specify that `y=x.load : swg.add(y)` where `swg` is the shared window object group. In order to execute this transform, the principal must have permission to run both `x.load` and `swg.add`, so restricting this operation such that only authorized principals can make access rights modifications is straightforward.

This model is influenced most strongly by the access control models of Hydra [36] and DTE [1]. Like Hydra, access control on the operations of abstract data types are possible, but access rights in our model are associated with principals rather than the content itself (procedures in Hydra). Therefore, management of rights is simpler and consistent with our applications. Like DTE, access rights information is aggregated to eliminate redundant specification. However, unlike DTE we use existing access control information to specify the domains. The sharing types permit us to implicitly restrict the access rights to a subset of the downloading principal's rights. Thus, verification that the domain is indeed a subset of the downloading principal's domain is not necessary. Also, we extend the application of DTE-like specification to non-file system objects.

7 Details and Implementation

The main tasks of the architecture are authenticating content, determining the access rights of content, and enforcing those access rights. Content must be authenticated to determining its source and verify its integrity and freshness. Determination of access rights involves combining trusts in the application developer and remote principal with the access rights implied by the application-specific interpreter's current state. Enforcement of these rights must be possible for the entire computation spawned by the content. This includes any external software or network services that are executed by the content. In this section, we detail these tasks and discuss their implementation.

We have developed a prototype architecture using Tcl version 7.5. We chose this version of Tcl because of its interpreter model and security model [16]. The interpreter model of Tcl 7.5 permits a hierarchy of interpreters to be constructed where one can be the master of another which is referred to as the slave. Therefore, we can build a hierarchy of interpreters where: (1) the browser is the master of the application-specific interpreters and (2) application-specific interpreters are masters of the content interpreters. In addition, Tcl 7.5 implements the "safe" interpreters of the style of Safe-Tcl [3]. Therefore, security is enforced by removing unsafe commands from

slave interpreters. However, some unsafe commands may be needed to complete the application, so Tcl provides a mechanism for masters to execute unsafe commands on behalf of slaves (called `alias`). Therefore, the browser implements commands that operate on system objects and the application-specific interpreter implements commands that operate on application objects. These unsafe actions cannot be added by the content interpreters, so they must use the commands in the masters and authorization is enforced.

7.1 Downloading Content

Content is downloaded in a *content message*. A content message has three purposes: (1) to provide the content; (2) to determine which interpreter should execute the content; and (3) to authorize the execution of the content in that interpreter. Therefore, a content message is defined as a quintuple $m = (r, c, t, a, s)$ where: (1) r is the identity of the remote principal; (2) c is the content; (3) t is the content type; (4) a is the content's authentication information; and (5) s is an optional session identifier (for computations involving multiple interactions). The content type is a MIME type of the form `browser/application` where `browser` refers to the fact that the content is controlled by the browser interpreter, and `application` identifies an application-specific interpreter in which the content will be executed. The `.mailcap` file is used to store this information. The `.mailcap` may be subject to denial-of-service attacks, so a secure (i.e., root-owned) `.mailcap` should be used. Therefore, the mapping of content type to application-specific interpreter is likely to be maintained by a system administrator.

The authentication information a is a double $a = (n, i)$ where n is a nonce and i is the integrity verification field. The nonce is a random value provided to prevent an attacker from replaying an old message. If an attacker could replay an old message, an attacker could modify the state of a transaction. We use a timestamp and a per-remote-principal counter as the nonce. The timestamp represents a recent, fresh session (i.e., no two sessions started by the same principal can use the same timestamp), and the counter indicates whether the message has been run in the session by this principal. i is the integrity verification field which is used to verify the integrity of the message. Using a public key algorithm, such as RSA [25] or DSA [19]), i is a digital signature of a message created from the concatenation of the remote principal name, content, type, and the nonce.

