VOTING ENABLED ROLE-BASED ACCESS CONTROL MODEL FOR DISTRIBUTED COLLABORATION

By

VIJAY MANIAN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2005

I dedicate this work to my parents, S.V.S. and Padma Manian.

ACKNOWLEDGMENTS

I would like to thank Dr. Richard Newman for getting me started on this project. I would like to express my gratitude to Dr.Steve Thebaut for his willingness to help me whenever I had a problem. I would also like to thank all the DCS group members, past and present, for their invaluable contributions without which I would not have been able to complete this dissertation.

Special thanks are due to Salil Gokhale for helping me stay on track during the last stages of my Ph.D. Finally, I would like to thank my parents for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

VOTING ENABLED ROLE-BASED ACCESS CONTROL MODEL FOR
DISTRIBUTED COLLABORATION

By

Vijay Manian

December 2005

Chair: Richard E. Newman
Major Department: Computer and Informational Sciences and Engineering

This dissertation presents a voting enabled role-based access control (RBAC) model designed for distributed collaboration that allows the group to decide and implement group policies. This model can also be used in many middle-level systems that require support for group voting. The primary goals of this model are security, scalability, flexibility and simplicity. One of the major drawbacks with many models in use today is the need for an external administrator or a user with "god-level" powers to implement the group policies. We overcome this by allowing subsets of users to vote on access control decisions that are deemed important. This takes the burden of group management from the system administrators and allows the group to be truly democratic and autonomous. In this dissertation, we give the motivation for the model, formally define it and give an example of how it can be used. We also define the access right leakage problem which we use to analyze the safety of the model. We first prove that for a simple version without any voting, the access right leakage problem can be decided in polynomial time.

We then introduce the notion of *trust-level* and discuss how the trust-level of the users affect the safety of the system. This dissertation identifies three simple

approaches by which a malicious entity can cause group members to vote in a manner that is not in the best interest of the group. We then analyze the safety of the model under each of these three approaches.

CHAPTER 1
INTRODUCTION

Distributed collaboration is more than just a buzzword in today's world. Most users log in from large networks, and often times, the collaborators are from different administrative domains. In such a scenario, access control can become an additional burden on the already overworked administrators and usually is difficult without enforcing many restrictions. On a stand-alone machine, access control is more or less straightforward with the operating system providing simple discretionary access control (DAC) mechanisms like access control lists (ACL) or capability lists (CL). Most of these mechanisms are based on the access control matrix (ACM) model [1, 2]. Various models have been proposed for multi-user systems and these models are defined with well-known abstractions like subjects, objects and access rights.

There are some major disadvantages to the current models when applied to distributed environments in general and distributed collaborative systems in particular. Most systems today are run over the network and have a centralized node to handle access control, which results in a single point of failure and more importantly, a single administrative authority, which is not always desirable. Many practical models restrict the types of accesses (for example, Unix has only three types: read, write and execute), which does not reflect real life. In addition, each object in the system can have only one owner, which might not be the case in a collaborative environment. Some of the major problems with classical models are discussed in section 1.1.

The Distributed Conferencing System-Access Control Model (DCS-AC) discussed in this dissertation has been designed to provide a simple, powerful,

implementable, scalable and flexible access control mechanism. One of the most important features of this model is the use of group decision-making mechanisms (voting) as a part of the access control decision process. Here, important access control decisions can be referred to a vote among a specified subset of the users, thereby allowing them to self-administer the groups. Many current models rely on a "superman" who can do almost anything in the system. The DCS approach models society more closely and also protects the system against malicious single-user attacks.

We analyze the safety of the model in terms of the Access Rights Leakage (ARL) problem. ARL is the problem of deciding whether a malicious entity can execute a set of commands which result in an unauthorized user gaining access to an object in a given DCS-AC instance. We approach this problem by first solving it for a system with no voting. When the subjects are allowed to vote on one or more access control decisions, it is difficult to decide on a yes or no answer. We assign a trust-level value to each subject and decide on the answer in terms of the trust-levels of all the subjects who can participate in the vote(s). Let us now consider why we need the DCS-AC model.

1.1 Motivation

Most access control systems in use today are based on the traditional access control matrix model.[1] This proposal considers access control policies in a distributed collaborative environment where the traditional models suffer from the following problems (a few more problems with current systems are given in Greenwald [3]).

[1] ACL and CL can be considered to be a part of the traditional access control matrix model

1. System administrators are usually overworked and underpaid. A centralized server that coordinates the entire distributed system will not work well.

2. If three subjects are collaborating on a document, under the current paradigm, any one of these three can unilaterally delete the object. This is not desirable and a more useful policy could be to require at least two of the author's approval in order to delete the object.

3. It is difficult to obtain access to objects located in a different administrative domain without getting the local system administrator's permission.

4. Even in systems that are specifically designed for distributed computing [4], like grid computing, the standard policy is for each domain to be responsible for the objects in its site [5]. Now, if Alice accesses an object from another administrative domain, a copy of the object is usually cached. The local system has no idea about the access control policies of this object and when Bob requests access to the same object, the request is forwarded to the other domain. If the local system were aware of the access control policies for the object, it could have made the decision on its own and saved Bob a lot of time.

5. The individual groups of collaborators cannot set their own group policies.

The DCS-AC model addresses these issues and provides a very flexible access control mechanism. This model was developed for Distributed Conferencing System (DCS) [6], a distributed collaborative architecture that allows users from different sites to work together. The first objective of the DCS-AC model is to provide flexible, implementable and fault tolerant access control. The second objective is to allow the users of the system to self-administer the access privileges, i.e., allow a subset of users (who are often the stakeholders) to make access control decisions for important objects on a case-by-case basis. This is done by using a set of decision templates each of which defines the subset of users that can participate in a vote.

Each entry in the ACM has a decision template pointer which has to be executed in order to arrive at a decision. Although the model enforces no constraints on which decisions require a vote, we expect that most entries will point to the *Always-yes* template which always returns a *yes* since many rights, once granted,

need not be referred to the voting group for every access. However, for important access control decisions the system can refer the decision to a subset of the users who vote on it. This allows the users to express access control policies for some accesses (like joining the group, accessing important objects, granting access rights, etc.) in the system.

The use of decision pointers also improves fault tolerance. It protects against single malicious user threats, since many users may have to agree for the vote to pass. In addition, if some of the users are unavailable, the system can still take decisions based on the users available (unless the system policy specifically states that a particular user must agree for the vote to pass). There are many real-life scenarios in which such multi-user authorizations are required and are often handled off-line before one of the participants "informs" the system if the request is approved.

1.2 Sample Scenario

Consider the following example of a project to be implemented by a team consisting of a *project leader* (PL) and many subordinates (*architect*s, *programmer*s and *tester*s). The *architect*s work on the design and once the design is approved by the PL, one or more *programmers* code the project, which is then tested by the *tester*s. The testers can either send it back to the programmers along with a bug-list or notify the PL that the code is acceptable. The PL then reviews the code with the review team, and if approved, the project is released to the client (refer to Figure 1–1).

Given below are a few company policies applicable.

1. The individuals concerned may be located in different branches; confidentiality and availability are important for the shared objects.

2. The architect does not need access to the code and the tester can look at the code only after the programmers are ready.

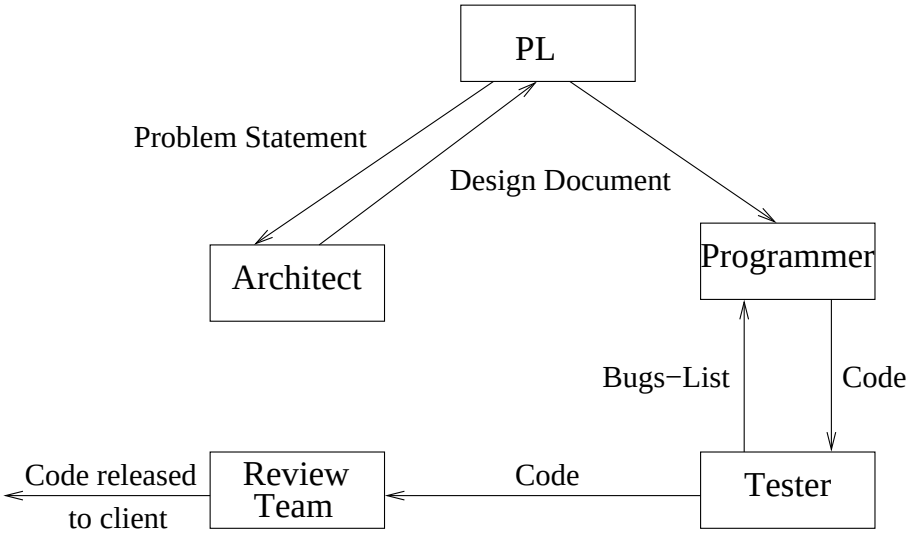

Figure 1–1: Class Library Scenario

3. The review team consists of all PLs in the company and the members do not have access to the code until it passes the testing phase.

4. The PL can access any document related to this particular project at any point of time.

5. The PL also decides who works on this project.

6. The PL cannot change the role of an employee; for example, the PL cannot assign a programmer to do the task of an architect.

7. There are many architects, programmers, etc. in the company, and only those involved in development should have access to the objects.

8. The programmers do an internal review and send the code to the testers after every programmer has approved it.

9. The review team members vote on whether the code can be added to the library.

The first eight requirements can be handled by RBAC in a straightforward manner using constraints. However, specifying constraints introduces runtime overhead [7]. Requirements eight and nine involve group decisions. Although the above example deals with a simplistic scenario, we believe that there are many examples

that involve similar issues. The DCS model is designed to handle scenarios of this nature.

1.3 Organization of this Dissertation

Chapter 2 gives a brief overview of the Distributed Conferencing System and chapter 3 provides the technical background for the DCS access control model. Chapter 4 deals with the formal definition of the DCS Model and discusses the features of the DCS model. Chapter 5 discusses the decidability of the ARL problem for the DCS-AC model without voting and chapter 6 discusses trust in voters and the trust worthiness of a DCS-AC system where voting is allowed. chapter 7 mentions some future directions of research and chapter 8 concludes this work.

CHAPTER 2
DCS OVERVIEW

2.1 Introduction

In today's highly-networked working environment, support for collaboration and group-work has become critical. The availability of low-cost Wide Area Networks (WANs) means that members of a group can be geographically scattered around the country (or even the world). There is a growing need for efficient architectures and applications that facilitate collaboration between such members. A system that allows users to log on from different locations, to share objects, and to work on the same object with other members of the group has become essential in such cases.

The Distributed Conferencing System (DCS) seeks to fulfill these requirements and many more. The system is based on a robust architecture and provides distributed file services, access control, secure communication, notification and group decision-making mechanisms. The DCS will provide basic distributed collaborative applications like text editors, graphics editors, etc., and can also support third-party applications. The scalable architecture can support multiple sites and objects can be replicated to ensure faster access and availability. One of the major design criteria is fault tolerance. The system is designed to handle network failures and system crashes by restoring from backups and contacting other sites for the most up-to-date information. We achieve this by replicating the objects across multiple sites and also by means of backup servers for each site which can take over if the primary server fails.

2.2 Definitions

Before getting into the details of DCS and the various services provided, we first define some terms that are required to understand the architecture:

- DCS instance This is a collective term for one or more DCS sites that share resources, communicate DCS specific information with and store information about each other. Any user id or CoG registered at one site is accessible from all sites in that DCS instance. An instance of DCS may cover multiple administrative domains.

- Site This term refers to the DCS server(s) located in a single administrative domain that provide services to the users. Sites that are a part of the same DCS instance can communicate with each other.

- Collaborative group (CoG) This term refers to a persistent or transient group consisting of members, objects, applications, and a DCS-AC system (consisting of roles, object types, allowable member-role bindings and decision templates). The members of a CoG work together on the objects and decide group policies for that CoG.

- User This is an entity in the system that initiates action and (in the context of access control) requests to access objects. This term usually refers to a human being.

- Member This term when used in the context of a CoG refers to a user who is a part of the CoG and has been granted rights that are not granted to non-members.

- Subject This represents an active user process that is allowed to perform some action within the system on behalf of the user.

- Object This is a passive entity in the system. Users can perform various accesses on them. Examples of objects are documents, files and DCS system specific tables.

- Application This is a program that the user launches from within the system to work on the objects.

- Collaboration-ware These are applications that are designed to support collaborative work. Therefore these applications will have a better access control features than the regular off-the-shelf applications.

- DCS-aware applications These are applications that are specifically designed to work within the DCS. These applications can use the features of DCS like event notification or new access type registration.

- Role This is a named group of subjects. Roles can also be viewed as a collection of job functions. A particular subject can bind to any number of roles but at any point of time, the subject can be bound to only one role. It must be noted that *role* as used here is slightly different from *role* as used in Role-based access control because there is no notion of a role-hierarchy in the DCS model.

- Object type This is a named group of objects. A particular object can belong to only one object type.

2.3 A Typical Day at Work

A typical instance of DCS consists of many sites and each site consists of a primary server with one or more backup servers (which are ranked from 1 to n where n is the number of backups) to ensure high availability. The primary server sends periodic *heartbeats* and updates to the backups [8]. If the backups do not get the heartbeat from the primary for a fixed amount of time, the lowest ranked backup takes over as the primary and informs the other sites in the instance. Similarly, the sites exchange *heartbeats* and if a site does not respond, the other sites can take over the management of objects controlled by the failed site.

Each site contributes resources (like disk space) to the instance and all objects are partially replicated to provide fault tolerance. The DCS architecture is divided in to various modules each of which provides a different service to the users (section 2.4). A user logs into a CoG through a site and is bound to a role (based on the user-role bindings maintained by the Access Control Services). The Distributed File Services provides a uniform view of the *DCS space*, which actually consists of objects located on different sites. The user can launch applications to access these objects, participate in decision making processes or interact with other members of the CoG. It must be noted that site are managed in the same way as CoGs. That is, On each site, there is a CoG corresponding to the site containing all subjects

who joined the DCS instance from that site; they can form and implement their own site management rules. The following section provides a brief description of the various services and tools available in DCS.

2.4 Services

The architecture of DCS v.2 is shown in figure 2–1. It is based on various services available in most networked domains (like local file systems, TCP/IP, cryptographic packages, database system, etc.). Since we are using Java [9, 10], the sites need not all run the same operating system. A brief description of the core services is given below.

2.4.1 CoG Manager (CM)

This is the module responsible for managing the collaborative groups (CoGs) [11]. This module instantiates the other modules and the clients interact with the other services mainly through the CM. Tasks like merging two instances of DCS or sites or CoGs, and splitting an instance, a site, or a CoG is handled by this module. The CM interacts with the secure messaging and the cryptographic layers during user login. The user's access control requests are routed through the CM. The CM also allows the user to import and export objects, add new members and start sub-groups (sub-CoGs).

2.4.2 Notification Services (NTF)

The NTF [12] provides asynchronous event notification to registered users. Users can define new event types and register to be notified on events. Subjects are automatically registered for notification if they are among the valid voters on a decision. The various services and applications running in the instance raise events as and when they occur and the NTF maintains a global and local database in which it stores the users to be notified on each event along with the delivery method.

