(9) Requirements of Role-Based Access Control for Collaborative Systems

Trent Jaeger and Atul Prakash

Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
Primary contact: jaegert@eecs.umich.edu, (313) 747-3780

Abstract

In many collaborative systems, users can trigger the execution of commands in a process owned by another user. Unless the access rights of such processes are limited, any user in the collaboration can gain access to another's private files; execute applications on another user's behalf; or read public system files, such as the password file, on another user's machine. However, some applications require limited sharing of private files, so it may be desirable to grant access to these files for a specific purpose. Role-based access control (RBAC) models can be used to limit the access rights of processes, but current implementations do not enable users to flexibly control the access rights of a process at runtime. We define a discretionary access control model that enables principals to flexibly control the access rights of a collaborative process. We then specify the requirements of RBAC models necessary to implement this discretionary access control model.

1.0 Introduction

We examine how the access control requirements of collaborative systems affect the design of RBAC models. A collaborative system or groupware is a computer-based system that supports groups of users engaged in a common task and that provides an interface to a shared environment [ELLI91]. Examples of collaborative systems include computer conferencing systems, multi-user editors, group decision support systems, and workflow systems. Collaborative systems permit multiple principals to execute commands on a single process, so the access rights of those processes must be limited to prevent unauthorized access to the process owner's system. RBAC models enable the access rights of a process to be limited, but current implementations do not support the dynamic, user-level access control required by collaborative systems. At present, RBAC is being used almost exclusively to develop mandatory access control (MAC) support, but we believe that within a MAC framework, RBAC models are also useful for discretionary access control (DAC).

Collaborative system processes must be run with limited access rights because multiple users can specify commands to a single process. For performance and fault-tolerance reasons, collaborative systems often use a replicated process architecture. In a replicated process architecture, each user has a process on his machine that maintains a copy of the state of the collaboration. Each command is executed independently by each replicated process. Therefore, collaborative processes execute commands written by users other than the owner of the process.

Unless the access rights of a collaborative process are limited, any user may obtain unauthorized access to the system objects owned by another user. For example, consider the use of mobile agents (also known as computational E-mail, command scripts, and enabled mail) as the mechanism for transporting commands to replicated processes (see Figure 9-1, *Collaboration Using Mobile Agent Systems*). First, the writer composes the agent's code to specify the command. Through some mechanism (e.g., HTTP or mail) the agent is sent to each of the other users, called the readers of the agent. When a reader reads the agent (number 2 in the figure), a process is created to execute the agent's code (or an existing process may be used). This process runs on the reader's machine and is owned by the reader, so the agent is executed with the reader's access rights. A malicious writer can use these additional access rights to read and write the reader's private objects; execute applications, such as mail, to masquerade as the reader to other users; and read the password file on the reader's machine. Note that this problem is endemic to any process that executes commands supplied by multiple principals.