Some applications involve a number of interactions, so to improve the performance of the message authentication in these applications, symmetric key cryptography should be used. It is fairly straightforward for two principals to exchange a symmetric key given that the two principals have securely obtained each other's public keys (e.g., SSL protocol [8]). If symmetric cryptography is used, i is a message authentication code (MAC) computed with a hash function (e.g., SHA [18]) rather than a digital signature (the same message is used, however). We prefer using a hash function MAC over a DES CBC MIC [28] because a DES CBC MIC is not guaranteed to be collision-free.

In addition, some applications involve more than two principals and a number of interactions, so using symmetric cryptography for authentication becomes more complicated. In general, an n-principal computation requires $O(n^2)$ keys total and $O(n)$ encryptions per message because each pair of principals must share a secret in order to be certain of the identity of the sender. The fact that in many applications multiple principals have the same access rights can reduce the number of keys and encryptions somewhat. Members in a principal group can share a symmetric key, but a key is required for each pair of group and non-member. Therefore, if g is the number of principal groups and n is the number of principals overall, the maximum number of keys required is $g * (n - n/g) + g$ [1] and the number of encryptions per message is the number of non-members + 1 (for the group). Note that if the number of principal groups is small (i.e., limited by a known constant), which is normally the case in a synchronous collaboration, the number of keys remains bounded by $O(n)$. Therefore, as long as n is less than 100, symmetric cryptography is still preferred.

The browser procedure for authenticating an untrusted program is shown in Figure 7. This procedure receives pointers to the remote principal's name, the content, the content type, and the authentication information from the content message. A pointer to a session object is obtained if a session identifier is provided. A session object is a tuple, $s = (auth, members, counter, keys, t)$ where: (1) $auth$ is the authentication type of the session (either `public` or `symmetric`); (2) $members$ is the set of principals involved in the session; (3) $counter$ is an array of message counters for each remote principal; (4) $keys$ is a

[1] This the number of keys required if there are 2 or more group members in each group. If $g = n$ (each principal is in a unique group), then the number of keys required is fewer because many keys are redundant. An internal key for the group is not needed because each principal is a group, so the number of keys is reduced by n. Also, the remaining half of the keys are redundant because, an extra key is added for every pair of principals because keys are distributed between each group and non-member normally. Therefore, the number of keys is $(n * (n - n/n))/2 + n - n = n(n - 1)/2$, as expected.

```
Authenticate(r c t a s)
    r is the remote principal name
    c is the content
    t is the content type
    a is the authentication information
    s is a session
    /* First, get the key for the principal from
    the session or use r's public key */
    If (s is null) then k = public_key(r)
    else k = key(s,r)
    /* Next, If session is unknown or r's counter = 0
    then use public key algorithm to compute y
    if ((s is null) || (s.counter[r] = 0)) then
        y = f_public(r, c, t, a.n)
    else y = f_s.auth(r, c, t, a.n)
    /* Finally, determine if y is the same as a.i for
    the expected counter value s.counter[r]
    If ((y != a.i) ||
        (s.counter[r] != a.n.counter)) then error(s,r)
    increment s.counter[r]
    return r
```

Figure 7: Authentication procedure in browser interpreter

mapping of principals to keys stored by the downloading principal; and (5) t is the content type. First, this procedure determines the key to use to authenticate the message using the remote principal's name and the *keys* map. If there is no session then r's public key is used. Next, the authentication function is determined from the session. If the session is null or this principal is new to the session the public key algorithm is used. The authentication value $f_{s.auth}(r, c, t, a)$ is compared to $a.i$ from the content message. If they match, then the message is authenticate to have been from r.

Once r's content is authenticated, the name of the application-specific interpreter for the content is retrieved from the `.mailcap` file using the content type t (hereafter called interpreter t). Next, we determine if the remote principal is permitted to execute content in interpreter t. If the remote principal is mapped to a principal group in interpreter t, then the content can be executed. This mapping is stored in file called `principal-groups.`t. This file must be write-protected. Once it is determined that the remote principal's content can be executed, then the appropriate keys can be distributed to the remote principal, if necessary. First, if there is no session object, then interpreter t is queried for its authentication type. If there is a session object, it is examined to determine if a keys for the principal group have been assigned. If not keys for the group and between the group and any non-members are generated. Also, the remote principal needs a key for each group it is not a member of.