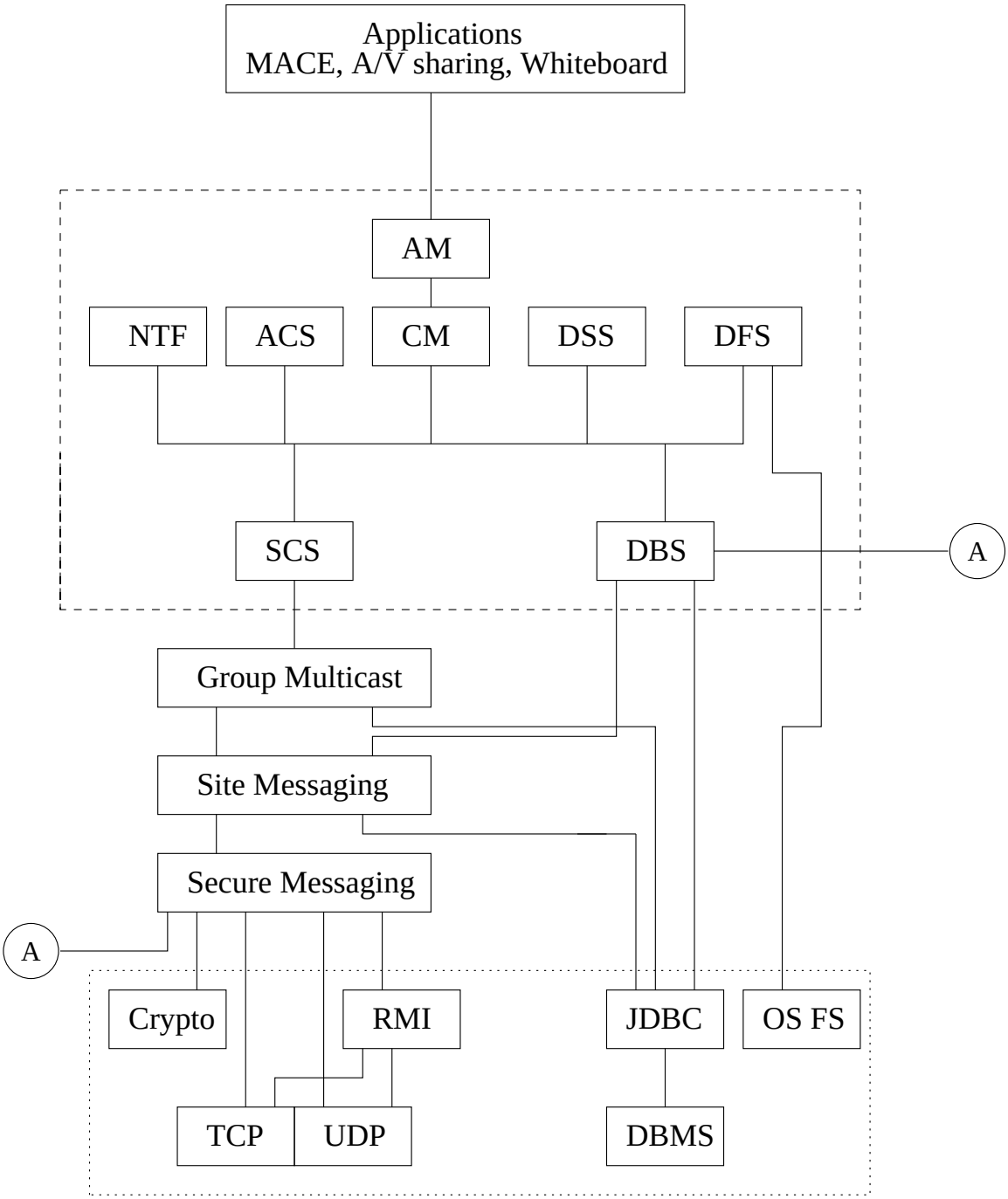

Figure 2–1: Architecture of DCS

2.4.3 Database Services (DBS)

The DBS [13] makes use of a backend Database Management System (DBMS) [14] to maintain the tables of all the DCS services and applications. Most tables are stored as partially replicated distributed databases. The DBS uses group multicast and ensures eventual consistency among all member sites by using vector timestamps. Since DBS uses Java Database Connectivity (JDBC) [15], we can use any backend DBMS system like MySQL [16] or PostgreSQL [17].

2.4.4 Decision Support Service (DSS)

Decision Support System [18] is a module that facilitates resolving issues by a group of users who have a joint responsibility to such decisions. Voting is one of the tools of DSS which is designed to be a flexible and friendly system that allows users to initiate, participate in and terminate decisions within the context of a CoG. It provides on-line voting and has automated vote collection mechanisms. It allows members who are currently off-line to be voters in a decision process. The initiator of the vote can set time constraints and specify voting parameters, voters, reporting styles and report recipients.

This service takes into account the user's role, weight assigned to the vote and access rights, etc. It has several features like reminders, request for status, change of vote, short circuit evaluation, withdrawal and termination of active votes, which add more flexibility to the decision process. It also allows other modules of DCS to make these requests.

The module supports different voting methods like majority, plurality, and ranking. Users can save their requirements as templates that can be reused in the future by anyone who has the access rights to do so.

2.4.5 Access Control Services (ACS)

The ACS [19], as the name suggests, controls access to (CoG-specific or instance-specific) objects. It makes use of decision templates of the DSS to provide

group decision-based access control. This puts the control of the CoG in the hands of the members. The ACS stores its tables in a database through the DBS. The ACS allows applications to register new access types. This allows more fine-grain access control if the application supports it.

The ACS interacts with all the other five modules and provides various services to the users. The DCS architecture requires the ACS to allow decisions to be made on access control requests at the source site itself. This coupled with the decision templates makes the ACS an interesting study in itself. The model used here is not specific to DCS and can be used in any distributed system that supports group decision-making mechanisms.

2.4.6 Secure Communications Services (SCS)

The Secure Communication Services [20] ensures the confidentiality, authenticity and integrity of all messages passed between sites and clients. To enforce these important security properties, it is necessary to establish the identity of the entities of the system. The entities of the system are users, hosts, and servers. The identification of the entities should be done reliably, that is, it should be authenticated. DCS poses a greater challenge since entities communicate through messages and hence all the entities must establish their identities and maintain confidentiality. In DCS, identity is first established using strong authentication and per-connection symmetric cryptography provides confidentiality and integrity. The SCS is also responsible for creation and maintenance of keys and certificates for sites and users.

2.4.7 Distributed File Services (DFS)

The Distributed File Services [21] provides a DCS-specific view of the objects located at various sites. It allows concurrent access of files using a versioning scheme with immutable shared files and is designed to provide reasonable service in the case of site crashes. Each CoG is represented by a directory in *DCS-space* and sub-CoGs are simply sub-directories under the directory corresponding to the

parent CoG. The DFS creates and maintains multiple copies of all objects, thereby ensuring high availability.

2.4.8 Application Manager (AM)

The Application Manager is responsible for registering and invoking applications that are available for each CoG. These applications could be DCS-aware (like MACE [22]) or DCS-unaware. The AM maintains a list of applications available for a particular CoG, and depending on the access rights of the user, it makes those available to the users.

2.4.9 Tools

The DCS framework provides a few basic tools for collaboration. These include instant messaging (Gossip), concurrent text (MACE) and graphics (Ensemble) editors.

- **Gossip** This is a simple instant messaging application that is a part of the client GUI. It allows the user to be in touch with his/her collaborators.

- **MACE** MACE [22] stands for Mother of All Concurrent Editors and is a very powerful concurrent text editor. In addition to regular text editor features, MACE allows users to lock various parts of the files and share views. MACE provides a very fine-grained (byte-level) locking mechanism and the user can set the view rights for others, i.e., the other users can see every key stroke or receive updates only when the writer commits the changes. This allows other users to "look over the shoulder" of the writer (useful in pair programming among other areas).

- **Ensemble** Ensemble is a concurrent graphics editors that provides object-locking and specific user pointer view. Multiple users can work on the same figure and each user can lock one or more objects and share cursor, allowing other users to see exactly what is going on.

2.5 Interaction With Other Services

Since we are primarily concerned with the ACS, which implements the DCS-AC model, let us look at ACS' interactions with the other services. As mentioned

earlier, ACS interacts with all the core services. Out of these, DBS and DSS provide vital services required for ACS.

2.5.1 Interaction With DBS

All ACS information about a CoG are stored as relational database tables that are replicated across all sites interested in the CoG by DBS. When a new CoG is created, the relevant ACS tables have to be created on the local site. Whenever a site wants to participate in a CoG, it obtains a dump of the ACS tables from a site that is already in the CoG and copies it on to the local database. Any database query can be handled by the DBMS at the local site but updates have to be forwarded to the site that "owns" the entry and then multi-cast to other sites.

2.5.2 Interaction With CM

The CM controls the entire conference and instantiates the objects that provide the services including ACS. Moreover, most user requests are passed on to the ACS through the CM. Most CM-originated actions have to be checked through the ACS for access privileges. For example, if a user wants to import a new file, the CM will have to check with the ACS before proceeding with the import.

2.5.3 Interaction With DSS

The DSS handles the decision-making process and is therefore closely tied to the ACS. If a request requires a vote, the DSS has to be invoked and the voting process initiated. The ACS must be able to check the status of a particular vote. When a vote is completed, the DSS should notify the ACS. Similarly, the ACS must be able to report a list of active votes to the DSS if the DSS decides to delete obsolete votes. Actions like defining new templates and modifying existing ones require a check by the ACS.

2.6 Summary

The core services described here provide a framework for distributed collaboration. The three tools mentioned are what we consider minimal applications

required by collaborating users. The Application Manager allows users to execute other applications from within the DCS framework. As will become clear in the next chapter, DSS is required by the DCS-AC model since it handles all voting related issues.

CHAPTER 3
BACKGROUND

3.1 Introduction

This chapter surveys some of the work in access control and trust in voters that is relevant to the Distributed Conferencing Service - Access Control (DCS-AC) model.

3.2 Access Control

Protection of objects is complex because the number of access points may be large, there may be no central authority through which all access requests pass, and the kind of access is not simply limited to read, write, or execute. According to Pfleeger [23], there are several complementary goals in such a mechanism.

- *Check every access.* We may want to revoke a user's privilege to access an object. If we have previously authorized the user to access the object, we do not necessarily mean the user should retain indefinite access to the object.

- *Allow least privilege.* A subject should have access to the smallest number of objects necessary to perform some task. Not allowing unnecessary access to objects guards against security weaknesses if a part of the protection mechanism should fail.

- *Verify acceptable usage.* The final decision on an access request is a yes-no decision. Of more interest is checking that the activity to be performed on an object is appropriate.

Access control policies are broadly classified as Mandatory Access Control (MAC) and Discretionary Access Control (DAC). MAC means that access control policy decisions are made beyond the control of the individual owner of an object. A central authority determines what information is to be accessible by whom, and the user cannot change access rights [23]. A classic example of MAC is in military

security where the owner of a "top-secret" object cannot decide who has top-secret clearance nor can the owner change the classification of the object to "secret". Examples of models that enforce MAC include Bell-La Padula Confidentiality model [24] and Biba Integrity Model [25]. DAC on the other hand, allows the owner of an object or anyone else who is authorized to control access to the object to specify who has what access to the object. The access rights in a DAC scheme can change dynamically [23]. One of the main differences between DAC and MAC is the fact that MAC is also concerned with the flow of information [26].

In this work, we are mainly concerned with DAC since we want the users to be able to decide their own access control policies without any central authority. In the following sections we look at various protection mechanisms. First we will consider the Access Matrix model. We also describe certain modifications like Access Control Lists and Capability lists along with the Graham-Denning and Harrison-Ruzzo-Ullman models, which are all DAC models. Next we will review role based access control, and the Distributed Compartment Model for resource management and access control proposed by Steven Greenwald. Finally, we will look at recent work on access control for collaborative environments.

3.3 Access Matrix Models

Access control matrices are the most popular means of modeling an access control system. An access control system has a group of users, objects and access rights for those users on those objects. The HRU [27] and Graham-Denning [2] models are early examples of such models. Such systems have a set of commands that test the presence of various rights in the matrix and then execute a sequence of operations. Most modern systems use a variation of the access control matrix like access control lists or capability lists. Access rights leakage is an interesting property of access control systems. One of the many means of analyzing a model is to try to classify the leakage problem for that model.

Table 3–1: Primitive Operations in HRU model

enter r into (X_s, X_o)
delete r from (X_s, X_o)
create subject X_s
create object X_o
destroy subject X_s
destroy object X_o

3.3.1 HRU Model

Harrison, Ruzzo and Ullman proposed a model in 1976 that is popularly referred to as the HRU model [27]. A HRU *protection system* consists of a static, finite set of rights R, and a static, finite set of commands, C. There are six primitive operations defined which are given in Figure 3–1. A *configuration* of a protection system is a triple (S, O, P), where S is the set of subjects, O is the set of objects, $S \subseteq O$, and P is a rights matrix, with a row for each subject and a column for each object. $P[s, o]$ $(\subseteq R)$ gives the rights to the object o possessed by the subject s. The formal definition of the model is as follows:

Definition: A *protection system* consists of the following parts:

1. a finite set of *rights*, R,

2. a finite set C of *commands* of the form:
   command $\alpha(X_1, X_2, \ldots, X_k)$
   if $(r_1 \in (X_{s1}, X_{o1})) \wedge \ldots \wedge (r_m \in (X_{sm}, X_{om}))$ **then**
   op_1
   $\ldots$
   op_n
   end

Here each subject or object X_{si} or X_{oi} is one of the parameters X_j and each operation op_i is one of the *primitive operations* given in Table 3–1.

Access right Leak

While we cannot expect a useful system to be safe in the strictest sense, the minimum tolerable situation is that the user should be able to decide whether a

given configuration could lead to some other user getting a specific right, i.e., if the right *leaks* to other users [27]. Needless to say, there are some users whom we trust and in fact expect to have the right. We remove the rows corresponding to those users and try to find out if the right leaks when some sequence of commands is executed. The formal definition of the safety question for access control systems is given below:

Definition Given a protection system, we say a sequence of commands $\alpha_1, \alpha_2, \ldots, \alpha_n$ *leaks a right* ρ *from a starting state* Q if the commands when run on Q (in sequence), can execute a primitive operation that allows some user the right ρ that was not allowed Q.

In other words, a system *leaks a right* ρ if and only if it is possible for an unintended subject to get the right ρ on an object o by the execution of a sequence of commands. The authors have proved that the safety question (also called access right leakage problem) is generally undecidable for the HRU model. It is interesting to note that if we enforce the mono-operation restriction, i.e., each command can execute only one of the primitive commands, then the safety question for such a system is NP-complete. However, most useful HRU systems cannot satisfy the mono-operation requirement.

The Typed Access Matrix (TAM) model [28],[29] adds strong typing to the HRU model. The access rights problem for the generic TAM model has been proven to be undecidable but there are certain restricted versions of TAM for which the problem has been proven to be NP-Hard.

3.3.2 Typed Access Matrix

Ravi Sandhu and others have worked on modifying the HRU model to make the safety problem decidable while at the same time maximizing the expressing power [30]. The Typed Access Matrix (TAM) [28] defined by Sandhu combines strong safety properties for propagation of access rights of Schematic Protection

Table 3–2: Primitive Operations in TAM model

enter r **into** $[X_s, X_o]$	**delete** r **from** $[X_s, X_o]$
create subject X_s **of type** t_s	**delete subject** X_s
create object X_o **of type** t_o	**delete object** X_o

Model [31] with the natural expressive power of the HRU model. TAM is obtained by incorporating strong typing into the HRU model [29]. It is important to understand that the types and rights are specified as part of the system definition. The system administrator specifies the following sets for this purpose:

- a finite set of access rights denoted by R, and
- a finite set of object types denoted by T.

The *protection state* is a TAM system is given by the four-tuple (OBJ, SUB, t, AM) interpreted as follows:

- OBJ is the set of objects.
- SUB is the set of subjects, $SUB \subseteq OBJ$.
- $t : OBJ \rightarrow T$, is the *type function* which gives the type of every object.
- AM is the access matrix. We have $AM[S, O] \subseteq R$ where S is a subject and O is an object.

The protection state of the system is changed by means of TAM commands. Each TAM command has the following format:

command $\alpha(X_1 : t_1,\ X_2 : t_2,\ \ldots, X_k : t_k)$

  if $r_1 \in [X_{s_1}, X_{o_1}] \wedge r_2 \in [X_{s_2}, X_{o_2}] \wedge \cdots \wedge r_m \in [X_{s_m}, X_{o_m}]$

  then $op_1;\ op_2; \ldots; op_n$

end

Here α is the name of the command; X_i are formal parameters whose types are t_i; r_i are rights. Each op_i is one of the primitive operations listed in table 3–2.

The rights, types and commands define the system scheme. The scheme can be modified only by the system administrator, who is not considered to be a part of the system. The type of an object is fixed and cannot be changed within the system. Sandhu has proved that TAM has considerable expressive power.

Safety problem in TAM

The access rights leakage problem for the general TAM model is undecidable (refer to the original paper [29] for proof). The author defines a few restrictions on TAM that allow us to make the problem decidable. A TAM system that does not have any command that deletes a right or destroys a subject or object is called Monotonic TAM. In such a system, S, O and $P[s, o]$ are all monotonically non-decreasing. In a given command α, we say that a type t_i is a *child type* in that command if one of the following primitive operations **create subject** X_i **of type** t_i or **create object** X_i **of type** t_i occurs in the body of α. The type t_j is said to be a *parent type* in α if one of the parameters of α is of type t_j and t_j is not a child type in α. The *creation graph* of an TAM scheme is a directed graph with vertex set T and an edge from u to v if and only if there is a command in which u is a parent type and v is a child type. A TAM scheme is *acyclic* if and only if its creation graph is acyclic. A Ternary TAM is identical to TAM except that all commands are limited to a maximum of three parameters. Given these properties, Sandhu has proved the following results:

1. Generic MTAM is undecidable.
2. Acyclic MTAM is NP-Hard.
3. The safety question for acyclic ternary MTAM is decidable in polynomial time in the size of the initial access matrix.