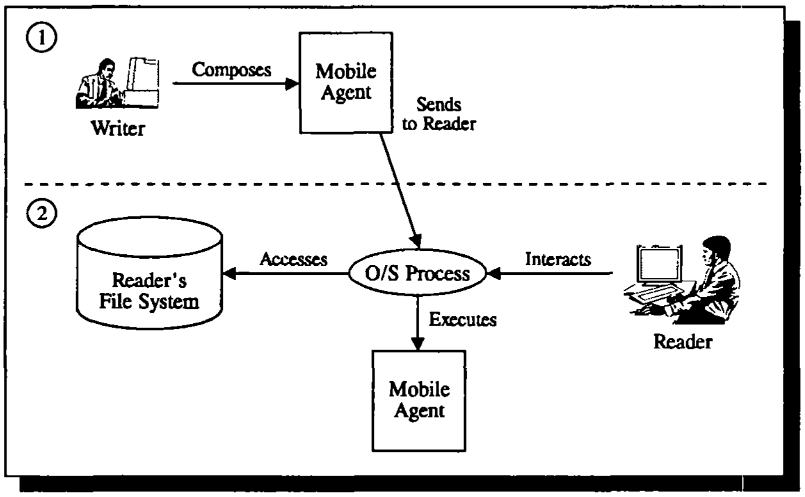

Figure 9-1. Collaboration Using Mobile Agent Systems

Current solutions are not suitable for enforcing the access rights of collaborative systems. To enforce access control, mobile agent systems preclude the agents from performing potentially useful types of actions, such as executing existing applications. For example, a Java [GOSL95] agent (called an *applet* in Java) cannot execute a non-Java application. Also, these systems use cumbersome approaches to control read and write access. Java's access model forces all concurrent agents to use the same access rights. RBAC models can flexibly specify the access rights of a process, but they do not enable users to dynamically specify access rights. At present, RBAC implementations are being developed only for MAC, so only system administrators can specify access rights. However, the appropriate access rights of a collaborative process are based on dynamic information, such as the set of collaborators or the purpose of the collaboration. Therefore, users or collaborative applications need to be able to limit access at runtime.

We present an access control model that can flexibly control the access rights of a collaborative process. Our model [JAEG95] is designed to restrict the access rights of a process at runtime. The model is simple because users or application writers need to know only a few common classes of objects to specify the access rights of a collaboration. Also, many of the access rights are deduced from the reader's current access rights, so their specifications are generally small. We then list the requirements of RBAC models necessary to implement this DAC model.

In Section 2.0 we present the problem definition of DAC on collaborative processes. In Section 3.0 we review related work. In Section 4.0 we define our DAC model. In Section 5.0 we list the requirements of RBAC models for implementing our DAC model. In Section 6.0 we outline future work and conclude the paper.

2.0 Problem Definition

We assume a conventional systems model, where *principals* (e.g., users, collaborators, etc.) execute processes that perform *operations* (e.g., read, write, etc.) on *objects* (e.g., files, printers, etc.). The permissions of a principal to perform operations on system objects are called the *access rights* of the principal in the system.

As an example, consider the DistEdit [KNIS90, KNIS93] system. DistEdit is a toolkit for building collaborative editors using a replicated process architecture. In a collaborative editing session, each user in the collaboration has an editor process (see Figure 9-2, *DistEdit System Architecture*). When a user edits the buffer, the DistEdit sends a copy of the command to each editor process to ensure the consistency of the

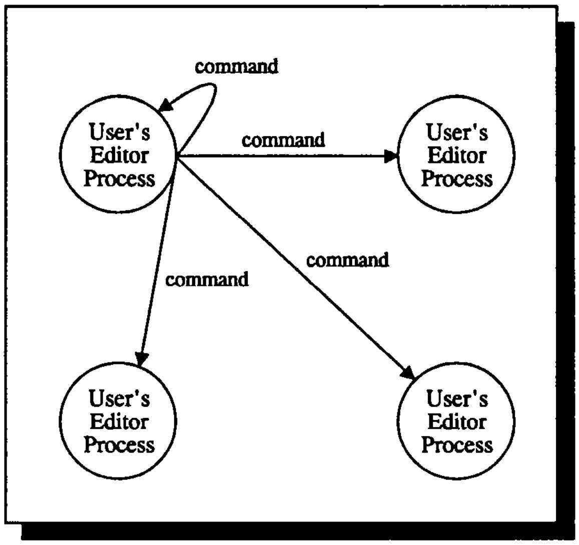

Figure 9-2. DistEdit System Architecture

editors' buffers[1]. Unfortunately, this also raises the possibility that, for example, if one user issues a command to save the buffer to a file, then a file with the same name on another user's machine may get overwritten. DistEdit avoids this problem by not broadcasting file I/O commands, but, in general, system designers must identify and manually close any security loopholes. This task could be arduous and error-prone.