7.2 Determining Access Rights

Once the browser has determined that r's content can be run in interpreter t, it is then necessary to determine the content's access rights. We first describe the ways in which access rights are expressed. The goal is to minimize the amount of specification required by the downloading principal, yet to still enforce least privilege access rights. Next, we describe how the rights are computed. The access rights of a remote principal's content in interpreter t active at state s are the intersection of: (1) the rights of interpreter t and (2) the rights of the principal group to which the remote principal is assigned in interpreter t at state s. Application-specific interpreters are responsible for deriving access rights within their domain, but the browser still enforces all system object accesses.

Application-specific interpreters are responsible for deriving the access rights of remote principals, but they lack information about either the exact names of system objects, such as the path names of files, or the identities of trusted remote principals. Therefore, logical objects are created to represent these system-dependent objects. Remote principals are represented by principal groups and system objects are represented by object groups, as defined in the access control model. Thus, the access rights of the interpreter t and any remote principals is based on the mapping of these principals to principal groups and the mapping of object groups to actual system objects.

We first describe how the access control domain of interpreter t is specified. Object groups are used to express the types of the objects to which interpreter t requests access. For example, suppose interpreter t implements the test data viewer application described in the Example Section. Thus, interpreter t requires access to the following object groups:

- UARC Server Socket
- Scientist Sockets
- Administrators Sockets
- UARC Environment Variables
- UARC System Files
- UARC Analysis Data
- Numerical Analysis

- Emacs Text Editor

Thus, the following rights are specified for interpreter t using our access control model (UARC developers are abbreviated by `dev`):

- $\{dev : t, communicate, uarc_servers\}$
- $\{dev : t, communicate, uarc_scientists\}$
- $\{dev : t, communicate, uarc_admins\}$
- $\{dev : t, r, uarc_environment\}$
- $\{dev : t, r, uarc_system\}$
- $\{dev : t, rw, uarc_data\}$
- $\{dev : t, rx, external_sw\}$

The file `/usr/local/uarc/system/mapping.`t is used to define the mapping from object groups to system objects for interpreter t. The contents of this file should be based on analysis (e.g., use in system that can log process actions) of the interpreter t. The result is an access control domain for interpreter t which describes the objects which it is trusted to protect. Therefore, significant trust may be placed in an application-specific interpreter. We do not think this is unreasonable given that traditional applications are typically run with the users' complete trust. The browser and operating system must be able to enforce the specified rights to ensure that unauthorized rights cannot be obtained by interpreter t, however.

We specify the following mapping for interpreter t in the file `/usr/local/uarc/system/mapping.`t:

- $uarc_servers := uarc_server_cert$
- $uarc_scientists := scientists_certs$
- $uarc_admins := uarc_admin_certs$
- $uarc_environment := uarc_environment$
- $uarc_system := /usr/local/uarc/system : public$
- $uarc_data :=\sim /.uarc : public\}$

  except: $w, \sim /.uarc/system : public$
- $external_sw := /usr/bin : public$

  except: $rx, /usr/bin/mail : all$

This mapping results in the following access rights for interpreter t:

- $\{dev : t, communicate, uarc_server_cert\}$
- $\{dev : t, communicate, scientists_certs\}$
- $\{dev : t, communicate, uarc_admin_certs\}$
- $\{dev : t, r, uarc_environment\}$
- $\{dev : t, r, /usr/local/uarc/system : public\}$
- $\{dev : t, rw, \sim /.uarc : public\}$
- except: $\{dev : t, w, \sim /.uarc/system : public\}$
- $\{dev : t, rx, /usr/bin : public\}$
- except: $\{dev : t, rx, /usr/bin/mail : all\}$

Thus, interpreter t can communicate with the UARC server, scientists, and the UARC administrators. Also, it can access UARC environment variables and the files in the `~/.uarc` directory. The `/usr/local/uarc/system` and `.uarc/system` directories contain the secure files needed by the browser to execute the application, so write access to these files is precluded from interpreter t. Also, in this specification, permission is granted for it to execute all software in `usr/bin` except the `mail` program. The mapping of local groups to objects is done using `~/.uarc/groups` (for all applications).