3.3.3 Access Control in UNIX Operating System

The UNIX operating system was developed at Bell Laboratories in the late 1960s. It evolved in a "friendly" environment, on systems that encouraged users to share their files [32]. However, UNIX is by design a very robust system. The following is a brief discussion of the basic principles in UNIX access control.

Files are central to UNIX in ways that are not true for other operating systems. Commands are executable files in specific directories like */bin*, */usr/bin*, etc. System privileges and permissions are controlled in a large part via access to files [33]. Access to files is organized around file ownership and protection.

Each file has a user owner and a group owner. UNIX supports three types of file access: read (r), write (w), and execute(x). A different set of access rights can be set for user owner, group members and others. For example, the access right $rwxr-xr--$ means that the user owner can read, write and execute the file, members of the group owner can read and execute the file but all others can only read the file. The command that is used to alter the access mode is *chmod*.

There are a few other defined file modes, namely, t (Sticky bit, keep executable in memory after exit), s (Set process user ID or group ID on execution), and l (set mandatory file locking on reads/writes). Files that begin with a period are hidden files and are not listed by *ls* command unless used with the $-a$ option. The $-l$ option of *ls* command lists the files along with the modes and owners of the files. The set of commands that are used to modify the access rights of files are:

- *chmod* Specify the protection modes for files.
- *umask* Specify the default mode for newly created files.
- *chown* Change the user owner of a file.
- *chgrp* Change the group owner of a file.

Some variants of UNIX, like AIX and HP-UX provide access control lists which offer a further refinement to the standard UNIX file permissions capabilities. These ACLs consist of three parts:

1. *Attributes* Specifies any special attributes like SETUID and SETGID.
2. *Base permissions* These correspond exactly to the UNIX file modes and specify access rights for owner, group and others.
3. *Extended permissions* These are access information specified by user and group name.

ACLs that specify a username and group are useful for accounting purposes. They are also useful if you add a user on a temporary basis [33].

UNIX was the first major network operating system. It provides various services like NIS (Network Information Services) and NFS (Network File System) that help in administering a network. Using NIS, users can log on to any machine on the network that belongs to the same NIS Domain. In a network with twenty machines, the administrator has to add the new user on just one machine and that user can log on to any one of these twenty machines. In order to provide a uniform directory structure when a user logs in, administrators mount file systems over the network. Usually, the user's home directories are mounted using NFS. This way the user's files are available to him/her on any machine in the network. The advantage with this arrangement is that controlling file permissions is the same as described earlier.

Users authenticate themselves to the system by providing a user name and password. It is very important for users to protect their passwords because if the password is compromised, then all logins may be compromised [32]. Passwords and file permissions are two basic ways of preventing security problems in UNIX. Passwords prevent bad guys from getting on the system in the first place and proper file permissions prevent normal users from doing things they are not

supposed to [33]. There are only three access types and users cannot define their own access types. Moreover, the owner of a file has full rights to do anything with the file. In many organizations, the files are not "owned" by the users but by the organization itself.

3.4 Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) [34, 35, 36, 37, 38] is a model that is gaining popularity. The principal motivations behind RBAC are the ability to articulate and to enforce enterprise-specific security policies and to streamline the typically burdensome process of security management.

In many organizations, the end users do not "own" the information for which they are allowed access. The corporation is the actual "owner" of system objects as well as the programs that process it and control is often based on employee functions rather than data ownership.

In RBAC, users do not have discretionary access to enterprise objects. Instead, access permissions are associated with roles and users are associated with roles. A role based access control policy bases access control decisions on the roles a user is allowed to take on within an organization. The determination of role membership and the allocation of transactions to a role is in compliance with organization-specific protection guidelines derived from existing laws, ethics, regulations, or generally accepted practices. With RBAC, users are not granted permission to perform operations on an individual basis, but operations are associated with roles. This has the advantage of simplifying the understanding and management of privileges.

System administrators control access at a level of abstraction that is natural to the way enterprises typically conduct business. RBAC addresses security primarily for application-level systems, as opposed to general purpose operating systems.

RBAC enforces the following rules on role authorization, role allocation, and operation execution. RBAC constraints are used to express separation of duty.

1. Role hierarchy If a subject is authorized to access a role and that role contains another role, then the subject is also allowed to access the contained role.

2. Static separation of duty A user is authorized as a member of a role only if that role is not mutually exclusive with any of the other roles for which the user already possesses membership.

3. Cardinality The capacity of a role cannot be exceeded by the addition of another role member.

4. Role authorization A subject can never have an active role that is not authorized for that subject.

5. Role execution A subject can execute an operation only if the subject is acting within an active role.

6. Dynamic separation of duty A subject can become active in a new role only if the proposed role is not mutually exclusive with any of the roles in which the subject is currently active.

7. Operation authorization A subject can execute an operation only if the operation is authorized for the role in which the subject is currently active.

8. Object access authorization A subject can access an object only if the role is part of the subject's current active role set, the role is allowed to perform the operation and the operation to access the object is authorized.

3.5 Distributed Compartment Model

The Distributed Compartment Model [39, 3] was proposed by Steven Greenwald as a solution to management of system resources and access control in a distributed computing environment. The model consists of two parts, (i) *Distributed Handles*, a means for user identification and access control and (ii) *Distributed Compartments*, a method for allowing users to manage resources within a distributed system across computer system boundaries with a measure of independence from any system administrations [39]. The distributed handles eliminate

groupware application userIDs as a means of identification and access control. A user joining a groupware session is queried for a handle that is unique to that application and is then verified by a groupware security manager.

Under this method, an individual user would first gain access to a particular computer system in the distributed system by having a valid user ID and password. The user would then need a valid distributed handle and would then need to be validated by the groupware's access control security in order to be allowed access to the application.

The major advantage of this approach is that security dependencies are reduced. Moreover, the handles can be more descriptive than user IDs, for example "Third programmer" instead of "av0". Multiple handles can be permitted for the same user and many users can share the same handle. Distributed compartment (also called discom) is the designated platform for access control and administration of distributed handles.

A discom is a logical group of objects that is conceptually similar to a standard hierarchical directory structure but does not necessarily reside in a single computer. The users of discoms gain access via distributed handles. A root discom is called an *empire discom*. Each discom must have at least one subject called *governor*. Governors have the maximum privileges for the discom they govern.

There are 24 basic operations called *initial privileges*. They include operations that create, destroy, or modify objects, create, delete, merge, split, or modify child discoms and empire, create or rescind privilege, create or remove governorships, subjects or resources, etc. This combination of subjects, objects and privileges, makes it possible to create a system similar to an access control matrix.

The Distributed Compartment Model has a set of axioms and properties some of which are:

- Divine Right Axiom A subject can create an empire discom only if given that privilege by the administrators.
- Creator Property The creator of a discom automatically becomes a governor of that discom.
- Government Property The governor of a discom may grant and revoke privileges to non-governor subjects of that discom.
- Nova property A non-governor subject may access a descendant discom only if made a member of that discom by a governor of an ancestor discom.
- Cordon Property Discoms may never intersect with other discoms.
- Demesne Property The governor of a discom always has unrestricted access to descendant discoms.
- Ceiling Property A subject may not access an ancestor discom without being a subject of that discom.

A distributed compartment is actually a groupware application, with access to the discoms determined by distributed handles. Management of discoms is not done by the local system administrators but by the individual users who are governors of empire discoms.

3.6 Access Control for Collaborative Environments

Based on the work by Greif [40] and Ellis [41], Bullock and Benford [42] have identified four basic requirements for collaborative access models:

- The mechanism must be simple.
- The mechanism should be unobtrusive to users.
- It should be easy to inspect and change access rights.
- The effects of access controls should be understood, and the consequences of any changes should be clear.

Haake et al. [43] "... found that end-users in order to conduct cooperative work in shared workspace system have to be able to (1) create groups dynamically

without prior planning of a system administrator, (2) employ different forms of group formation, and (3) control access rights to their workspaces." Many current systems lack sufficient support for these requirements.

Shen and Dewan [44] have extended the traditional access matrix model for groupware with more than 50 basic access rights to handle the various operations required for collaboration and inheritance based on right groupings. They also support the notion of *negative rights* to allow explicit denial of rights.

Park and Hwang in [45] tailor the generic architecture for controlled Peer-to-Peer (P2P) computing environments to support RBAC. For collaborative enterprise in such an environment, they identify three different policies: enterprise, community and peer. A community policy defines the community's User-Role Assignment (URA), Permission-Role Assignment (PRA), constraints and role-hierarchy. An enterprise policy defines the enterprise's URA, PRA, constraints, role-hierarchy and role ontology (an equivalence relation between roles in different communities [46]). Each peer defines its own peer policy including the peer's PRA and constraints. The enterprise policy is enforced by a centralized mechanism that applies to all communities in that enterprise, while a community policy is enforced by a centralized mechanism within the community. These two mechanisms are based on the brokered P2P model. If a conflict occurs between policies, enterprise policies are superior to community policies, which are superior to peer policies unless specific policy priority is defined.

Jaeger and Prakash [47] define a discretionary access control (DAC) model for specifying the access rights available to a mobile agent which enables the reader and writer in a mobile agent computation to flexibly control access to system objects. They also specify the requirements of RBAC models necessary to implement the DAC model. In this type of RBAC model, system administrators define roles for users (as in regular RBAC) but can also define a model that will

enable users to limit the access rights of their own processes. First, system admins specify the object types that can be used to specify operation access rights. Also, system admins specify the users who are authorized to limit their own rights dynamically.

3.7 Trust and Voting

While significant work has been done in the fields of Political Science, Psychology, and, Ethics [48, 49], "most of the work concerning trust in computer science have been concentrated in the area of security ... mainly in the form of formal logics to analyze cryptographic protocols for design flaws and correctness." [50].

One of the most popular definitions of trust is given by Deutsch [51].

> (a) an individual is confronted with an ambiguous path, a path that can lead to an event perceived to be beneficial or to an event perceived to be harmful; (b) he perceives that the occurrence of these events is contingent on the behavior of another person; and (c) he perceives the strength of a harmful event to be greater than the strength of a beneficial event. If he chooses to take an ambiguous path with such properties, he makes a trusting choice; else he makes a distrustful choice.

Gambetta [52] has defined trust as " a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action." This definition has been widely accepted by computer scientists Gambetta introduced the concept of using values for trust and also defended the existence of competition among cooperating agents. Trust is closely related to reputation [53]. Abdul-Rahman and Halies [50] define reputation as an expectation about an individual's behavior based on information about or observations of its past behavior. In online communities, where an individual may have very less information to determine the trustworthiness of others, their reputation

information is typically used to determine the extent to which they can be trusted. An individual who is more reputed is generally considered to be more trust worthy. Reputation can be determined in several ways. For example, a person may either rely on his direct experiences, or rely on the experiences of other people, or a combination of both to determine the reputation of another person.

As mentioned earlier, work concerning trust in computer science has been primarily focused on cryptography like in [54] and work in voting has be dominated by electronic voting [55] where the emphasis is on authentication and guaranteeing that the vote counted as voted. Marsh's model [56], as far as we know, is one of the few attempts to formalize trust based on the real world social properties of trust.

3.8 Summary

The HRU model is a very powerful model in terms of expressive power. However, it is very primitive in its construction and has no roles, object types or other abstractions. Therefore the matrix can be very large (and sparse). It is not easy to implement this model in a distributed system.

The TAM model takes a huge step forward with the addition of typing. However, the set of types is static and there are no operations defined in the system to change the type of an object or subject. As we will show in the later sections, making it possible for the users to define their own types and change the types allows us to implement many important and interesting system policies.

RBAC provides a means of naming and describing relationships between individuals and rights. Some form of RBAC is used in commercial systems today and some attempts have been made to formalize the model (refer [36]). One of the major areas in which the DCS model scores over RBAC is group decisions. Another restriction is the absence of some mechanism that allows defining one-to-one relationships between subjects and objects. For instance, consider the ownership relation. Pure RBAC can handle the case where each object has only one owner,

however, if some objects in the system have a small subset of users as its owner(s), it is impractical to define new roles such as *owner of object 1* (we will need as many such "owner roles" as there are objects).

One major disadvantage with the Discom model is that users will have to have a separate handle for each application they use. This means that they will have to remember not just their user ID and password but the handle and password for each application. It would have been preferable to avoid such multiple "logins". Since many users can share handles, this reduces accountability. The Discom model practices a type of monarchy in which the governors have absolute power whereas a democratic form of government will be more practical in the long run. The Discom model is really concerned with resource management in hierarchical groups distributed over multiple domains. The DCS model is concerned with authorization issues and these two models can be used together, providing a secure, flexible system.

All the above models require a system administrator to handle the changes in the system. Incorporating group voting mechanisms allow the users to decide and implement the group policies. This is one of the main features of the DCS model.

The Jaeger and Prakash model allows system administrators to specify some DAC features and who can use them. The users with the privilege can control access to their processes but this is still very limited DAC and does not allow the groups to decide their own policies (they can do so only for those objects the system admins allow). The RBAC scheme for P2P environments described above relies extensively on server controlled decisions and does not support group decisions. It does however support community-level autonomy which allows policy decision to be made off-line and changes are enforced by someone with admin-level privileges.

CHAPTER 4
THE DCS-AC MODEL

4.1 Introduction

In this chapter, we will introduce and formally specify the Distributed Conferencing Services - Access Control (DCS-AC). We will also discuss the constraints and rules that are imposed on any system that implements this model. Finally, we will discuss some of the key features of the model and show the DCS-AC solution to the software problem mentioned in section 1.2.

4.2 The DCS-AC Model

The access control matrix is often very big and sparse. Role-based Access Control (RBAC) groups subjects into *roles* and objects into *object types*, thereby shrinking the matrix. In these systems, each cell in the matrix correspond to the set of rights the corresponding role (identified by the row) had with respect to the object type (identified by the column). In the DCS-AC model, we are concerned with two additional aspects, decision template and target.

The decision template is a pointer to some voting template (see Section 6.2.2) which should be invoked in order to make a decision on the access request. We expect most entries in the matrix to point to the *Always-yes* template which simply returns a *yes* without calling for a vote. Typically, requests that result in granting a right to some role might require a vote but once granted, exercising that right may not require one. It is up to the users to decide which requests require a vote and which do not require one. Needless to say, any request that requires a vote will take some time to be processed and so, the group policies should be designed carefully.

The target field is a very powerful idea which helps us achieve fine-grained access control (see Section 4.6.5). Instead of saying "role *President* can allow subjects to bind to the role *ExecutiveMember*", by using a target field, we can specify the right in finer detail by saying "role *President* can allow subjects of role *TrustedMember* to bind to the role *ExecutiveMember*". Here, the target is the role *TrustedMember* since we are narrowing down the scope of this right to just this role. Depending on the type of command being executed, the target can be a role, object type or a right. In the case of rights for which the target field is not applicable, it is *null* and the keyword *any* specifies that the right can be exercised on any target.

The distribution of rights in the system are can be viewed as an access matrix (implemented as a five-field RDBMS table), A. The cell $A[X, Y]$ contains a set of (ρ, Z, dp) tuples where ρ is a right which role X has for objects of type Y with respect to a target, Z and dp is a pointer to a decision template that must be activated each time X makes a request to perform ρ on (Y, Z).

We achieve a great deal of flexibility by allowing the group to dynamically change a major part of the system components. Unlike the TAM model, we allow types (i.e., roles and object types) and type bindings (subject-role and object-type bindings) to be added and deleted dynamically by the group according to the group policy as represented by the matrix. Similarly, the set of rights and the available decision templates can also be changed while the system is in operation. The only static component of the model is the set of commands which are the same for all instances of the model.

Each subject can bind to one or more *roles* but at any point of time, the subject can be active in only one role. The user can temporarily change his/her active role to another role to make a request that is not allowed for his/her current role. One of the main reasons for this restriction is that the user has to be aware

of the increased responsibility while requesting access that requires membership to a privileged role. Each object is bound to one *object type*. These bindings can be changed by anyone who has the right to do so. This is a major difference from the TAM model in which the bindings are a part of the system definition and can only be altered by an external administrator. In the DCS-AC model, the administrator is considered to be a part of the system and is subject to the access restrictions imposed by the matrix. An *instance* of DCS-AC has the following finite dynamic sets:

- a set of access rights denoted by R,
- a set of decision templates denoted by D,
- a set of object types denoted by T,
- a set of roles denoted by $T_R(\subset T)$,
- a set of objects denoted by OBJ, and
- a set of subjects denoted by SUB.