Instead, we would like to each editor be able to flexibly limit its access rights given the purpose of the collaboration. When a user enters a collaborative editing session, the editor first authenticates the other collaborators. In this description, we assume that all the other collaborators are trusted. Then the user sets access rights for the collaborators to read the user's files. When the user decides to edit a file, he specifies the access rights for the collaborators to the file. For example, suppose a DistEdit Emacs editor is being used to collaboratively edit a Lisp program. A reader allows others to edit the Lisp program, but the reader may also want to edit documentation files that he does not want the other users to overwrite. Also, some editors, such as Emacs, permit applications to be executed, so the user may grant access to execute other applications. For example, a collaborator may want to demonstrate the execution of the Lisp program. The user would need to grant permission to execute a Lisp interpreter on his machine for this part of collaboration to proceed.

Therefore, the reader's security requirements in this application are as follows:

- Any collaborator can perform the read operation on any of the reader's public objects, except system-specific files like the password file.
- Any user can perform the write operation on the Lisp program file.
- Access to perform the write operation on any other system objects, such as the documentation files, is prohibited.
- Any collaborator can perform the execute operation on the Lisp interpreter.
- Access to perform the execute operation on other system objects, such as /bin/sh, is prohibited.
- The Lisp interpreter must have the same access rights as the editor.

The user described above wants to limit access rights based on the application, the identities of the collaborators, and the purpose of the collaboration. The level of trust in the collaborators determines the types of actions they may be permitted. For example, only trusted collaborators should be permitted to execute applications on another user's machine. Also, the purpose of the collaboration further specifies the access rights of the collaboration. A collaborative editor can be used

[1] The commands are ordered as well, but this task is outside the scope of the access control problem.

to edit a variety of files, but only access to files necessary to the collaboration should be granted to enforce least privilege.

Also, users need to be able to flexibly control access rights at runtime. The identities of the collaborators and the purpose of the collaboration are often not known until runtime, so a user must be able to set access rights on demand. Collaborations are often dynamic, so new collaboration groups and purposes may evolve. Therefore, access rights for a collaboration may be ad hoc, so users must be able to flexibly specify those rights. In our example, write access to the Lisp program, but not the documentation files, is granted. Future collaborations may involve collaborators that are trusted to edit the documentation files, and new files may enter into the collaboration as well, such as technical papers or marketing reports.

3.0 Related Work

We review the access control capabilities of current mobile agent systems and RBAC models. Access control in mobile agent systems is restrictive because the access rights of other applications cannot be controlled. RBAC models provide system-wide access control, but are designed for MAC not DAC. Therefore, it is not possible for users or their applications to limit the access rights of their processes.

Historically, access control of mobile agent systems has been implemented using the following techniques: preventing processes from accessing system objects [BORE92], limiting access to read and write operations in a single public directory [BORE94], using a global properties file to describe permissible access rights [SUN95], and providing an intersection of the access rights of the collaboration's principals [JAEG94].

The restrictive security provided by the first two options, implemented in ATOMICMAIL [BORE92] and Safe-Tcl [BORE94], prevents access to shared data at its normal location and prevents the execution of applications. Moving data to a safe location is time-consuming and can lead to inconsistencies between versions if a crash occurs. The execution of existing applications is also necessary for some applications. For example, we want to use an existing editor in our collaborative editing application.

In the third option, Java [SUN95] enables agents to read and write files in their normal locations, but the execution of existing applications is still not permitted. In Java, each user defines a properties file that describes the access rights of any agent. Therefore, the same rights apply to any agents being run by the user at the same time, so it is not possible to run two agents simultaneously with different rights.

In [JAEG94], we describe a service that enables a reader to limit the agent's access rights to the intersection of the reader and writer's rights. This service enables trusted users to collaborate while protecting their private system objects from access. The service is implemented using the UNIX operating system[2], so the access rights of agents whose writers do not belong to the reader's domain are more difficult to control. In this implementation they are given no access rights, but if the reader wants to grant some access rights to an agent to execute an application it is difficult to limit this process's access. Current file systems grant access to all world-readable or world-executable files to any process, so permitting execution of arbitrary applications while preventing access to sensitive world-readable files, like the password file, is not possible. Typically, access rights for a UNIX process are restricted using chroot, but chroot requires files to be moved to the safe area at runtime.