The access rights of the remote principals in interpreter t is specified by defining the rights for principal groups in the interpreter and assigning remote principals to principal groups. The access rights for a principal group are specified by the developers in terms of object groups, as described above for the interpreter itself. Suppose the UARC application has two principal groups: `lead scientists` and `scientists`. Lead scientists can save new analyses on the machines of all collaborators, run the numerical analysis package, and replay and annotate analyses. Regular scientists can only replay and annotate analyses on behalf of the downloading principal. Therefore, the application developer specifies the access rights of these principal groups as follows (not including the communication and environment rights) (lead scientists are abbreviated `lsci`:

- **Lead Scientists**
  - $\{lsci, rw, new_analyses\}$
  - $\{lsci, rw, annotations\}$
  - $\{lsci, r, old_analyses\}$
  - $\{lsci, rx, num_analysis\}$
- **Scientists**
  - $\{sci, rw, annotations\}$

– $\{sci, r, old_analyses\}$

The system administrator and/or the downloading principal defines a mapping between these object groups and a domain of real system objects. This permits the browser to describe the maximal domains of principal groups to the downloading principal, so he can choose a principal group for a remote principal. The mapping is specified in the file `object-group.`t in the write-protected `~/.uarc/system` directory (or perhaps in the `/usr/local/uarc/system` directory for a default mapping). For example, UARC's old analyses are mapped to `~/.uarc/analyses` directory (similarly for annotations). However, initially there are no new analyses. A principal must create a new analysis in the `~/.uarc/analyses` directory to write to it. The numerical analysis package is mapped to `/usr/bin/num_analysis`, but note that the developers must know the browser's API for executing it in order to correctly initiate its execution. This is a problem because we do not want to evaluate arbitrary commands in the browser. Evaluation of arbitrary code in a trusted interpreter can result in undesirable modification of the trusted interpreter's security data, so this cannot be allowed. Therefore, the application-specific interpreter requests execution of the logical numerical analysis package object, and the browser maps the logical object to the actual system object.

Once the files are mapped to objects groups, the initial access rights of the principal groups are well-defined. For example, the rights below result for the two principals:

- **Lead Scientists**
  - $\{lsci, rw, \sim/.uarc/analyses : new\}$
  - $\{lsci, rw, \sim/.uarc/annotations : public\}$
  - $\{lsci, r, \sim/.uarc/analyses : public\}$
  - $\{lsci, rx, /usr/bin/num_analysis : public\}$
- **Scientists**
  - $\{sci, rw, \sim/.uarc/annotations : public\}$
  - $\{sci, r, \sim/.uarc/analyses : public\}$

Now, the mapping from remote principals to principal groups can be defined in the write-protected file `~/.uarc/system/principal-group.`t. Note that if consistency is a requirement of the application, then all collaborators must agree on the principal group assignment. This is where a secure agreement protocol is necessary.

The actual rights of the remote principals depend on the actions that have been taken by the downloading principal in interpreter t. The interpreter uses the access control model's transforms to modify the access rights of principal groups implicitly based on operations by the downloading principal. For example, if a downloading principal loads or joins an analysis, then the remote principals are granted the right to update the associated annotation file on the downloading principal's system. The transform is `x.load : uarc_annotations.add(x.annotations)` where `x` is an analysis.

Therefore, the actual access rights of a remote principal's content are the rights which are delegated by the application-specific interpreter on behalf of the downloading principal that are within both the access control domains of the remote principal's principal group and the interpreter itself. Therefore, if the interpreter is malicious it will only affect files in its domain. If only the content is malicious, then it can only affect the dynamically-generated domain. Therefore, authentication of application-specific interpreters is vital, to obtain assurance that the interpreter is being granted the proper trust. We use the BETSI protocol to authenticate application-specific interpreters [26].