The access control matrix is denoted by A. We have three mapping functions, f_r which maps each subject to a subset of T_R (corresponding to the roles to which the subject can bind), f_o which maps each object to an element of T (its object type)[1] , and f_a which maps each *active* subject to his/her *active* role.

$$f_r \ : \ SUB \rightarrow 2^{T_R}$$

$$f_o \ : \ OBJ \rightarrow T$$

[1] If an object is allowed to be of more than one type, access control decisions are difficult. Should a user have rights to access all object types of an object in order to access the object? An alternative is to define an object type hierarchy, which increases the complexity of the model.

Table 4–1: Example of DCS-AC Matrix Fragment

	Role	**Object Type**	**Target**	**Right**	**Template**
1	Programmer	Code	*null*	Read	d_{yes}
2	Programmer	Code	*null*	Write	d_{yes}
3	Programmer	Code	Read	GrantRight	d_1

$$f_a \ : \ SUB \rightarrow T_R$$

The state of a DCS system is the tuple $(A, R, D, T, T_R, OBJ, SUB, f_r, f_o)$ and as noted earlier, the set of commands C (defined in section 4.4) is the only static component of the model. A few simple examples of the matrix entries are given in table 4.2. Here, Programmer is a role and Code is an object type. Read, and Write are access types and d_{yes} is a pointer to the *Always-yes* template. The first three entries allow subject who are actively in the role Programmer to read and write objects of type Code. The third entry allows Programmers to grant Read access to anyone if the decision template d_1 returns yes. The decision template d_1 might require the approval of the Project Leader and Dev Team Manager. *GrantRight* is a special access right that corresponds to a primitive operation that adds an entry to the matrix.

4.3 Operations

Lets us now look at the primitive operations that can be executed on a DCS-AC system. These are operations that modify one or more components of the system and are guaranteed to be atomic. We will formally define the operations by specifying their effect on the system. Let (A, R, D, T, T_R, OBJ, SUB, f_r, f_o) denote the system before the operation and (A', R', D', T', T'_R, OBJ', SUB', f'_r, f'_o) denote the system after the operation. Table 4.3 defines the primitive operations in DCS-AC. Note that sets not mentioned in the definition of an operation are unchanged by that operation. There is one command for every operation but we might want to have commands with two or three operations especially if we want roles and object types to have pedigree (see section 7.1).

Table 4–2: Specification of DCS-AC Operations

Operation	Precondition	Postcondition
grantright(r, ot, ρ, t, dp)	$r \in T_R, t \in T \cup R,$ $t \in T, \rho \in R, dp \in DT,$ $\forall dp_1 \in DT,$ $(\rho, t, dp_1) \notin A[r, ot]$	$\forall\ (r_1, ot_1) \in\ T_R \times T,$ $((r_1, ot_1) \neq\ (r, ot)) \rightarrow A'[r_1, ot_1] = A[r_1, ot_1]$ $A'[r, ot] = A[r, ot] \cup\ \{(\rho, t, dp)\}$
revokeright(r, ot, ρ, t)	$r \in T_R, t \in T \cup R$ $ot \in T$ $\rho \in R$	$\forall\ (r_1, ot_1) \in\ T_R \times T,$ $((r_1, ot_1) \neq\ (r, ot)) \rightarrow A'[r_1, ot_1] = A[r_1, ot_1]$ $A'[r, ot] = \{(\rho_1, t_1, dp_1) :$ $(\rho_1, t_1, dp_1) \in\ A[r, ot] \wedge\ (\rho_1 \neq \rho) \wedge\ (t_1 \neq t)\}$
createrole(r)	$r \notin T_R$	$T'_R = T_R \cup\ \{r\}$ $\forall\ (r_1, ot_1) \in\ T_R \times T, A'[r_1, ot_1] = A[r_1, ot_1]$ $\forall\ ot_1 \in T,\ A'[r, ot_1] = \emptyset$
deleterole(r)	$\neg\exists s, f_r(s) = \{r\}$	$T'_R = T_R - \{r\}$ $\forall\ (r_1, ot_1) \in\ T'_R \times T,\ A'[r_1, ot_1] = A[r_1, ot_1]$ $\forall\ s_1 \in SUB,\ f'_r(s) = f_r(s) - \{r\}$
createot(ot)	$ot \notin T$	$T' = T \cup \{ot\}$ $\forall (r_1, ot_1) \in T_R \times T', A'[r_1, ot_1] = A[r_1, ot_1]$ $\forall r_1 \in T_R,\ A'[r_1, ot] = \emptyset$
deleteot(ot)	$\neg\exists o, f_o(o) = ot$	$T' = T' - \{ot\}$ $\forall (r, ot) \in T_R \times T',\ A'[r, ot] = A[r, ot]$
addobject(o, ot)	$o \notin OBJ$ $ot \in T$	$OBJ' = OBJ \cup \{o\}$ $\forall o_1 \in OBJ,\ f'_o(o_1) = f_o(o_1)$ $f'_o(o) = ot$
delobject(o)		$OBJ' = OBJ - \{o\}$ $\forall o_1 \in OBJ',\ f'_o(o_1) = f_o(o_1)$
addsubject(s, r)	$r \in T_R$	$SUB' = SUB \cup \{s\}$ $\forall\ s_1 \in SUB,\ f'_r(s_1) = f_r(s_1)$ $f'_r(s) = \{r\}$

Table 4.2: Continued

Operation	Precondition	Postcondition
delsubject(s)		$SUB' = SUB - \{s\}$
		$\forall s_1 \in SUB',\ f'_r(s_1) = f_r(s_1)$
addaccess(ρ)		$R' = R \cup \{\rho\}$
delaccess(ρ)		$R' = R - \{\rho\}$
		$\forall\ (r, ot) \in\ T_R \times T,$
		$A'[r, ot] = \{(\rho_1, t, dp) :\ (\rho_1, t, dp) \in\ A[r, ot] \wedge\ (\rho_1 \neq \rho)\}$
addrolebinding(s, r)	$s \in SUB$	$\forall s_1 \in SUB - \{s\},\ f'_r(s_1) = f_r(s_1)$
	$r \in T_R$	$f'_r(s) = f_r(s) \cup\ \{r\}$
delrolebinding(s, r)	$s \in SUB$	$\forall s_1 \in SUB - \{s\},\ f'_r(s_1) = f_r(s_1)$
	$r \in T_R$	$f'_r(s) = f_r(s) - \{r\}$
	$f_r(s) \neq \{r\}$	
changedp(r, t, ot, ρ, dp)	$r \in T_R, t \in T \cup R$	$\forall\ (r_1, ot_1) \in\ T_R \times T,$
	$ot \in T$	$((r_1, ot_1) \neq\ (r, ot)) \rightarrow A'[r_1, ot_1] = A[r_1, ot_1]$
	$\rho \in R$	$A'[r, ot] = \{(\rho_1, t_1, dp_1) : (\rho_1, t_1, dp_1) \in A[r, ot] \wedge \rho_1 \neq \rho \wedge t_1 \neq t\}$
	$dp \in DT$	$\cup \{\rho, t, dp\}$
changeot(o, ot)	$o \in OBJ$	$\forall o_1 \in OBJ, o_1 \neq o \rightarrow f'_o(o_1) = f_o(o_1)$
	$ot \in T$	$f'_o(o) = ot$

4.4 Commands in DCS-AC Model

The access control matrix can only be modified through the set of commands which is the only static component in the DCS-AC model. Unlike the HRU and other models, the set of commands does not vary from system to system. There is one command for each primitive operation; it checks if the subject's role has the right to perform the operation and if so, then it performs the operation. A *command* consists of two parts: a conditional part that checks if the role is allowed to make the request and an execution part that executes one operation. All objects belonging to the access control model (like A, f_o, DT, T, T_R etc.) are of the object type Γ. Given below is the list of commands in DCS-AC. In all these commands, r is the role of the subject executing the command.

- `command CreateRole(`$r \in T_R, r_{new} \notin T_R$`)`
  `if ((`***CREATEROLE,null,dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $createrole(r_{new})$

- `command DeleteRole(`$r \in T_R, r_1 \in T_R$`)`
  `if` $\neg\exists s, f_r(s) = \{r_1\}$ `and ((`***DELETEROLE,***$null$***, dp***`)` $\in A[r, r_1]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $deleterole(r_1)$

- `command GrantRight(`$r \in T_R, r_1 \in T_R, t \in T \cup R, ot \in T, \rho \in R, dp_1 \in DT$`)`
  `if` $\neg\exists dp_2, (\rho, t, dp_2) \in A[r_1, ot]$ `and ((`***GRANTRIGHT,***ρ***, dp***`)` $\in A[r, ot]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $grantright(r_1, ot, \rho, t, dp_1)$

- `command RevokeRight(`$r \in T_R, r_1 \in T_R, t \in T \cup R, ot \in T, \rho \in R$`)`
  `if ((`***REVOKERIGHT,***ρ***, dp***`)` $\in A[r, ot]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $revokeright(r_1, t, ot, \rho)$

- `command CreateOT(`$r \in T_R, ot \notin T$`)`
  `if ((`***CREATEOT,***$null$***, dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $createot(ot)$

- `command DeleteOT(`$r \in T_R, ot \in T$`)`
  `if` $\neg\exists o, f_o(o) = ot$ `and ((`***DELETEOT***`,`$null$`, ***dp***`)` $\in A[r, ot]$`) for some`
  $dp \in DT$ `and` dp `returns` $true$ `then`
  $deleteot(ot)$

- `command AddSubject(`$r \in T_R, s_{new} \notin SUB, r_{initial} \in T_R$`)`
  `if ((`***ADDSUBJECT***`,`$r_{initial}$`, ***dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $addsubject(s, r_{initial})$

- `command DelSubject(`$r \in T_R, s \in SUB$`)`
  `if ((`***DELSUBJECT***`,`$null$`, ***dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $delsubject(s_1)$

- `command AddObject(`$r \in T_R, o \notin OBJ, ot \in T$`)`
  `if ((`***ADDOBJECT***`,`$null$`, ***dp***`)` $\in A[r, ot]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $addobject(o, ot)$

- `command DelObject(`$r \in T_R, o \in OBJ$`)`
  `if ((`***DELOBJECT***`,`$null$`, ***dp***`)` $\in A[r, ot]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $delobject(o)$

- `command AddRoleBinding(`$r \in T_R, s \in S, r_1 \in T_R$`)`
  `if ((`***ADDROLEBINDING***`,`r_s`, ***dp***`)` $\in A[r, r_1]$`) for some` $dp \in DT$ `and`
  $r_s \in f_r(s)$ `and` dp `returns` $true$ `then`
  $addrolebinding(s, r_1)$

- `command DelRoleBinding(`$r \in T_R, s \in S, r_1 \in T_R$`)`
  `if` $f_r(s) \neq \{r_1\}$ `and ((`***DELROLEBINDING***`,`$null$`, ***dp***`)` $\in A[r, r_1]$`) for some`
  $dp \in DT$ `and` dp `returns` $true$ `then`
  $delrolebinding(s, r_1)$

- `command ChangeOT(`$r \in T_R, o \in OBJ, ot_{new} \in T$`)`
  `if ((`***CHANGEOT***`,`$f_o(o)$`, ***dp***`)` $\in A[r, ot_{new}]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $changeot(o, ot_{new})$

- `command AddAccess(`$r \in T_R, \rho \notin R$`)`
  `if ((`***ADDACCESS***`,`$null$`, ***dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp
  `returns` $true$ `then`
  $addaccess(\rho)$

- `command DelACCESS(`$r \in T_R, \rho \in R$`)`
  `if ((`***DELACCESS,***ρ***, dp***`)` $\in A[r, \Gamma]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $delaccess(\rho)$

- `command ChangeDP(`$r \in T_R, r_1 \in T_R, t \in T \cup R, ot \in T, \rho \in R, dp_{new} \in DT$`)`
  `if` $\exists dp_1, (\rho, t, dp_1) \in A[r_1, ot]$ `and ((`***CHANGEDP,***ρ***, dp***`)` $\in A[r, ot]$`) for some` $dp \in DT$ `and` dp `returns` $true$ `then`
  $changedp(r_1, t, ot, \rho, dp_{new})$

It is obvious that all the commands can be executed in constant time (assuming an that looking up the access matrix and other DCS-AC components can be done in constant time). Moreover, the roles and object types created by the *CreateRole* and *CreateOT* commands have no pedigree, i.e., the creator of the role or object type has no special rights over that role or object type. We also allow the object type, access right and the target to be replaced by the keyword *ANY*. For example, if $(ANY, ANY, dp) \in A[Secretary, ANY]$ where dp is a decision template that calls for a simple majority of all subjects who can bind to the role *Senator*, then a Secretary can make any change to the system if approved by a majority of the senators. Since the roles and object types have no pedigree, there has to be a group policy for granting rights to these roles and object types (typically, $(GRANTRIGHT, ANY, someDP) \in A[SomeRole, Any]$).

4.4.1 Constraints

There are a few constraints that have to be enforced on the system, some of which have been mentioned earlier.

- Active Role At no point of time can a subject be active in more than one role.

- Role Authorization A subject can only be active in roles he/she is authorized to be in.

- Operation Authorization A subject can perform only those operations that his/her active role is authorized to do.

- Object Access Authorization A subject can only access those objects his/her active role is authorized to do.

- Amendment Rule There should always be a rule that allows the group to modify any component of the system.

- Delete Restriction A role or object type cannot be deleted if it results in a violation of the rules and constraints.

- Graceful Deletion If an active subject is deleted from the system, the deleted subject should lose all right immediately.

- Decision Template Overwrite Rule A *GrantRight* command should not be allowed to change the decision template of an already granted right.

The *Active Role* constraint is enforced to simplify the access checking process. The *Role Authorization*, *Operation Authorization*, and *Object Access Authorization* are RBAC enforced constraints. *Amendment Rule* is like an emergency override to allow the group to recover from unusable states. Typically, this is enforced by a matrix entry that allows a particular role to change anything if some template returns a yes. For example, the *Chairman* can change any rule if all the *Faculty* and the *College Dean* agree.

The *Delete Restriction* is imposed on deletion of roles and object types. If there is a subject that can bind to only one role, then deleting that role means that the subject cannot bind to any role. This is an inconsistent state and should be avoided. This subject should be deleted or allowed to bind to another role before deleting this role. Furthermore, if there is a subject currently active in a particular role, then that role cannot be deleted. Similarly, if there is at least one object that belong to a particular type, that type cannot be deleted.

The *Graceful Deletion* rule is for deleting subjects who are currently active. They should be immediately logged out of the system with suitable notification messages. The *Decision Template Overwrite Rule* prevents some subject from changing the decision template for an entry in the matrix by using the *GrantRight*

command. This is enforced by ensuring that no *GrantRight* command grants an already existing command. The decision template pointers for an entry can only be changed by the *ChangeDP* command.

4.4.2 Expressive Power

It is well known that the HRU model has very good expressive power but is very weak in terms of provable safety. It can be seen that the DCS-AC model also has very good expressive power. Any HRU protection state can be converted to the DCS-AC model.

Mapping from HRU to DCS-AC

Consider a protection system (R, C) and a state (S, O, P). Let us define a new role for each user and allow only that user to bind to that role and call the set of these roles T_R. Similarly, define a new object type for each object and call this set T. We define $DT = \{dp\}$ where dp is a pointer to a decision template that always return *yes*. Let the new DCS-AC instance be (R, DT, T, T_R, O, S, f_o, f_r, A).

- $\forall\ o_i \in O$ add a new object type ot_i to T and let $f_o(o_i) = ot_i$
- $\forall\ s_i \in S$ add a new role r_i to T_R and let $f_r(s_i) = \{r_i\}$.
- $A[r_i, ot_i] = \{(\rho, null, dp) : \rho \in P[s_i, o_i]\}$

Now we have an instance of DCS-AC model defined by (R, DT, T, T_R, O, S, f_o, f_r, A). Although the system state in the two models are equivalent, the systems themselves are not equivalent because we have not mapped the commands of the HRU model to something equivalent in the DCS-AC model.

Mapping from DCS-AC to HRU

Similarly, we can do a mapping from DCS-AC to HRU. Here again, the two systems will not be equivalent because of dynamic typing and decision pointers. Consider an instance of the DCS-AC model (R, DT, T, T_R, OBJ, SUB, f_r, f_o, A).