Current implementations of access control models based on RBAC also are insufficient for collaborative applications. RBAC implementations for file systems [GASS90, TING92, VINT88, WOBB94] are designed to provide MAC. Therefore, only system administrators can define the roles that a user can assume. As described above, access rights of collaborative applications may not be known until runtime, so DAC models are necessary to enable a user or application to limit access rights.

A few RBAC models do enable some dynamic modification of a role's access rights [BORN94, MOHA94]. These models permit system administrators to specify rules that can modify the access rights of a process at runtime. Because the access requirements of many collaborative applications are ad hoc, it is unlikely that the rules necessary to control access for these applications have been specified. Also, the purpose of an application cannot be deduced very easily, so the specification of rules for collaborative applications is difficult if not impossible.

Another important issue in RBAC model implementations is the power of the specification language. In some RBAC model implementations, when a new role is added the access control lists of all the affected system objects must be updated. Unlike other RBAC models, the Domain Type Enforcement (DTE) [BADG95] model provides a concise language for specifying process access rights. For example, read access to all the system objects in a directory tree can be specified in a single statement.

Currently, DTE is designed for MAC, so users cannot dynamically modify their rights. Even if DTE permitted users to limit their own access rights, specification of these new rights is cumbersome. Suppose the reader wants to limit the access rights of a process such that only the rights shared by the reader and writer are available to the process. First, the reader must create a new domain (i.e., role) to represent the new set of access rights. Next, the reader must specify the access rights for this domain given the reader's and writer's access rights. Because general directory structures are graphs (due to the existence of links), the reader may have to check whether the writer has access to every file to which he, the reader, has access. However, the number of links should be significantly less than the number of files, so handling these

2 UNIX is a registered trademark of the Unix Open Foundation, Inc.

specially should reduce the complexity significantly. Finally, the current domain must be permitted the ability to create an instance of the new domain. Thus, the mechanism to create a new domain may be inefficient for large file systems, and it requires that users understand the relationships between domains in order to create a new one.

4.0 Our Approach

In [JAEG95], we define a DAC model for specifying the access rights available to a mobile agent. The goal of the DAC model is to enable the reader and writer in a mobile agent computation to flexibly control access to system objects. Therefore, using this DAC model a user can specify any access rights desired, although the model is biased toward the easy specification of the access rights that we think will be common.

Below, we define the major concepts of this DAC model:

- **Definition 1:** A *writer*, *w*, is a principal that forwards the mobile agent to the reader for execution. Note that the writer does not necessarily compose the agent.

- **Definition 2:** A *reader*, *r*, is a principal that executes the mobile agent.

- **Definition 3:** An *object access right* of a mobile agent, $oba \in OBA$, is a tuple, $ob = (obj, OP_{obj})$, where *obj* is a unique identifier of the object and OP_{obj} is a set of operations (e.g., read, write, execute) that the mobile agent can perform on the object. Object access rights can be either granted or revoked.

- **Definition 4:** A *sharing function* of a principal *i* for a mobile agent, *sf(i)*, is a function, $sf(i) = s_i$ or $sf(i) = s_i \cap j$, where s_i is a sharing value (one of none, public, or all) that specifies a class of objects accessible to *i* and *j* is another principal. For example, if the writer specifies $sf(writer) = \text{all} \cap reader$, the writer grants permission to all of the writer's objects shared by the reader. This is the set of objects shared by the writer in ② of Figure 9-3, *Operation Access Rights*.

- **Definition 5:** An *operation access right* of a mobile agent, $opa \in OPA$, is a tuple, $opa = (op, sf(r), sf(w))$, where *op* is an operation, *sf(r)* is the sharing function of the reader for *op*, and *sf(w)* is the sharing function of the writer for *op*. The value of *opa* specifies that the agent has permission to perform *op* on the union of the objects represented by the domains *sf(r)* and *sf(w)*. (See Figure 9-3.) If an value is not specified for an operation *op*, then $opa = (op, none, none)$.