We would prefer that the downloading principal delegate rights to the application-specific interpreter and remote principals using the browser because it is trusted. However, we want delegation to be implicit relative to some application rather than require the downloading principal to specify access rights at runtime. The problem is to get the downloading principal's content interpreter to run with the application-specific interpreter's transforms securely in the trusted browser. Because the application developer is not completely trusted, allowing this code to be run in the trusted browser is problematic. Further work is needed here.

Since the application-specific interpreter provides access control information to the browser, authorizes access to application objects, and transforms the access rights of principal groups, this interpreter will require some security infrastructure. The architecture provides a service to modify application-specific interpreters to add security infrastructure given the access control requirements of the interpreter. First, a call is made to collect the access control predicates for the operation. There is always one predicate to test whether the operation is permitted on the object (i.e., first argument). Next, a call to the authorization procedure (described below) is added after the access control of each controlled operation (class operation that is not mandatory, see the Access Control Model Section). Finally, the transform operation is added to the end of each operation with a transform specification. If there are multiple they are ordered as

```
operation(args)
    args is a list of operation arguments
    arg_types is a list of the types of args
    arg_ops is a list of the ops to be run on args

    arg_types = types(args)
    If operation is not mandatory
        then collect predicate(operation, first(args))
            into arg_ops
    Foreach arg in args
        collect access predicates(operation, arg)
            into arg_ops
    If (authorize args arg_types arg_ops)
        then results = execute(procedure, args)
    Foreach transform in transforms
        apply-trans(transform,args,results)
```

Figure 8: Generated operation with authorization and transformation calls

specified (order may be important). An example of a modified operation is shown in Figure 8. Finally, an operation is added for obtaining the access rights of any principal group.

Content interpreters for the principal groups can also be generated in advance. However, the content interpreter cannot have actual implementations of controlled functions, otherwise a malicious content provide could override them with versions that circumvent authorization, for example. Tcl provides a mechanism called `alias` which enables an interpreter to call a command in another interpreter. Therefore, aliases are provided for all controlled operations in the content interpreter, so trusted implementations of the operations can be executed in the browser (for accessing system objects) and the application-specific interpreter (for accessing application objects).

7.3 Enforcing Access Rights

Lastly, the architecture provides support for enforcing the access rights specified above while executing content and any external software and network services used by the content. The browser: (1) authorizes content operations; (2) provides safe implementations of these operations; (3) enables controlled execution of external software and network services.

The actual authorization procedure is shown in Figure 9. The access rights of content to system objects specified in the previous section are stored in the browser. For each `arg`, the browser determines the object group of the `arg` and `arg_type`. The browser

```
authorize(args arg_types arg_ops)
    For arg in args, arg_type in arg_types,
        ops in arg_ops
        Find object group for arg and arg_type
        For op in ops
            If domain rights grant op
                AND no exception precludes op on arg
                then continue
                else return FALSE
    return TRUE
```

Figure 9: Authorization procedure

then determines whether the object access rights permit all the operations in `arg_ops`. Developers are not required to specify the operations on arguments because when another procedure is called, the subsequent calls will be authorized as well. However, specifying argument operations are useful to catch bugs in the procedure call and to restrict the access rights of external software. Typically, the only operation that is authorized is the application of the procedure to the first argument, the object identifier (i.e., the first argument in an object-oriented method indicates the object for which the procedure is intended). To authorize an operation on an object, the browser must be able to find: (1) a domain right that grants permission to perform this operation on this object and (2) that no exception exists which precludes performing this operation on this object. Note that we had to create object identifiers for Tcl objects since Tcl is not object-oriented.

Once authorized, the browser can execute the procedure. A procedure consists of two kinds of code: (1) calls to other procedures and (2) accesses to controlled objects. A browser executes calls to existing procedures in the content interpreter using the *`slave`* `eval` *`args`* call in Tcl. This ensures that all calls to other controlled procedures will be authorized using the principal group rights. Operations on controlled objects within the procedure are assumed to be authorized if access to the procedure is authorized. These commands must be run in the browser because only the browser has access to the systems object. It is hard to automatically distinguish from existing procedures in Tcl because Tcl is not object-oriented. Therefore, these methods are generated manually at present. We expect that the use of an object-oriented language would enable automatic generation of these procedures.