We can convert this into a HRU system (R', C) with the configuration (S, O, P) as follows:

1. $R' = R - R_m$ where R_m is the set of all matrix manipulation operations not supported by HRU.

2. $S = SUB$ and $O = OBJ$

3. $\forall\ (s, o) \in SUB \times OBJ$
   (a) $\forall r \in f_r(s),\ P[s, o] = \{\rho : (\rho, null, dp) \in A[r, f_o(o)] \wedge \rho \notin R_m\}$

4. Add commands to perform each of the six operations if the subject has the right similar to the one specified in section 4.4.

Note that the resulting HRU instance is a mono-operational one and hence the safety of this is decidable in NP time. Here, we drop all typing information, decision pointers and operations not allowed by HRU.

4.5 Comparison with existing models

One major difference between the DCS-AC model and the other models is the fact that the ACM itself is an object and there is a right corresponding to each of the above mentioned operations. If a user wants to perform a matrix modifying operation, then his/her role should have the right to do so. It must be noted that in TAM, R, T, T_R and t are all part of the system definition and cannot be altered by anyone within the system, whereas these are a part of the DCS-AC model instance and can be modified by anyone who has the right to do so. TAM does not have the notion of decision pointers either. TAM allows any number of parameters in its commands but the DCS-AC allows at most five.

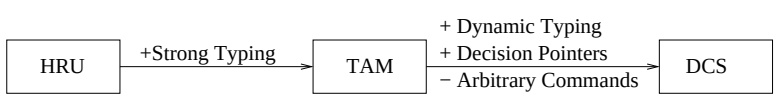

Figure 4–1: Comparison of HRU, TAM and DCS-AC models

The TAM model added the notion of strong typing to the HRU model. The DCS-AC model has dynamic typing, static set of commands (as opposed to

arbitrary commands) and most important of all, decision template pointers that allows the members of the group to decide on access right issues. Another major difference is the dynamic set of access rights. As mentioned earlier, the term *role* as used in the DCS-AC model is different from what it means in RBAC. Here are some differences between the two:

- Role Hierarchy There is no role hierarchy in the current implementation of the DCS-AC model. Subsection 7.3 talks about how the DCS-AC model can be extended to implement role hierarchy.

- Static Separation of Duty The DCS-AC model does not support static separation of duty. However, it is possible to implement dynamic separation of duty using function calls. All separation of duty policies can be implemented as dynamic which provides greater flexibility to the users.

- Decision Pointers The use of decision pointers is the new paradigm introduced in this model. As mentioned earlier, this addition is very useful in specifying various system policies.

4.6 Features of DCS-AC Model

The access control matrix and the functions mapping various sets are stored as tables in a backend database which is replicated across all sites. All updates and deletions are propagated to other sites by a separate Database Services module (DBS) which guarantees processor consistency for updates, i.e., updates originating from the same processor will be committed at all sites in the same order. This is achieved by assigning an owner site for each tuple in the table. When a new tuple is to be added, the site the request originated is the owner. Modifications to a particular tuple are forwarded to its owner which then multicasts to other sites. The decision templates are maintained by the Decision Support Services module (DSS) which stores the vote participants, percentage of votes required for success and other information about the template. When the access control module initiates a decision, the participants are notified and the votes are collected. Once a decision is reached, the DSS passes it to the access control module which takes

further action as required. Note that in many cases, the decision pointer may point to a template that always returns a *yes*.

4.6.1 Decision Templates

Decision templates are one of the most interesting features of the DCS-AC model. The use of group voting mechanisms allow the users to self-administer their groups. It is quite clear that the safety of the system will depend very heavily on the decision pointer. It will be very interesting to study the various types of collusion attacks possible based on a given set of decision pointers. A decision template contains the following information: (i) A list of voters along with the weight of each entity's (role/subject) vote, (ii) number of voters for quorum, (iii) start and end times, (iv) percentage of voter who have to agree for the vote to pass, (v) default policies if quorum is not reached, and (iv) other bookkeeping details like who should be notified of the result etc. The template also specifies what kind of vote to start (ranking, simple majority, multiple rounds, etc.). It is expected that most access requests will invoke a simple decision template that always returns *yes* and only a few access requests (like granting a right to a role or any thing deemed important by the group) will require a vote.

If we allow the decision pointers to be pointers to a voting process or a function call that returns a decision based on the state of the system, we can implement interesting policies like the separation of duty. If the global policy states that any user who has access to objects of type O_1 should not have access to objects of type O_2, then the decision pointer will execute a function that allows the access only if the cell corresponding to $[R, O_1]$ is null for every R in the set of all roles the subject can bind to. However, the use of function pointers might make the access right leakage problem undecidable and is currently not a part of the DCS-AC model.

4.6.2 Dynamic Type Binding

The binding between objects/subjects and their types is static in TAM (HRU does not have types). There are many scenarios in which this binding has to be changed. By allowing the bindings to change, we can also model process work flows. For example, if the group policy states that the review team should not be able to access the code until it passes the testing phase. We can define three different object types, *working code, ready-for-test* and *ready-for-review*. When the code is ready for testing, the project leader or the developer can change the object type of the files to *ready-for-test*. Now the testers can do their job. An additional advantage in doing things this way is that the programmers cannot modify the files that are being tested (if they do not have that access for objects of type *ready-for-test*. If the code fails testing, the PL can change its type back to *working code* else, he/she can change it to *ready-for-review*.

4.6.3 Dynamic set of Access Rights

As mentioned earlier, one of the differences between the DCS-AC model and the other models is the fact that the access rights are not a part of the system definition. When a new application is added to a system with a static set of access rights, fine-grained access control has to be managed by the application itself. In the DCS-AC model, when a new application is added, we can define new access types to the system and make access control decisions at the system level. For example, if a new audio-video application is added, we can define new access types like *play audio file*, *recompress the file*, *cut and crop files*, etc. Now, based on the roles (like producer, sound engineer, etc.) it is straightforward to define access control policies like a producer can play an audio file but cannot modify it, a sound engineer can play, recompress, cut and crop an audio file. We can also define special relations so that only the producers and sound engineers working with a

particular artist can access the audio files recorded by that artist. As we can see, the realm of possibilities is very large.

4.6.4 Dynamic set of Object Types

In most real world scenarios, it is impossible to keep the set of object types a constant. We cannot foresee all possible object types during system initiation and although the need may not be very frequent, it is very likely that any system will need new object types at some point of time. When a the audio-video application mentioned in the previous section is installed, we have to create a new object type (called *audio file*). It is quite possible to just use the type *generic file* and individually specify the access rights for each audio file. This is how it will have to be done in TAM. However, the main advantage of using types is the reduction in the size of the ACM and hence, specifying the rights for each and every audio file is counter-productive in terms of ACM size. Thus it is more logical to allow the set of object types to change with the needs of the users.

4.6.5 Fine Grain Access Control

As can be seen from the commands in DCS-AC (Section 4.4), the use of the *target* field allows us to define very fine grained access control policies. For example, we can specify that role r has the right to grant $read$ access to objects of type ot ($(GRANTRIGHT, read, dp) \in A[r, ot]$) and we can similarly specify revoke rights for specific accesses. Another instance of fine grained access control is the $ChangeOT$ command. If a manager can declassify a $TopSecret$ document to $Secret$ but too only with the approval of the CEO, then $(CHANGEOT, Secret, dp_1) \in A[Manager, TopSecret]$ where dp_1 is a decision template that has only one voter, the CEO. This level of granularity is not available in most access control models in use today.

4.7 DCS-AC Model Solution for Software Project Example

Here is a solution to the software project example mentioned in section ref-motivation using the DCS-AC model. Let the set of roles be $T_R = \{$PL, Architect, Prog, Tester$\}$. When the project (lets call it X) is started the following object types are created: *XDesignDoc, XCode, XWorkingCode, XTestedCode* and *XShipCode*. The following roles are also created: *XPL, XArchitect, XProg* and *XTester*. We now add the entries specified in table 4–3 to the access control matrix, A.

Here, dp_1 is a decision pointer that always returns yes, dp_2 requires a vote of all subjects who can bind to $XProg$ and dp_3 requires a vote of all subjects who can bind to PL. Entries 1,2 and 3 allow the X PL to decide who works on a project but ensures that policy 6 (PL cannot change the role of an employee) is enforced. This is a simplistic view of things and it is very easy to envision a system where *compile* and *execute* are valid accesses which are used by a DCS-aware development environment.

Thus class libraries example is solved by using the DCS-AC model. Similar solutions are suitable for collaborative contract editing and many other distributed collaborative applications. This example also illustrates how the DCS-AC model can be used to model process work flow.

Table 4–3: DCS-AC Solution for Software Project Example

No.	Entry	Explanation
1	$(ADDROLEBINDING, Architect, dp_1) \in A[XPL, XArchitect]$	The PL can assign architects to the project
2	$(ADDROLEBINDING, Prog, dp_1) \in A[XPL, XProg]$	PL can add programmers to the project
3	$(ADDROLEBINDING, Tester, dp_1) \in A[XPL, XTester]$	PL can add testers to the project
4	$(ADDOBJECT, null, dp_1) \in A[XArchitect, XDesignDoc]$	Architects can create design documentation
5	$(read, null, dp_1) \in A[XArchitect, XDesignDoc]$	Architects can read the docs
6	$(read, null, dp_1) \in A[XPL, XDesignDoc]$	The PL can read the docs
7	$(write, null, dp_1) \in A[XArchitect, XDesignDoc]$	Architects can write to docs
8	$(read, null, dp_1) \in A[XProg, XDesignDoc]$	Programmers can read the docs
9	$(ADDOBJECT, null, dp_1) \in A[XProg, XCode]$	Progs can create code files
10	$(read, null, dp_1) \in A[XProg, XCode]$	Progs can read the code files
11	$(read, null, dp_1) \in A[XPL, XCode]$	PL can read the code files
12	$(write, null, dp_1) \in A[XProg, XCode]$	Progs can write to the code files
13	$(CHANGEOT, XCode, dp_2) \in A[XProg, XWorkingCode]$	Progs take a vote to submit code for testing
14	$(read, null, dp_1) \in A[XTester, XWorkingCode]$	Testers can read working code
15	$(CHANGEOT, XWorkingCode, dp_1) \in A[XTester, XCode]$	Testers can return code to programmers
16	$(CHANGEOT, XWorkingCode, dp_1) \in A[XTester, XTestedCode]$	Testers can mark code ready for review
17	$(read, null, dp_1) \in A[PL, XTestedCode]$	All PLs in the company can look at tested code.
18	$(CHANGEOT, XTestedCode, dp_3) \in A[XPL, XShipCode]$	PL can ship code if review team agrees

4.8 Summary

In this chapter, we defined and discussed the DCS-AC model. One of the main features of this model is the ability of the subjects to vote on important (as defined by the group) decisions. All subjects belong to one or more roles and all objects belong to an object-type. The access matrix entry for a particular role and object type contains a set of ordered pairs corresponding to the rights the role has over the object type and the decision template that is executed when a subject of that role tries to use that particular right over the object type. The decision template might point to the *Always-yes* template (which always returns *yes*) or any other template defined by the group. There are sixteen primitive operations and sixteen commands (each of which correspond to one of the operations). These are the only static components of the model. The group can define and modify all other components like roles, object types, etc. dynamically. There are a few constraints that have to be enforced when implementing this model. The commands in DCS-AC are designed in such a way so as to allow fine-grained access control, i.e., we can specify rules like *Role X can allow a subject of role Y to bind to the role Z* instead of just a more general rule like *Role X can allow a subject to bind to role Z*.

It should be remembered that while all these features are really powerful, care must be taken to avoid getting into a state that makes the system difficult to use. The *Amendment Rule* constraint enables the group to recover from any such bad states. In the next chapter, we will prove that the model is provably safe.

CHAPTER 5
SAFETY ANALYSIS

5.1 Introduction

The presence of decision pointers makes the safety analysis of the DCS-AC model very tricky. Consider a restricted version of DCS-AC where there is only one decision pointer in the system, dp, and that pointer always returns $true$. This is a valid restriction because, if trusted deciders control a decision, they will vote no, so we can remove these entries from the matrix and any decision controlled by untrusted deciders will, in the worst case, always result in yes. We will first analyze the safety of such a restricted model and in the next chapter, we will discuss the safety of the full model.

5.2 Definitions

Let us now look at some definitions that will help us understand the safety properties of the DCS-AC model.

Definition The state, S, of a DCS-AC system is a snapshot of the system at a given time. It consists of the access control matrix A, the role binding function f_r, the object type binding function f_o, and the following sets:

- set of rights R,
- set of subjects SUB,
- set of objects OBJ,
- set of object types T,
- set of roles T_R ($T_R \subseteq T$), and
- set of decision templates DT.

Definition For a given state S, object type ot, and right ρ,

$$\Phi_{S,\rho,ot} = \{s : \ s \in SUB \ \wedge \ \exists \ r \in f_r(s), t \in (T \cup R \cup \emptyset) \text{ such that } (\rho, t, dp) \ \in \ A[r, ot]\}$$

In other words, $\Phi_{S,\rho,ot}$ is the set of all subjects who have the right ρ for objects of type ot in S. We can get the set $\Phi_{S,\rho,ot}$ in $O(|SUB| \cdot |T_R| \cdot |R|)$ time as follows:

```
function getPhi(S, ρ, ot)

1. result = ∅

2. for all s ∈ SUB
   (a) for all r ∈ f_r(s)
       i. for all (ρ', t', dp) ∈ A[r, ot]
          A. if ρ' = ρ then
             • result = result ∪ {s}
             • Process next s

3. return result
```

Definition Given two states, S and S', and a command, α, we say $S \rightharpoonup_{\alpha} S'$ if the command α can be executed on S and S' is the result of performing the operation in α on S (i.e., applying α on S results in S').

Definition Given two states S and S', and a sequence of commands σ (containing k commands, $\alpha_1, \alpha_2, ..., \alpha_k$), we say

$$S \rightarrow_{\sigma} S' \text{ if } \begin{cases} k = 0 \ \Rightarrow \ S = S' k = 1 \ \Rightarrow \ S \rightharpoonup_{\alpha_1} S' \\ k > 1 \ \Rightarrow \ S \rightharpoonup_{\alpha_1} S_1 \ \wedge \ S_1 \rightarrow_{\sigma'} S', \text{ where } \sigma' = \alpha_2, \alpha_3, ..., \alpha_k \end{cases}$$

Definition For a given DCS-AC state S, object o, and right ρ, we say that a sequence of commands σ results in a leaking state if and only if

$$S \rightarrow_{\sigma} S' \Rightarrow \Phi_{S',\rho,f'_o(o)} - \Phi_{S,\rho,f_o(o)} \neq \emptyset$$

Let us now define the access right leak problem in the context of the DCS-AC model.

Definition For a given DCS-AC state S, object o, and right ρ, the access right leak problem is a decision problem is the question, does there exist a sequence of commands on S that results in a leaking state.

5.3 Attributes of a State

In order to help us keep track of the commands that have been exectuted and those that can be executed, we identify certain attributes which are dependent on the current state of the system. Each of these attributes can be active (i.e., the command has been executed), enabled (the command has not been executed but there is some role in the current state that can execute the command), or inactive (the command has not been executed nor can any role execute the command in the current state).

For any state S^t (reached by executing 0 or more commands on the starting state S^0), the following attributes are valid:

1. $HasRight_{r',ot',\rho',t}$ This attribute tracks the *GrantRight* commands and is said to be active for S^t if $(\rho', t, dp) \in A(r', ot')$ for some $dp \in DT$. If the attribute is not active, we say that this attribute is enabled if $\exists\ r, dp'$ such that $(GRANTRIGHT, dp') \in A(r, ot')$

2. $CanBind_{s',r'}$ This attribute tracks subject-role bindings and is said to be active for S^t if $r' \in f_r(s')$. If the attribute is not active , we say that this attribute is enabled if $\exists\ r, dp'$ such that $(ADDROLEBINDING, dp') \in A(r, r')$

For a given starting state S^0 and a descendant state S^t (reached by executing 1 or more commands on the starting state S^0, i.e., $\exists\sigma$ such that $|\sigma| \geq 1 \wedge S^0 \rightarrow_\sigma S^t$), we identify the following attributes that are valid for S^t and all states that result in executing a sequence of commands on S^t:

1. $NewRole$ This attribute is said to be active for S^t if one of the commands we have executed on S^0 is a *CreateRole* command. It cannot be active for S^0. If the attribute is not active, we say that this attribute is enabled if $\exists\ r, d'$ such that $(CREATEROLE, d') \in A(r, \Gamma)$

2. $NewOT$ This attribute is said to be active for S^t if one of the commands we have executed on S^0 is a $CreateOT$ command. It cannot be active for the initial state S^0. If the attribute is not active, we say that this attribute is enabled if $\exists\ r, d'$ such that $(CREATEOT, d') \in A(r, \Gamma)$

3. $NewSubject_{r'}$ This attribute is said to be active for a state S^t if one of the commands executed is a *AddSubject* command with the initial role r'. These attributes cannot be active for S^0. If the attribute is not active , we say that this attribute is enabled if $\exists\ r, dp'$ such that $(ADDSUBJECT, dp') \in A(r, r')$

These attributes are activated by executing the corresponding commands.