- **Definition 6:** A *mobile agent computation* is a set of processes that execute a mobile agent. In our case, a computation is a set of processes, $p \in P$, including the process that executes the agent, its descendant processes, and any service processes that these processes use.

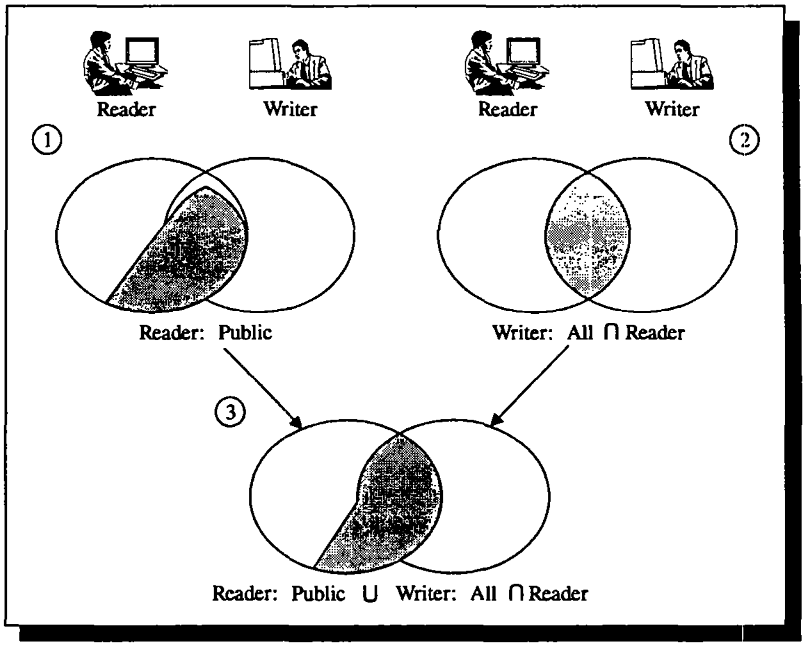

Figure 9-3. Operation Access Rights

- **Definition 7:** *Mobile agent access rights*, *ar*, is a tuple, $ar = (r, w, OPA, OBA_g, OBA_n)$, where *r* is the identity of the reader that executes the agent, *w* is the identity of the writer of the agent, *OPA* is a set of operation access right specifications, OBA_g is a set of object access rights granted to the agent, and OBA_n is a set of negative object access rights of the agent. The order of precedence of the access rights specifications is (from highest to lowest): OBA_n, OBA_g, and *OPA*. The mobile agent access rights must be enforced on all processes in the mobile agent computation.

In this DAC model, the access rights of a mobile agent to the file system are specified by operation and by object. Operation access rights permit the reader and the writer to limit the operations that can be performed on a class of objects. For example, the read operation can be limited to only the writer's public objects. Object access rights permit the reader and writer to grant or revoke operations on a specific object. Read access may be precluded for the password file, but write access may be granted to a private object, such as the Lisp program.

In the DistEdit example, the writer does not provide any access rights to the reader, but the reader needs to limit the access rights of the agent. We specify the mobile agent access rights for this example as shown in Table 9-1, *Mobile Agent Access Rights for DistEdit*. When a file system access is requested, the access rights specifications are checked in the following order: (1) object access revoked; (2) object access granted; and (3) operation access rights. First, the object access revoked specification is checked to determine if access to the object is prohibited. The specification (/etc,read,write,execute) prohibits access to any file in the system's /etc directory, such as the password file. Object access rights revoked always supersede those granted if there is a conflict. Next, the object access granted specifications are checked to determine if access has been granted directly to the agent. The specification, (/usr/bin/lisp,execute) and (~ /lisp/program.lisp,read,write), grants access to the Lisp interpreter and the Lisp program, respectively. Note that the reader must already possess these access rights in order to grant them. This can be verified at specification time. Finally, the operation access rights are checked. In this case, the specification (*read,public,none*) grants read access to all the reader's public objects. Because operation access rights are not provided for other operations, these operations are precluded on all other objects.