The browser provides safe implementations of `open`, `socket`, `env`, and `exec` operations. All these

implementations are fairly straightforward except for `exec` which we describe in detail below. The other operations are implemented by a call to authorize which checks the content access rights prior to calling the actual operation. Since these commands are only available in the browser and are protected by the authorization operation, only authorized accesses to system objects are possible. An example is the implementation of the `open` command in [13] (called `safe_open`).

The browser also provides support for controlled execution of external software and network services. When an operation for executing executing external software, the browser must: (1) authorize this execution and prepares for a safe execution; (2) determine a limited access control domain for executing the software or service; (3) transfer the domain specification to the system responsible for controlling the software; and (4) execute the software. This is not a easy task because current operating systems do not support dynamic modification of a principals access rights nor do they control remote communication.

In general, software execution is authorized by determining if the remote principal (i.e., principal group) has execute rights for the first argument in the `exec` call. If so, then the `exec` operation executes the software using the current rights of the principal group (described further below).

If the software should be further restricted or if we want to be sure that the software can be executed successfully, then we suggest the creation of classes for software to represent this information. Software classes contain strongly-typed "constructors" for executing the software. Also, access control predicates can be assigned to the arguments in the call. These predicates should authorize operations required of arguments as well as the values of arguments. For example, the command `cat x | xterm -e` pipes file `x` into a new xterm and executes it. We may want to prevent an argument of `cat` from being a pipe. Also, we'd like to determine if we have the rights to read file `x`. In addition, the access control predicates specify the rights required to execute the software (excepting some start-up privileges which can be specified separately), so the execution of software can be limited more strictly than the content.

Once the software execution is authorized and the access rights have been determined, it is necessary to convert the rights to a form that can be enforced by the operating system. Unfortunately, current operating systems are not designed to permit a principal to run a process with limited access rights. We describe a technique for limiting the access rights of a service in the Distributed Computing Environment (DCE) [21]. DCE has the advantage that its authorization model is very flexible. DCE associates ACL managers with services to implement the authorization mechanism of those services. Therefore, each service can use its own ACL manager. We define an ACL manager for implementing our authorization mechanism which will be used by DCE's distributed file system (DFS).

We first define our DCE access control model. DCE uses an extended Unix-style access model where foreign users join the standard owner, groups, and others principal types. Modification of access rights involves changing: (1) the user identity (UUID); (2) the groups identities and operations: `group_obj`, `group`, and `foreign group`; and (3) the operations that are permitted to the general classes: `other_obj`, `foreign_other`, and `any_other`. In addition, if intersection rights are granted, the remote principal and the downloading principal must be stored to determine if both have a specific right. The modified DCE model consists of the following fields and values (for a domain right):

- **Root**: The root path name for the rights
- **User**: Either the downloading principal (sharing type = all), `nobody@local` (public or intersection with local principal), or `nobody@foreign` (foreign or intersection with foreign principal), or none (none or new)
- **Group_Obj**: None
- **Groups**: Intersection of groups shared by the downloading principal and a remote principal (intersection)
- **Users**: Set of principals which must all have the right (intersection)
- **Masks**: ACL types and permissions granted to that ACL type (for all ACL types)
- **Communications**: List of principals or principal groups

Exceptions are specified similarly, but masks imply the rights precluded. Thus, the new ACL manager's authorization mechanism determines if an operation on an object is permitted by a domain rights specification by: (1) finding a domain rights specification whose root is an ancestor the the object; (2) determining if any instance of an ACL type grants access to perform that operation given the masks. For example, if the operation is write, then ACL type mask must permit write operations, and the instance of that type must have write permission on the object.