$Active(S^i)$ is the set of attributes that are active in a state S^i and $Enabled(S^i)$ is the set of attributes that are enabled in S^i.

5.4 Proofs

Let us now look at a few lemmas and theorems that will lead to a polynomial time solution to the ARL problem for at DCS-AC system without voting.

Definition We say that two states S^i and S^j (with the same starting state S^0) are *equivalent* in terms of access right leakage if:

$$S^i \text{ is a leaking state } \Leftrightarrow S^j \text{ is a leaking state}$$

.

Axiom 1 *In a DCS-AC system with an initial state S, two sequences of commands, σ_1 and σ_2, result in equivalent states if the active attributes of the two resulting state are the same.*

$$(S \rightarrow_{\sigma_1} S^1) \wedge (S \rightarrow_{\sigma_2} S^2) \wedge (Active(S^1) = Active(S^2)) \Rightarrow S^1 \equiv S^2$$

Lemma 2 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' results in a leaking state and*

- *σ' does not contain any command that revokes a right or deletes a subject, object, role, object type or access.*

Proof Let $\sigma = \alpha_1, \alpha_2, ..., \alpha_k$ and let α_i be the first revocation or deletion command.

Let j ($1 \leq j \leq k$) be the smallest number such that the sequence $\sigma_1 = \alpha_1, ..., \alpha_j$ results in a leaking state S'. Clearly, $j \neq i$ since α_i is a revocation or deletion command.

Case 1: $j < i$

If $j < i$, then the sequence σ_1 results in a leaking state and does not contain any revocation or deletion commands (since α_i is the first such command and $j < i$) and hence σ_1 is the sequence we are looking for.

Case 2: $j > i$

Consider the sequence $\sigma_2 = \alpha_1, ..., \alpha_{i-1}, \alpha_{i+1}, ..., \alpha_j$. We know that the commands $\alpha_{i+1}, ..., \alpha_j$ can be executed even if we had executed α_i (which removes something from S). Moreover, all commands check for the presence of some right and not for the absence. Therefore, the commands in σ_2 can still be executed and hence, this sequence will also result in a leaking state.

Repeating the above step for all revocation and deletion commands in σ, we get σ', a sequence which does not contain any revocation or deletion command and results in a leaking state if σ results in a leaking state. ∎

Lemma 3 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*
- *σ' does not contain any* AddObject, AddAccess, *or* ChangeDP *commands.*

Proof Let $\sigma = \alpha_1, \alpha_2, ..., \alpha_m$ and let α_i be the command that adds a new object o_1. We construct σ_1 by removing α_i and all other commands in σ that change

the object type of o_1. Since *ChangeOT* is the only command that depends on the existence of o_1, all the other commands can still be executed.

Since the presence or absence of o_1 does not affect the right leak for o (as per the definition of right leak problem above), if σ results in a leaking state, then σ_1 also results in a leaking state.

Repeating the above steps for all *AddObject* commands, we get a sequence σ_2 which does not contain any *AddObject* commands and results in a leaking state if σ results in a leaking state.

Using similar arguments, we can remove all *AddAccess* commands from σ_2. Moreover, since there is only one decision pointer in the system, *ChangeDP* commands can ignored.

So we now have a sequence σ' which does not contain any *AddObject, AddAccess* or *ChangeDP* commands and results in a leaking state if σ results in a leaking state. ∎

Corollary 4 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*
- *σ' does contains only* CreateRole, CreateOT, AddSubject, GrantRight, AddRoleBinding, *and* ChangeOT *commands.*

Lemma 5 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*
- *σ' does not contain more than one* CreateRole *and one* CreateOT *command.*

Proof Let $\sigma = \alpha_1, \alpha_2, ..., \alpha_m$. We will construct a new sequence, σ_1, as follows:

1. Let σ create i new roles, $r_1, r_2, ..., r_i$.
2. Remove all *CreateRole* commands in σ commands except the first one (only r_1 is added).
3. Replace all occurrences of r_j $(1 < j \leq i)$ with r_1 in all the remaining commands.

The roles have no pedigree (i.e., the creator of a role does not have any special rights over the role). Therefore, if some role r' can grant a right or allow a subject to bind to r_2, then r' can do the same to r_1. Since new roles have no pedigree, if some role, r, can execute a command α that affects r_j in some way, then r can execute α to affect r_1 in the same way. Therefore, for a given state S, if σ results in a leaking state, then σ_1 results in a leaking state and vice versa.

Similarly, we can remove all but the first *CreateOT* command from σ_1 and the resulting sequence, σ' results in a leaking state if σ results in a leaking state and σ' contains no more than one *CreateRole* and one *CreateOT* commands. ∎

Remark When the only *CreateRole* command is executed, it activates the $NewRole$ attribute. Similarly, the execution of *CreateOT* command activates the $NewOT$ attribute.

Lemma 6 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*
- *σ' does not contain more than $|T_R| + 1$* AddSubject *commands.*

Proof By lemma 2, we can assume that σ does not have any revocation or deletion commands.

Let σ contain n commands that add a new subject with the initial role r_1. *AddRoleBinding* is the only command other than *AddSubject* that directly addresses a subject.

Let $\wp(k, r_1)$ be the statement that if σ adds k new subjects with initial role r_1, then there exists a sequence of commands that results in a leaking state and adds only one new subject with initial role r_1.

Base Case: $k = 2$

When the first subject s_1 is created, $f_r(s_1) = \{r_1\}$. When the second subject s_2 is created, $f_r(s_2) \subseteq f_r(s_1)$, therefore, any new role binding added for s_2 can also be added for s_1. Let us construct a new sequence σ_1 by removing the *AddSubject*(r', r_1, s_2) command, replacing every occurrence of s_2 with s_1 in all *AddRoleBinding* commands and retaining all other commands.

Let $S \rightarrow_{\sigma} S'$ and $S \rightarrow_{\sigma_1} S_1$. By construction, s_1 has all the rights in S_1 that s_2 has in S' and neither s_1 nor s_2 were a part of S. We also know that S' is a leaking state (since σ results in a leaking state). Now, we will prove that S_1 is also a leaking state.

Case 1: The newly created subject s_2 caused the leak.

If s_2 gets the right ρ over o in S', then s_1 gets ρ over o in S_1. Therefore, if s_2 was the cause of the leak in S', then s_1 will cause a leak in S_1.

Case 2: The right leak was due to some other command in σ.

Since all the commands other than those that have s_2 as a parameter have been retained in σ_1, if commands other than the ones affecting s_2 caused the leak in S', then those same commands will cause a leak in S_1. Therefore, S_1 also results in a leaking state.

Hence, if σ adds two new subject with initial role r_1, then we can get a new sequence that also results in a leaking state but creates only one subject with initial role r_1. Therefore $\wp(2, r_1)$ is true.

Induction Step: Assume $\wp(k-1, r_1)$ is true ($k > 2$).

Let the k subjects added by σ with initial role be $s_1, s_2, ..., s_k$. We can construct a sequence of commands, σ_k which does not add the subject s_2 but still results in a leaking state using the construction mentioned above. Now, we have a sequence of commands that adds $k-1$ new subjects with initial role r_1 that results in a leaking state.

Since $\wp(k-1, r_1)$ is true, we can replace σ_k with a new sequence σ' that adds only one new subject with initial role r_1 and still results in a leaking state.

$$\wp(k-1, r_1) \Rightarrow \wp(k, r_1)$$

Therefore, by the principle of mathematical induction, $\wp(k, r_1)$ is true for all positive integer values of k. If $k = 0$, then we already have a sequence of commands that result in a leaking state and adds no more than one subject with initial role r_1.

Applying the above result for every role, we can construct a sequence of commands that add only one subject for every role in the state. By lemma 5, we can retain just one *CreateRole* command in σ, therefore the number of roles after the execution of all commands is at most one more than the number of roles in S. Therefore the new sequence will have at most $|T_R| + 1$ number of *AddSubject* commands.

Each of these $|T_R + 1|$ *AddSubject* commands activates a $NewSubject_r$ attribute. If the initial role of the new subject is r', then the corresponding *AddSubject* command activates the attribute $NewSubject_{r'}$.

Lemma 7 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*

- *σ' does not contain more than $N_G(S)$* GrantRight *commands, where $N_G(S) = 7(|T_R|+1)(|T|+2)(|T|+|R|+2)$.*

Proof We are interested only in those *GrantRight* commands that grant ρ, *ADDSUBJECT, CREATEROLE, CREATEOT, GRANTRIGHT, ADDROLEBINDING*, and *CHANGEOT* rights because:

1. By the definition of access right leak problem for the DCS-AC model, granting some role a right ρ' that is not one of the sixteen DCS-AC system rights or ρ (the right we are interested in) does not affect the access right leak problem.

2. By lemma 2, we are ignoring all revocation and deletion commands and by lemma 3, we can ignore all *AddObject, AddAccess,* and *ChangeDP* commands. Since we are not executing these commands, we need not execute *GrantRight* commands that allow roles to execute these commands.

Applying lemmas 2, 3, 5, and 6, we can construct a sequence of commands σ_1 from σ that results in a leaking state and contains:

- no revocation or deletion commands

- no *AddObject, AddAccess,* or *ChangeDP* commands

- no more than one *CreateRole* and one *CreateOT* command

- no more than $|T_R|+1$ *AddSubject* commands.

Let each *GrantRight*(r, r_1, t, ot, ρ_1) command activate the $HasRight_{r_1,ot,\rho_1,t}$ attribute. We know that ρ_1 has to be one of the seven rights identified above. Furthermore, if a *GrantRight* command, α_j, activates a attribute that was active in the inital state S or was activated by an earlier command α_i, we can ignore this command since there are no revocation or deletion commands. We construct a new sequence of commands σ' from σ_1 that does not contain any *GrantRight* command that activates an already active attribute. Since we are throwing away only those

commands that are meaningless in the context of access right leak, σ' will still result in a leaking state.

Therefore, the number of *GrantRight* commands in σ' is no more than the number of $HasRight_{r',ot',\rho',t'}$ attributes (active or inactive). We know that:

- number of possible values for r' is $\leq |T_R| + 1$
- number of possible values for ot' is $\leq |T| + 2$ [1] .
- number of possible values for ρ' is 7.
- number of possible values for t' is $\leq |T| + 2 + |R|$ (since $t' \in T \cup R$).

Let $S \rightarrow_{\sigma'} S'$. This means that the number of $HasRight$ attributes in S' is no more than $7(|T_R| + 1)(|T| + 2)(|T| + 2 + |R|)$.

Therefore, the number of *GrantRight* commands in σ' is no more than $7(|T_R| + 1)(|T| + 2)(|T| + |R| + 2)$. ∎

Lemma 8 *For a given state S, object o, and right ρ, if there exists a sequence of commands σ that result in a leaking state, then there exists a sequence of commands σ' such that:*

- *σ' also results in a leaking state and*
- *σ' does not contain more than $N_R(S)$* AddRoleBinding *commands, where $N_R(S) = (|T_R| + 1)(|SUB| + |T_R| + 1)$.*

Proof Applying lemmas 2, 3, 5, 6, and 7, we get a new sequence of commands σ_1 from σ which also results in a leaking state.

Now, each *AddRoleBinding*(r, s, r_1) command activates the $CanBind_{s,r}$attribute. We construct a new sequence of commands, σ' by removing all *AddRoleBinding*

[1] We can create one new role and one new object type. Since $T_R \subset T$, in the resulting state, $|T'|$ can be up to 2 more than $|T|$.

commands that activate a attribute that was already active in S or was activated by an earlier command in σ'.

Therefore, the number of *AddRoleBinding* commands in σ' is no more than the number of *CanBind* attributes in that instance of DCS-AC. Moreover, σ' does adds no more than one new role and no more than $|T_R| + 1$ new subjects (by lemmas 5 and 6 respectively).

Therefore, the number of *AddRoleBinding* commands in σ' is not more than $(|T_R| + 1)(|SUB| + |T_R| + 1)$.

Since we are removing only those *AddRoleBinding* commands from σ_1 that do not change the state of the system, σ' will also result in a leaking state. ∎

Definition An *enabler command* is a command that activates a currently inactive attribute. Therefore, for a state S, $Enabled(S)$ is the set of inactive attributes for which an enabler command exists.

Definition For a given state S, we say that a sequence of commands is monotonic if and only if the sequence contains only enabler and *ChangeOT* commands

Corollary 9 *For a given state S, right ρ, and object o, if there exists a sequence of commands σ that results in a leaking state, then there exists a sequence of commands σ' that is monotonic and results in a leaking state.*

Lemma 10 *For a given state S, the set of commands that activate the attributes in $Enabled(S)$ can be executed in any order and the resulting state will be the same.*

Proof Let $\lambda(n)$ be the proposition that for a sequence of n commands σ $(= \alpha_1, \alpha_2, ..., \alpha_n)$ that activate n enabled attributes, for any permutation π of the commands $\alpha_1, \alpha_2, ..., \alpha_n$, $(S \rightarrow_\sigma S_\sigma) \Rightarrow (S \rightarrow_{\sigma_\pi} S_\sigma)$ where σ_π is the permuted sequence of the commands in σ

BASE CASE: $n = 2$ (There are only two permutations, $1, 2$ and $2, 1$)

$\sigma = \alpha_1, \alpha_2$. Let $\sigma' = \alpha_2, \alpha_1$

Let α_1 activate the attribute χ_1 and α_2 activate χ_2. Let $S \rightarrow_{\sigma} S_{\sigma}$ and $S \rightarrow_{\sigma_1} S_{\sigma'}$.

S_{σ} is the state obtained as the result of adding the attributes χ_1 and χ_2 to S and $S_{\sigma'}$ is the result of adding the same two attributes to S.

$\therefore \quad S_{\sigma} = S_{\sigma'}$. Hence $\lambda(2)$ is true.

Induction Step:

Assume that $\lambda(k)$ is true for all $k(2 \leq k < n)$.

Let π be a permutation of the commands in σ and let $\sigma_{\pi}, = \alpha'_1, ..., \alpha'_n$, be the permuted sequence of commands. Furthermore, let $S \rightarrow_{\sigma_{\pi}} S_{\pi}$. $\lambda(n)$ is true if $S_{\pi} = S_{\sigma}$.

Case 1: $\alpha'_n = \alpha_n$

Let $\sigma_{n-1} = \alpha'_1, ...\alpha'_{n-1}$ and $S \rightarrow_{\sigma_{n-1}} S_{\pi}^{n-1}$. Since $\alpha'_n = \alpha_n$,

$$S_{\pi}^{n-1} \rightharpoonup_{\alpha_n} S_{\pi} \tag{5.1}$$

Let π' be a permutation of the commands $\alpha'_1, ..., \alpha'_{n-1}$. Let the new sequence formed by the permuted commands be σ'_{n-1}.

$$\lambda(n-1) \Rightarrow (S \rightarrow_{\sigma'_{n-1}} S_{\pi}^{n-1}) \tag{5.2}$$

Now if π' is a permutation such that $\pi'(\pi(\alpha_i)) = \alpha_i$, then

$$\sigma'_{n-1} = \alpha_1, \alpha_2, ..., \alpha_{n-1} \tag{5.3}$$

From 5.1, 5.2, and 5.3,

$$S \rightarrow_{\sigma} = S_{\pi} \tag{5.4}$$

But, $S \rightarrow_{\sigma} = S_{\sigma}$.

$\therefore \quad S_{\pi} = S_{\sigma}$.