Table 9-1. Mobile Agent Access Rights for DistEdit

***ar* Attribute**	**Value**
Reader	DistEdit_reader
Writer	one of collaborators
OPA	{(read,public,none)}
OBA_g	{(/usr/bin/lisp,{execute}), (~ /list/program.lisp,{read,write})}
OBA_n	{(/etc,{read,write,execute})}

This DAC model permits principals to limit the access rights of a process and its descendants relative to its current access rights, so the effort necessary to specify the restricted access rights is reduced. Operation access rights define an intersection of the principal's current access rights with the rights to a class of objects. Therefore, it is not necessary for the user to specify the set of objects in the class explicitly.

Using this model, principals do not need to be aware of domains and their relationships, but rather, the principal must be aware of some general types of objects. Only a few general types are necessary, so the model is fairly simple.

Finally, the performance of an authorization mechanism that uses this model should be satisfactory. Using other implementations, object access rights can be checked in constant time (e.g., using a hash table representation), and two authorizations must be made for operation access rights: one for the principal's current role and one for the object group of the operation access rights. If both are authorized, then the access is permitted.

5.0 RBAC Requirements

In [SAND94b], a set of dimensions for RBAC model requirements are proposed. Below, we specify values for these dimensions to implement the DAC model described above. We list values only for the dimensions relevant to the DAC model implementation.

- **Nature of privileges:** Access rights are represented by roles, operation access rights, and object access rights. A role represents the MAC domain of a principal. The operation access rights can be used to dynamically limit a process's access rights given the current role. The object access rights specify positive and negative capabilities, so the access rights of a mobile agent computation can be flexibly specified.

- **User assignment:** System administrators assign users to roles. System administrators select which users can limit their access rights using operation or object access rights.
- **Privilege assignment:** The system administrators set the privileges of roles and define the object groups for operation access rights. Users select operation access rights and objects access rights for a mobile agent computation.
- **Object attributes:** Operation access rights or object access rights may be defined using object groups.

In this type of RBAC model, system administrators define roles for users as they did previously, but also can define a model that will enable users to limit the access rights of their own processes. First, system administrators specify the types of object groups that can be used to specify operation access rights. Also, system administrators specify the principals that are authorized to limit their own rights dynamically. Then, authorized users may flexibly control access to their system objects using these operation access rights and object access rights. Therefore, system administrators enforce MAC using the RBAC as well as specifying some DAC features and who can use them. Users with the privilege to use the DAC model can flexibly control access to their processes.

6.0 Conclusions

The access control requirements of collaborative systems place interesting demands on RBAC models. Because commands can originate from multiple principals, each principal that owns a collaborative system process must be able to restrict the process's access rights to protect its system from unauthorized access. These restrictions are often ad hoc because the choice of access rights is based on the identities of the collaborators and the purpose of the collaboration. Current RBAC model implementations do not permit users to define new roles, so it is not possible for users to dynamically control access to their system objects. Also, the specification models used by RBAC model implementations require significant effort for a user to create a new role and require the users to understand complex concepts.

We propose requirements of an RBAC system for implementing a DAC model that enables users and their applications to flexibly control the access of collaborative processes at runtime. The DAC model enables users to define the access rights of a process relative to their current role. Access rights are specified either by operation on a commonly understood group of objects or by a specific object. This DAC model enables access rights to be specified using a small number of statements and only requires that users understand the meaning of the object groups used by the operation access rights. To implement this model within an RBAC framework, RBAC systems need to provide a DAC model using simple primitives, permit some users to define more limited versions of their roles, and be able to generate roles efficiently using these specifications.