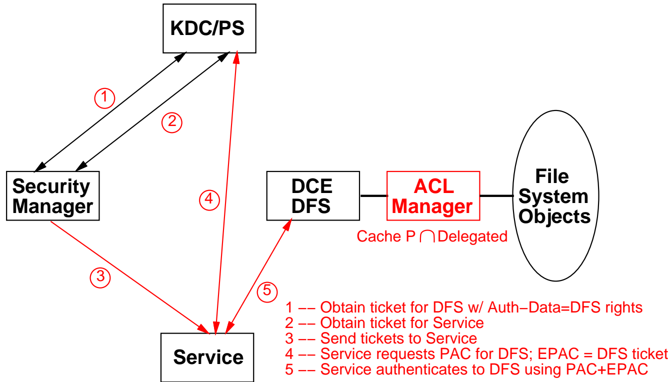

Figure 10: The protocol for a downloading principal to grant a service rights to his objects in DCE's Distributed File System (DFS)

DCE [21] utilizes an extended version of Kerberos version 5 tickets [14, 20] called Privilege Attribute Certificates (PACs) and further extensions to those tickets called Extended PACs (EPACs) for authentication. Kerberos documentation suggests that the `authorization-data` field be used to represent authorization information. However, DCE already uses that field for storing the UUID and groups for authentication, so we use the EPACs to store our authorization data. This enables a principal (the downloading principal via the browser) to grant rights to another principal (the service) which can be enforced by a third principal (the DFS). The protocol is shown in Figure 10. In this protocol, the browser obtains an EPAC for the DFS and a PAC for the service. Only the DFS can decrypt the EPAC and the EPAC is integrity-protected, so the DFS can verify that the EPAC is from the downloading principal. Therefore, the DFS can grant the downloading principal's rights to the service. Note that the service can only obtain the downloading principal's rights if the EPAC is presented to the DFS. The DFS ACL manager can interpret the rights, so it can authorize actions given those rights.

Note that using a DTE [1] makes the specification of rights much simpler, and enables the computation of intersections to be done at runtime. This is because access rights can be expressed more concisely. Therefore, we would prefer to use a DTE-style access control model in the future.

Limiting access to environment variables is easily implemented by restricting which variables are made available to the software in the `exec` call. Therefore, the transfer of application object rights and environment rights to the kernel is not necessary.

Control of other objects used by existing software, such as remote communications and network services, involves interaction between the kernel and network computing services. Therefore, the kernel should be able to interact with DCE as well to enforce strong authentication on all communications. As a proof of concept, this model of communication has been implemented in the Taos operating system [35]. In Taos, processes are associated with authenticators for proving the identity of the owner of a process to other processes. The Taos kernel is integrated with the authentication service, so all communication is authenticated. The ability to dynamically restrict the access rights of a Taos process is not possible given the current design. Similar services are also being developed for Trusted Mach [32] (upon which DTE is the access control model), so a nearly sufficient system environment is not that far off.

8 Future Work and Conclusions

We define an architecture that flexibly controls the access rights of downloaded executable content. In contrast to current interpreters: (1) strong authentication is used to verify actions by remote principals; (2) application-specific access control requirements are obtained which enable least privilege access to be enforced flexibly; and (3) these access rights can be enforced at both the application and system levels which enables comprehensive access control on the application.

By using this architecture, a variety of attacks can be avoided. For example, attacks in which data is sent to unauthorized principals (e.g., [31]) can be avoided because principals are authenticated and applications can identify the set of authorized remote principals. The use of services with poor security records, such as `sendmail`, can be avoided entirely. In addition, untrusted applications, such as ones that use `setuid`, can also be avoided by not granting rights to execute them.

Also, the architecture permits the use of rights not typically available in current interpreters. We provide specification models for developers to specify the access rights of principals in their applications and how rights can be transformed given user actions. The architecture provides a service to modify applications to integrate these security requirements with the application code. This enables developers to build systems which are less likely to contain security infrastructure bugs that can lead to attacks. Also, this enables application-specific requirements to be used to control

access. Thus, arbitrary restrictions, such as precluding communication with parties other than the originating host and preventing the execution of existing applications, can be replaced with more semantically-meaningful, application-specific access domains.

In the future, we plan to assist application developers in verifying the access rights granted to remote principals. That is, we want to be able to show application developers what a remote principal can do given a specific set of access rights. Thus, application developers can decide whether they have specified a satisfactory set of access control requirements.