Case 2: $\alpha'_n \neq \alpha_n$

Let $\pi(\alpha_n) = \alpha'_j$ $(1 \leq j < n)$.

$$\lambda(n-1) \Rightarrow S \rightarrow_{\sigma_1} S_\pi, \text{ where } \sigma_1 = \alpha'_1, ..., \alpha'_{j-1}, \alpha'_{n-1}, \alpha'_{j+1}, ..., \alpha'_{n-2}, \alpha'_j, \alpha'_n$$

$$\lambda(2) \Rightarrow S \rightarrow_{\sigma_2} S_\pi, \text{ where } \sigma_2 = \alpha'_1, ..., \alpha'_{j-1}, \alpha'_{n-1}, \alpha'_{j+1}, ..., \alpha'_{n-2}, \alpha'_n, \alpha'_j$$

Since $\alpha'_j = \alpha_n$,

$$\lambda(2) \Rightarrow S \rightarrow_{\sigma_2} S_\pi, \text{ where } \sigma_2 = \alpha'_1, ..., \alpha'_{j-1}, \alpha'_{n-1}, \alpha'_{j+1}, ..., \alpha'_{n-2}, \alpha'_n, \alpha_n$$

Since α_n is the last command, we can follow the same steps as in Case 1 and prove that $S_\pi = S_\sigma$.

$\therefore \forall\ k\ (2 \leq k < n)\ \lambda(k)\ \Rightarrow\ \lambda(n)$

By the principle of induction, since the base case is true and $(\forall k(2 \leq k < n)\lambda(k)) \Rightarrow \lambda(n)$, $\lambda(n)$ is true for all $n \geq 2$. ∎

Definition Let the number of attributes that are not active in a state S be $N(S)$. From the above lemmas, we know that

$$N(S) \leq 1 + 1 + (|T_R| + 1) + N_G(S) + N_R(S)$$

Theorem 11 *The access right leak problem for a given DCS-AC state can be decided in polynomial time.*

Proof Let the initial state be S^0, and let o and ρ be the object and right we are interested in. We will now provide a polynomial time algorithm to decide the access right leak problem for (S^0, o, ρ).

We know from corollary 9 that S^0 can lead to a leaking state if and and only if we find a monotonic sequence of commands.

Initialization Step

We create the set $InactiveAttributes$ which contains all attributes that are inactive in S. Initially, $InactiveAttributes = \{NewRole, NewOT\}$.

- $\forall\, r \in T_R$ add $NewSubject_r$ to $InactiveAttributes$.

- $\forall\, r \in T_R$, $\forall\, ot \in T$, $\forall\, \rho' \in R, \forall\, t \in (T \cup R)$
  if $HasRight_{r,ot,\rho',t} \notin Active(S)$ then add it to $InactiveAttributes$.

- $\forall\, s \in SUB$, $\forall\, r \notin f_r(s)$, add $CanBind_{s,r}$ to $InactiveAttributes$.

The above step can be completed in $O(N(S^0))$ time. Once we have identified the inactive attributes, we can proceed to the next step.

Saturation Step

We will now saturate the instance, i.e., activate every attribute that can be added till nothing more can be added. Starting from S^0, we execute all the enabled commands in the current state (S^i) to reach a new state (S^{i+1}) and repeat this step till no more attributes can be activated.

By lemma 10, we can execute the commands that enable the attributes in $Enabled(S^i)$ in any order. We execute them all taking care of the following book-keeping steps:

- If the $NewRole$ attribute is activated, then:
  1. For all subjects currently in the system, add a corresponding attribute ($CanBind_{s,r_{new}}$ where s is the subject and r_{new} is the new role created).
  2. For all object types, rights, and targets, add $HasRight_{r_{new},ot,\rho',t}$ to $InactiveAttrinutes$.

- If the $NewOT$ attribute is activated, then add corresponding $HasRight$ attributes.

- If a $NewSubject_{r'}$ attribute is activated, then for all roles r_1 currently in the system other than r', add $CanBind_{s_{new},r_1}$ to $InactiveAttributes$, where s_{new} is the new subject added.

- Remove all attributes that are activated from $InactiveAttributes$ once the enabling command is executed.

Repeat the above step till we reach a state S^s for which $Enabled(S^s) = \emptyset$. At this point the system is saturated and we cannot activate any new attribute.

Since there are at most $N(S^0)$ attributes, this step takes $O(N(S^0)^2)$ time. We now proceed to the next step.

Type Checking Step

Let $S^s = (A', R', T', T'_R, SUB', OBJ', f'_o, f'_r)$. Now, we can tackle the *ChangeOT* command as follows: We first identify the set of *leaking types*, LT and then check if o can be changed to any of these types.

$$LT = \{ot | ot \in T' \wedge \Phi_{S^s,\rho,ot} - \Phi_{S^0,\rho,f_o(o)} \neq \emptyset\}$$

We can identify the elements of the set LT in $O(|T| \cdot |SUB| \cdot |T_R| \cdot |R|)$ time.

Once we have identified the elements of LT, we can check if the type of o can be changed to any of these types by creating a *Typechange* graph, $G = (V, E)$ where G is a directed graph such that $V = T' - T'_R$ and

$$E = \{(ot_1, ot_2) | ot_1, ot_2 \in T' \wedge \exists r \in \ T'_R \text{ such that } (CHANGEOT, ot_1, dp) \in A'[r, ot_2]\}$$

That is, the nodes in G correspond to object types in S^s and there is an edge from ot_1 to ot_2 if some role can change the object type of an object from ot_1 to ot_2.

Now, the problem of deciding if the object type of o can be changed to one of the *leaking types* is reduced to a graph reachability problem which can be solved by performing a depth first search starting from $f_o(o)$ (the object type of o in the initial state) and checking if the current node is in LT. If one of the types in LT can be reached from $f_o(o)$, then there is a sequence of commands that can lead to a leaking state else there does not exist a sequence of commands that result in a leaking state.

The graph can be constructed in $O(|T|^2)$ time and the graph traversal also takes the same amount of time. The type checking step takes $O(|T| \cdot |SUB| \cdot |T_R| \cdot |R|)$ time.

Of the three steps mentioned above, the saturation step takes the most amount of time and hence the ARL problem can be decided in $O(N(S)^2)$ time. ∎

5.5 Summary

In this chapter, we restricted the DCS-AC model and saw how to decide in polynomial time whether a given state could result in a leaking state by executing an arbitrary sequence of commands if voting is disallowed. We assumed monotonicity and proved that there is a polynomial upper bound on the number of meaningful commands (i.e., commands that when executed result in a change of state). Using this, we constructed a directed state graph where each node of a graph corresponds to a state and two nodes are connected if there is a command which when executed on the starting state results in the ending state.

We then identified those states which result in a leak and if there is a path from the node corresponding to the starting state to a node corresponding to one of the leaking states, then the system can leak the right. The commands that will result in a leaking state is the sequence of commands that correspond to the edges along such a path. Hence the access right leak problem for this restricted version of the DCS-AC is decidable in polynomial time. In the next chapter, we will see how to solve this problem if voting is allowed.

CHAPTER 6
TRUST AND VOTING

6.1 Introduction

In this chapter we will analyze the safety of the DCS-AC model where subjects are allowed to vote. Since we cannot deterministically predict how the subjects will vote we will approach the problem from a different angle. We assign a numerical value, *trust-level*, to each subject which represents the amount of resources a malicious entity needs to expend in order to make the subject vote against the best interests of the group. We will identify three simple approaches on how subjects can be "turned" and analyze the access right leak problem based on these approaches.

6.2 Trust and Trust-Worthiness

One of the key questions in any vote is "How much can we trust the voters to make the right decision?". There are many aspects involved in *trust*, some of which are identified below:

- understanding of and belief in the mission of the group
- sound decision-making process
- rational behavior
- susceptibility to blackmail or bribery
- protection of identity

Different groups give different weight-ages to these and other aspects of trust. Based on these, we assign a *trust level* to each subject s denoted by $\theta(s)$, which is the amount of resources required to turn the subject s. Our main concern here is

the susceptibility of the subjects to make a wrong decision. By wrong, we mean a decision that is not in the best interests of the group. The subject might decide to do the wrong thing by intention (needless to say, our trust level in such a person would be low). We assume that we know the trust level of all subjects in the group and that this trust level is constant. For the sake of simplicity, we will assume that each subject in the system has a unique trust level.

$$\theta(s_1) = \theta(s_2) \Leftrightarrow s_1 = s_2$$

Note that we do not talk about how the trust levels are assigned as this may vary from group to group depending on the group proirities. We should consider all the factors mentioned above and other aspects that the group might feel important. We could use a reputation-based trust management scheme like the model defined by Shmatikov and Talcott [57] or a feedback-based scheme similar to the one used in eBay [58]. In our safety analysis, we will assume that each subject has a unique trust-level that is assigned outside of the DCS-AC and is correct.

6.2.1 Approaching Trust

There are many ways of looking at trust and susceptibility. The key idea in here is to look at trust level as the amount of resources needed to make a particular subject or group of subjects vote in a manner that is against the best interests of the group. With this in mind, we will now look at three different approaches to analyzing susceptibility through trust levels as defined here.

Ad-Approach

In the first case, if someone can make a subject s_1 make a decision in a certain way, then that person can make any other subject s_2 make the same decision in the same way with no additional effort if $\theta(s_1) \geq \theta(s_2)$. We will call this the *Advertisement approach* (ad-approach) since we can look at this as paying for an advertisement which all the voters will see. If the ad is good enough to convince s_1,

then s_2 will also be convinced. We will use $\theta_A(s)$ to denote the trust in a subject s under the ad-approach. It must be noted that in this approach, one "ad" will cover all decisions required.

Pay-As-You-Go Approach

It is possible to bribe or cheat the voters to turn one decision at a time. This means that more resources are required to convince the same voter if he/she gets to participate in another decision down the line. We will call this the Pay-as-you-go approach (P-approach) and use $\theta_P(s)$ to denote the trust in a subject s under this approach.

Honest Politician Approach

In the words of former U.S. senator Simon Cameron, "an honest politician is one who when bought, stays bought". The key idea behind this approach is the assumption that all the subjects are "honest politicians". In this case, $\theta(s)$ is the amount of resources that someone has to expend to make s_1 make a certain decision (that person has to expend $\theta(s_2)$ additional resources to "turn" s_2). In addition, once bought, the subject stays bought and no additional resources are required if the subject gets to vote on another decision later. We will call this the *Honest Politician approach* (H-approach). We will use $\theta_H(s)$ to denote the trust in a subject s under the H-approach.

Hybrid Approach

The approaches mentioned above are very simplistic and in reality, a mixture of these approaches is more likely to be used. These three approaches provide a starting point for discussion and are more useful in introducing the ideas discussed here. We hope to extend this to model real-life approaches in the near future. We will concentrate only on the first three approaches here.

6.2.2 Decision Templates

In order to clarify how decision templates work, let us look at a simple decision template. Here, all votes are weighted equally and parameters like quorum, voting period, percentage of votes required for a *yes* vote are all part of the template. Each such voting template is represented by the tuple (V_R, k, q, t_d, y_d) where,

- V_R is a set of roles that can participate in the vote,
- k is a number between 0 and 1 which represents the ratio between the minimum number of *yes* votes required to pass and the total number of participating voters,
- q is another number between 0 and 1 which represents the quorum (i.e., ratio of minimum number of voters required to participate and the total number of voters),
- t_d is the time duration over which the vote is active (example: 2 days), and,
- y_d is the default decision which is returned if the deadline has passed and the quorum has not yet been reached.

In addition, the set of decision pointers, DT, also contains an *Always-Yes* template (d_{yes}) which always returns *yes*. The reason for this is that many decisions do not require a vote. An example of the template is $(\{Faculty, Staff\}, 0.5, 0.8, 2 \text{ days}, N)$. In this template, all subjects who can bind to the roles Faculty and Staff can participate in the vote which requires a simple majority with at least 80% voter turnout. If after 2 days, the quorum is not reached, a result of *No* is returned. If after two days, at least 80% of the voters have cast their votes, then a *Yes* decision is returned if at least half the votes cast are *yes* votes, if not a result *No* is returned.

In our model, each subject gets to cast one vote even if he/she can bind to more than one role in V_R. A discussion of the actual working of the voting mechanism is beyond the scope of this work and we will restrict ourselves to a brief description of the life-cycle of a vote. When a group decision is called for, we instantiate a vote v based on the template. $V = (E_v, Vote_v, Voted_v, T_v, k_v, q_v, default)$ where:

- E_v is the set of all eligible voters
- $Vote_v$ is a mapping between each subject and his/her vote in the decision. The possible values are D (defer), Y (Yes), N (No), and, A (abstain)
- $Voted_v$ is the set of all eligible voters who have cast a valid vote (Y, N, or, A)
- T_v is the deadline for the vote which is obtained by adding t_d to the time of vote initialization.
- k_v is the value of k in the template
- q_v is the value of q in the template
- $default$ is the value y_d of the template

$$E_v = \{s \in SUB | V_R \cap f_r(s) \neq \emptyset\}$$

$$Vote_v : E_v \rightarrow \{Y, N, A, D\}$$

$$Voted_v = \{s \in E_v | Vote_v(s) \in \{Y, N, A\}\ \}$$

Initially, since nobody has voted yet,

$$\forall s \in E_v,\ Vote_v(s) = D$$

When an eligible voter, s, casts a valid vote ($Y, N,$ or, A), his/her vote is updated in $Vote_v$ and s is added to $Voted_v$. When the deadline T_v is reached, all new votes are rejected and a decision is reached.

We identify the following sets:

$$Y_v = \{s \in E_v | Vote_v(s) = Y\}$$

$$N_v = \{s \in E_v | Vote_v(s) = N\}$$

$$A_v = \{s \in E_v | Vote_v(s) = A\}$$

We first check for the quorum,

$$\frac{|Voted_v|}{|E_v|} \geq q_v$$

if this is false, we return the default decision. If every subject who voted has abstained, ($|Y_v| + |N_v| = 0$), we return the default decision. Next, we check if enough yes votes have been cast.

$$\frac{|Y_v|}{|Y_v| + |N_v|} \geq k$$

if this is true, we return yes and we return no otherwise.

6.3 ARL Problem And States

Since excecution of certain commands is dependent on the subjects who are allowed to vote, we have to factor our trust in the subjects while deciding the safety of the system. With this in mind, we redefine the access right leak problem for a DCS-AC system with voting as follows:

Definition For a given state S^0, object o, right ρ, and, budget Υ, the access right leak problem (ARL) is a decision problem, does there exist a sequence of commands in S^0 with trust level less than Υ that results in a leaking state.

$$ARL \equiv \{< S^0, o, \rho, \Upsilon > | \exists \sigma \text{ such that } S^0 \rightarrow_\sigma S^t \text{ and}$$

$$\Phi_{S^t,\rho,f'_o(o)} - \Phi_{S^0,\rho,f_o(o)} \neq \emptyset \text{ and} \Psi(\sigma) \leq \Upsilon\}$$

where $\Psi(\sigma)$ is our trust in the sequence of commands σ by applying one of the three approaches.

In other words, can a malicious party, with a given budget, corrupt enough voters to enable an unauthorized subject access to an object.

Let $N_v(S^0)$ denote the number of commands that can be executed on the initial state S^0 and all its descendent states. We are only concerned with the commands *CreateRole*, *CreateOT*, *GrantRight*, *AddSubject*, *AddRoleBinding*, and *ChangeOT* (this is due to Corollary 4). As proven in the lemmas 3, 5, 6, 7, and 8, the total number of such commands is bounded.

$$\begin{aligned} N_v(S^0) &\leq 1 + 1 + (|T_R| + 1) + N_G(S^0) + N_R(S^0) + (|T - T_R| + 1) \\ &\leq 4 + N_G(S^0) + N_R(S^0) + |T| \end{aligned}$$

Definition For a given DCS-AC system with initial state S^0, we define the set of all possible reachable states, $\widehat{S^0}$ as follows:

$$\widehat{S^0} = \{S | (S = S^0) \vee (\exists \alpha \text{ such that } S' \rightharpoonup_\alpha S \wedge S' \in \widehat{S^0})\}$$

The size of this set is dependent on the number of commands that can be executed on the initial state and its descendants. Each state can be represented by a $N_v(S^0)$-bit number where each bit corresponds to a particular attribute. The bit is 1 if the corresponding attribute is active in that state and 0 otherwise.

$$|\widehat{S^0}| \leq 2^{N_v(S^0)}$$

6.4 Trust Analysis of DCS-AC

Let us now prove the tractability of the ARL problem for a DCS-AC system. As mentioned earlier, we will use the trust-worthiness of subjects to decide on the trust-worthiness of the system.

6.4.1 State Graph

We will now construct a directed graph $G_{S^0} = (V, E)$ which corresponds to a DCS-AC system with the initial state S^0 (We will call this graph the *State Graph* of S^0). The set of vertexes is $V = \widehat{S^0}$ and there is an edge from vertex S^i to S^j

labeled (α, r) if S^j is the result when a subject bound to the role r executes the command α in S^i.