In the future, we plan to extend the DAC model to enforce the communication security requirements of collaborative systems. Collaborating principals must be able to communicate with one another, but unlimited communication is not possible because there are security loopholes that can result in an agent sending a reader secrets to an attacker [DEAN95].

Acknowledgments

The authors would like to thank the workshop participants for their valuable comments regarding this work.

References

[BADG95] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical Domain and Type Enforcement for UNIX," *IEEE Symposium on Security and Privacy*, Oakland, CA, 1995, 66-77.

[BORE92] N. S. Borenstein, "Computational Mail as a Network Infrastructure for Computer-supported Cooperative Work," *Proceedings ACM 1992 Conference on Computer Supported Cooperative Work (CSCW)*, Toronto, Ontario, Canada, 1992, 67-74.

[BORE94] N. S. Borenstein, "Email with a Mind of its Own: The Safe-tcl Language for Enabled Mail," *ULPAA '94*, Barcelona, Spain, 1994, 389-402. Available from: ftp://ics.uci/edu/safe-tcl/safe-tcl.tar.Z.

[BORN94] E. Born and H. Stiegler, "Discretionary Access Control by Means of Usage Conditions, *Computers and Security*, 13:5, 1994, 437-450.

[DEAN95] D. Dean and D. Wallach, *A Security Analysis of the HotJava Web Browser*, 1995. Available from: http://www.cs.princeton.edu/sip/java/.

[ELLI91] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM*, January 1991, 38-51.

[GASS90] M. Gasser and E. McDermott, "An Architecture for Practical Delegation in a Distributed System," *IEEE Symposium on Security and Privacy*, 1990, 20-30.

[GOSL95] J. Gosling and H. McGilton, *The Java Language Environment: A White Paper*, 1995. Available from: http://java.sun.com/whitePaper/java-whitepaper-1.html.

[JAEG94] T. Jaeger and A. Prakash, "Support for the File System Security Requirements of Computational E-mail Systems," *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, VA, 2-4 November 1994, 1-9. Available from: ftp://ftp.eecs.umich.edu/people/aprakash/collaboration/papers/cccs94.ps.Z.

[JAEG95] T. Jaeger and A. Prakash, "Implementation of a Discretionary Access Control Model for Script-Based Systems," *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Dronquinne Manst, Kenmare, County Kerry, Ireland, 13-15 June 70-84. Available from: ftp://ftp.eecs.umich.edu/people/aprakash/collaboration/papers/csfw95.ps.Z.

[KNIS90] M. Knister and A. Prakash, "DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors," *Proceedings of the Third ACM Conference on Computer-Supported Cooperative Work*, Los Angeles, CA, 7-10 October 1990, 343-355.

[KNIS93] M. Knister and A. Prakash, "Issues in the Design of a Toolkit for Supporting Multiple Group Editors," *Computing Systems—The Journal of the Usenix Association*, 6:2, 1993, 135-166.

[MOHA94] Imtiaz Mohammed and David M. Ditts, "Design for Dynamic User Role-Based Security," *Computers and Security*, 13:8, 1994, 661-671.

[SAND94b] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control: a Multi-dimensional View," *Proceedings of the Tenth Computer Security Applications Conference*, Orlando, FL, 5-9 December 1994, 54-62.

[SUN95] Sun Microsystems, *Frequently Asked Questions: Applet Security*, Version 1.0 Beta 2, 1995. Available from: http://java.sun.com/sfaq.

[TING92] T. C. Ting, S. A. Demurjian, and M. Y. Hu, "Requirements, Capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model," *Database Security V: Status and Prospects,* C. Landwehr and S. Jajodia (Eds.), North-Holland, 1992, 275-296.

[VINT88] S. T. Vinter, "Extended Discretionary Access Controls," *IEEE Symposium on Security and Privacy*, Okland, CA, 18-21 April 1988, 39-49.

[WOBB94] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the Taos Operating System, *ACM Transactions on Computer Systems*, 12:1, February 1994, 3-32.