References

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[3] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, 1994. Available via anonymous ftp from ics.uci.edu in the file mrose/safe-tcl/safe-tcl.tar.Z.

[4] E. Born and H. Stiegler. Discretionary access control by means of usage conditions. *Computers & Security*, 13(5):437–450, 1994.

[5] D. Dean, E. Felten, and D. Wallach. Java security: From HotaJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[6] R. Clauer et. al. A prototype upper atmospheric collaboratory (UARC). AGU Monograph: Visualization Techniques in Space and Atmospheric Sciences. In press.

[7] S. Foley and J. Jacob. Specifying security for CSCW systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 136–145, 1995.

[8] A. O. Freier, P. Karlton, and P. C. Kocher. The ssl protocol version 3.0. Internet Draft, March, 1996.

[9] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *IEEE Symposium on Security and Privacy*, pages 20–30, 1990.

[10] J. Gosling and H. McGilton. The Java language environment: A white paper, 1995. Available at URL http://java.sun.com/whitePaper/java-whitepaper-1.html.

[11] R. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rasmussen. Corona: A communication service for scalable, reliable group collaboration systems. In *Proceedings of the Sixth ACM Conference on Computer-Supported Cooperative Work*, November 1996. To appear.

[12] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[13] T. Jaeger and A. Prakash. Implementation of a discretionary access control model for script-based systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 70–84, 1995.

[14] J. T. Kohl and B. C. Neuman. The Kerberos network authentication service. Internet RFC 1510, September, 1993.

[15] J. Lee, A. Prakash, and T. Jaeger. A software architecture to support open distributed collabroatories. In *Proceedings of the Sixth ACM Conference on Computer-Supported Cooperative Work*, November 1996. To appear.

[16] J. Levy and J. Ousterhout. Safe Tcl: A toolbox for constructing electronic meeting places. In *The First USENIX Workshop on Electronic Commerce*, 1995. To appear.

[17] I. Mohammed and D. M. Dilts. Design for dynamic user-role-based security. *Computers & Security*, 13(8):661–671, 1994.

[18] National Institute of Standards and Technology, U.S. Department of Commerce. *NIST FIPS PUB 186, Secure Hash Standard*, May 1993.

[19] National Institute of Standards and Technology, U.S. Department of Commerce. *NIST FIPS PUB 186, Digital Signature Standard*, May 1994.

[20] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[21] Open Software Foundation. *Introduction to OSF DCE*, revision 1.0 edition, December 1992.

[22] A. Prakash and H. Shim. Distview: Support for building efficient collabroative applications using replicated objects. In *Proceedings of the Fifth ACM Conference on Computer-Supported Cooperative Work*, October 1994.

[23] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[24] D. M. Ritchie and K. Thompson. The UNIX timesharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[25] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] A. D. Rubin. Trusted distribution of software over the internet. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1995.

[27] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[28] B. Schneier. *Applied Cryptography.* Wiley & Sons, 1994.

[29] J. G. Steiner, B. C. Neuman, and J. J. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Conference*, pages 191–202, 1988.

[30] D. F. Sterne, T.V. Benzel, L. Badger, K. M. Walker, K. A. Oostendorp, D. L. Sherman, and M. J. Petkac. Browsing the web safely with Domain and Type Enforcement. In *IEEE Symposium on Security and Privacy*, 1996. Abstract for a 5-minute presentation.

[31] Computer Emergency Response Team. Java implementations can allow connections to an arbitrary host. CERT Advisory CA:96:05, 1996. Available at URL ftp://info.cert.org/pub/cert_advisories/CA-96.05.java_applet_security_mgr.

[32] Trusted Information Systems, Inc. *Trusted Mach System Architecture*, TIS TMACH Edoc-0001-94A edition, August 1994.

[33] S. T. Vinter. Extended discretionary access controls. In *IEEE Symposium on Security and Privacy*, pages 39–49, 1988.

[34] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper.

[35] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[36] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.