Since each state has a unique set of properties, there cannot be more states than the total number of properties in any given path.

$$|V| = |\widehat{S^0}| \leq 2^{N_v(S^0)}$$

In any given state, we cannot execute more than $N_v(S^0)$ commands and each of these commands can be executed by at most $|T_R|$ roles.

$$|E| \leq |V| \cdot |T_R| \cdot N_v(S^0) \leq |T_R| \cdot N_v(S^0) \cdot 2^{N_v(S^0)}$$

If we use an adjacency list, we can construct the graph in $O(|E|)$. Therefore, the graph can be constructed in $O(|T_R| \cdot N_v(S^0) \cdot 2^{N_v(S^0)})$ time.

Let S_L denote the set of all leaking states.

$$S_L = \{S^i \in \widehat{S^0} | \Phi_{S^i,ot,\rho} - \Phi_{S^0,f_o(o),\rho} \neq \emptyset\}$$

As mentioned in the previous chapter, we can identify the elements of S_L in $O(|V| \cdot |T| \cdot |T_R| \cdot |SUB| \cdot |R|)$ time.

Before looking at the three approaches to trust from the DCS-AC point of view, let us define a few useful sets. For a decision $d = (V_R, k, q, y_d, t_d)$, let $W(d)$ be the set of $\tau(d)$ weakest (in terms of susceptibility) voters. These voters are the most likely to vote against the best interests of the group.

$$W(d) = \{s|\ s \in\ V(d)\ \wedge \nexists s_1, s_2, ..., s_{\tau(d)}\ \in\ V(d) \text{ such that}$$

$$i \neq j \Leftrightarrow s_i \neq s_j\ \wedge\ \forall\ i \in\ [1, \tau(d)]\ \theta(s_i) < \theta(s)\}$$

For a command α, let $W_c(\alpha)$ denote the set of voters that are most likely to vote against the best interests of the group.

$$W_c(\alpha) = W(d)$$

where d is the decision that guards the command. Moreover, some subject has to issue the command in order to execute it. Therefore, our trust in a command has to include our trust in the role that can issue the command. Let $\gamma(r)$ denote our trust in a role r. Our trust in a role is the same as our trust in the least trustworthy subject that can bind to that role.

$$\gamma(r) = min\{\theta(s)|s \in SUB \text{ and } r \in f_r(s)\}$$

Needless to say, our trust in the decision pointer d_{yes} is 0.

6.4.2 Ad-Approach

Let $\Theta_A(d)$ denote our trust level in a decision. In ad-approach, $\Theta_A(d)$, is the maximum trust level of all the voters in $W(d)$.

$$\Theta_A(d) = max\ \{\theta_A(s)|\ s \in\ W(d)\}$$

Let d be the decision template that guards a command α. If the right that guards α is enabled, then we can trust α only as far as we can trust d. Let $\psi_A(\alpha)$ denote our trust level in a command α. Here, we are looking at the command by itself with no prior "history". There may be more than one role that can issue the command. Let r_l be the least trust worthy of all the roles that can execute the command.

$$\psi_A(\alpha) = max(\Theta_A(d), \gamma(r_l))$$

Now, for a given sequence of commands $\sigma(= \alpha_1, \alpha_2, ..., \alpha_n)$, let $\Psi_A(\sigma)$ denote the trust level in σ using the ad-approach.

$$\Psi_A(\sigma) = \ \psi_A(\alpha_i) \text{ such that } \ \forall\ j \in \ [1, n], \psi_A(\alpha_j) \leq \psi_A(\alpha_i)$$

In order to determine the minimum cost required to ensure a leaking state, we have to find a path from S^0 to S_L that has the least Ψ_A value. We achieve this by doing a modified depth first traversal on the graph.

Function DFT(v)

//The array $visited[]$ is global and initially set to false

//The value $MinCost_A(S^0)$ is initially ∞

//The stack is initially empty.

1. $visited[v] = true$

2. if $v = S_L$ then
   (a) let σ be the sequence of commands in the stack
   (b) $c = \Psi_A(sigma)$
   (c) if ($c < MinCost_A$) then $MinCost_A = c$
   (d) return

3. for each u such that $(v, u) \in E$
   (a) if $visited[u] = false$ then
      i. push α into stack where α is the label of the edge (v, u)
      ii. DFT(u)
      iii. pop α from stack

We invoke this function by calling DFT(S^0).

$$< S^0, 0, \rho, \Upsilon >\in ARL \ \Leftrightarrow \ \Upsilon \geq MinCost_A(S^0)$$

The above depth-first traversal can be completed in $O(|E|)$ time. Hence the ARL problem using the Ad-approach can be solved in $O(|T_R| \cdot 2^{N_v(S^0)})$ time.

6.4.3 Pay-As-You-Go Approach

Let $\Theta_P(d)$ denote our trust level in a decision. In this approach, $\Theta_P(d)$, is the maximum trust level of all the voters in $W(d)$.

$$\Theta_P(d) = \sum_{s \in W(d)} \theta_P(s)$$

Let d be the decision template that guards a command α. If the right that guards α is enabled, then we can trust α only as far as we can trust d. Let $\psi_P(\alpha)$ denote our trust level in a command α. Here, we are looking at the command by itself with no prior "history". If r_l is the least trust worthy role that can issue the command α,

$$\psi_P(\alpha) = \Theta_P(d) + \gamma(r_l)$$

Now, for a given sequence of commands $\sigma(= \alpha_1, \alpha_2, ..., \alpha_n)$, let $\Psi_P(\sigma)$ denote the trust level in σ using the this approach.

$$\Psi_P(\sigma) = \sum_{\alpha_i \in \sigma} \psi_P(\alpha_i)$$

In order to determine the minimum cost required to ensure a leaking state, we have to find a path from S^0 to S_L that has the least Ψ_P value. We achieve this by using the single-source shortest path algorithm starting from S^0. For each edge (S^i, S^j), the edge weight is $\psi_P(\alpha)$ where α is the label of the edge.

Let the minimum cost to reach S_L be $MinCost_P(S^0)$.

$$< S^0, 0, \rho, \Upsilon > \in ARL \;\Leftrightarrow\; \Upsilon \geq MinCost_P(S^0)$$

Here again, the single source shortest path takes $O(|E|)$ time (since we are using adjacency lists) and hence the ARL problem can be solved using the P-Approach in $O(|T_R| \cdot 2^{N_v(S^0)})$ time.

6.4.4 Honest Politicial Approach

In the Honest Political approach, a subject once "bought" stays bought. There are two problems with applying an algorithm similar to the ones used for the other approaches. Consider the two decisions d_1 and d_1 and two subject s_1 and s_2 such that: (i) s_1 and s_2 are both valid voters for d_1 and s_2 is a valid voter for d_2, and (ii) s_1 is among the potentially weak voters in d_1 while s_2 is not.

$$s_1, s_2 \in Voters(d_1) \text{ and } s_2 \in Voters(d_2)$$

$$s_1 \in W(d_1) \text{ and } s_2 \notin W(d_1)$$

1. If $s_2 \in W(d_2)$, i.e., s_2 is among the potentially weak voters in d_2, then instead of spending resources on both s_1 and s_2, a malicious entity can "buy" only s_2 and still get both the decisions thus saving on the resources required to buy s_1.

2. If $s_2 \notin W(d_2)$ but there exists a subject $s_3 \in W(d_2)$ such that $\theta_H(s_1) + \theta_H(s_3) > \theta_H(s_2)$, then it would be cheaper to buy s_2 instead of both s_1 and s_3.

The above two scenarios can be extended to any number of subjects and any number of decisions such that replacing a single subject can lead to a cascading effect on other decisions if the subject being replaced is among the weak voters in the other decisions.

As we can see, the main problem is with voters who can participate in more than one decision in a sequence of decisions. Our solution to this problem is very simple. If a subject s_1 with a trust level of $\theta_H(s_1)$ participates in $n(< |\sigma|)$ decisions in a sequence of commands σ, then let $\theta_H^\sigma(s_1)$ denote the cost per decision in the sequence.

$$\theta_H^\sigma(s_1) = \frac{\theta_H(s_1)}{n}$$

Let the set $V(\sigma)$ denote the set of all voters in all the decisions in σ.

$$V(\sigma) = \cup_{\alpha_i \in \sigma} Voters(d_i) \text{ where } d_i \text{is the decision that guards } \alpha_i$$

Let $W_s(\sigma)$ denote the minimal set of voters required to get all the decisions in σ. Let us now look at an algorithm to find this set.

Function $W_s(\sigma)$

//Returns the minimal set of voters

//Initially W_s is an empty set

1. sort all subjects in $V(\sigma)$ according to their θ_H^σ
2. For each α_i in σ
   (a) Let d_i denote the decision guarding α_i
   (b) Let *count* be the number of voters in α_i already bought. $count = |Voters(d_i) \cap W_s|$
   //We now have to find $\tau(d_i) - count$ eligible voters
   (c) while $count < \tau(d_i)$
      i. Find $s \in Voters(d_i) - W_s$ with smallest θ_H^σ
      ii. Add s to W_s
      iii. $count = count + 1$
3. Return W_s

Now, let $\Psi_H(\sigma)$ denote the trust level in σ using this approach.

$$\Psi_H(\sigma) = \sum_{s \in W_s(\sigma)} \theta_H(s)$$

In order to determine the minimum cost required to ensure a leaking state, we have to find a path from S^0 to S_L that has the least Ψ_H value. We achieve this by using the DFT algorithm given above (we calculate Ψ_H and $MinCost_H$ instead of Ψ_A and $MinCost_A$.

$$< S^0, 0, \rho, \Upsilon > \in ARL \;\Leftrightarrow\; \Upsilon \geq MinCost_H(S^0)$$

We know that the length of the longest possible sequence is $N_v(S^0)$. We also know that the maximum number of voters that can participate in a decision is $|SUB|$. Hence, we can find the Ψ_H for any sequence of commands in $O(|SUB| \cdot N_v(S^0))$ time. Therefore, we can decide the ARL problem using the H-approach in $O(|SUB| \cdot N_v(S^0) \cdot |T_R| \cdot 2^{N_v(S^0)})$ time.

6.5 Summary

In this chapter we looked at the safety analysis of the DCS-AC model with voting. We assumed that each subject has a numerical *trust-level* and introduced three approaches to trust and suseptibility namely, Ad, Honest Politician and Pay-As-You-Go. We then saw how to determine the trust in a vote, a command and a sequence of commands based on each of these three approaches. With this as building blocks, we analyzed the safety of the DCS-AC model with a modified access rights leak problem.

The key idea in the Ad-approach is that if a malicious entity can convince a subject, s, to vote in a manner that is not in the best interests of the group, then that entity does not need any additional resources to convince another subject whose trust level is less than that of s. In the Pay-As-You-Go approach, each subject has a specific amount of resources required to make him/her vote in a manner that is not in the best interests of the group and the malicious entity has to expend additional resources for every other subjects and also for the same subject if he/she is required to vote again in another decision.

The most complex of the three approaches is the Honest Politician approach, in which, a subject is "bought" for the entire set of decisions. This complicates the safety analysis since a malicious entity is better off spending resources on a more expensive subject who can participate in many decisions as opposed to a large number of less trustworthy subject each of whom can participate in one decision.

In order to solve this problem, we looked at cost per decision while deciding the least expensive set of voters.

In each of these three approaches, we construct a directed graph which we call state graph and use graph traversal algorithms to solve the access right leak problem. There is a combinatorial explosion of the number of states and we proved the decidability of the ARL problem in each of these three approaches by providing exponential time algorithms.

CHAPTER 7
NEXT STEPS

The model as presented is powerful, flexible and scalable. However, more work needs to be done in order to make this model model real-life policies. Some of these are enhancing the commands in the model, extending the model itself to allow subject-to-object privilege definitions, incorporating a role hierarchy and time-limited access rights. As mentioned earlier, a major feature of the DCS model is the use of decision templates which can protect against single malicious user attacks. Further work needs to be done in analyzing multi-user collusion attacks.

7.1 Enhancing DCS Commands

There are some additions that can be done to improve the model. The commands as described here are all mono-operational (i.e., perform only one primitive operation). In many cases, the subject creating a new object has some special rights with respect to that object, i.e., the creator has the right to read/write the object or grant rights to others. While someone has the *GrantRight* privilege for objects of that type, this issue is more critical for creation of new roles and object types. If someone creates a new role or object type, the usability of this new role or object type is severely restricted if no one has the right to allow subjects/objects to bind to that role/object type. Therefore an additional operation needs to be performed that allows the creator's role some basic rights.

The safety analysis needs to be revisited since the roles and object types have pedigree and we will have to factor this into the analysis. However, the resulting system would be more usable.

7.2 Special Relations

We would like to be able to specify subject-to-object level access rights, for example, even though all programmers have the right to read any code, only the programmer who wrote a particular class has the right to modify it. We can implement this by creating a new object type or a new role for each class/programmer. This is inefficient and a simpler way is to use *Special Relationships*. These are a mapping from (subject,object) pairs to roles. ($f_s : SUB \times OBJ \rightarrow T_R$). We can define a role *ClassWriter* and specify $f_s(programmer, class) = \{ClassWriter\}$ for each class. Now, each time an access request is made, we should check for any special relations between the subject and the object before checking for the role and object type (the reason being, special relations are usually more "powerful" than regular roles for that particular object). This clearly changes the definition of all the commands and the safety analysis but provides a very useful way to specify finer-grained access control.

7.3 Role Hierarchy

One of the major differences between the roles as used in the DCS model and RBAC is *role hierarchy*. We can implement role hierarchy by maintaining a partial ordering of the roles in the form of a directed graph, $G_R = (V, E)$ where $V \subseteq T_R$ and $(r_1, r_2) \in E$ if and only if r_1 inherits all the rights from r_2. When an access request is made, we first check for special relations, failing which the rights of the user's role is checked. If this fails, a breadth first search is performed on G_R with the mode corresponding to the user's role as the starting point. At each node visited, the corresponding role is checked for the right to perform the requested access. We use breadth first search instead of depth first search since roles higher up on the hierarchy are more "powerful" than those further down the graph. It might be useful to constrain the role ordering to be acyclic for security purposes. This can be ensured if the role hierarchy can be specified only when the role is

defined and cannot be changed. This is too rigid and can hurt the utility of the model. Therefore, we can allow the hierarchy to be changed anytime but the graph could be checked for cycles before committing the change. Since the role graph is not likely to change frequently, this is an acceptable overhead on performance. The addition of role hierarchy increases the overhead while searching for all the users in a role[1] .

7.4 Use-Cases and Scenarios

One of the most important steps in proving the usability of the model is to identify various scenarios and show how this model can be used there. For example, we can model the CISE department in terms of subjects, roles, objects, etc. and show how we can use the DCS model (Only Ph.D. candidates can register for doctoral research credits and a Ph.D. student can bind to the Ph.D. candidate role if his/her supervisory committee chair nominates him/her and all the members of the committee agree). One other scenario we can consider is the operations of a large organization like IEEE. The idea is to show how the DCS model can be used in these scenarios, starting from bootstrapping the system to getting it fully functional.

7.5 Multi-User Collusion Attacks

The use of decision templates makes the DCS model very powerful and helps minimize the role of the local domain system administrators. This also opens up an interesting area of research, multi-user collusion attacks. For a given state of the system, if a group of subjects want to tip a vote (with a given decision template, dp_1) in their favor, and they try to do so by allowing some of their *buddies* to bind to some roles which participate in that vote, how many *AddRoleBinding*

[1] We may have to find the list of all users who can bind to a role for voting and notification purposes.

commands do they need? If the *AddRoleBinding* commands require another vote (say, dp_2), they will have to be able to carry those votes in order to get a favorable result for dp_1. The question is, for any given dp_1 and dp_2, the minimum number of "conspirators" required to win the vote dp_1 is the minimum number required to win dp_2. A more complex case is when you need to control two votes, dp_2 and dp_3 to win dp_1.

7.6 Time-Limited Access

Another useful addition is $Time - Limited\ Access$ (TLA)[2] . Instead of returning a simple yes/no answer, the decision pointer could return the time for which the access is valid. If the access is denied, then it returns a zero else it returns the time for which the access is valid. During that time, the user is granted access to the object without starting the decision-making mechanism. This is very useful for "proxies" or substitutes. For example, if the user *Alice* who can bind to the role $Instructor$ is leaving town for a week, she can allow *Bob* (bound to role TA) to temporarily bind to the role $Instructor$. We can also create a new role $sub - Instructor$ with a subset of the rights of $Instructor$. For example, *Bob* can not assign grades to the students. In order to implement TLA, we need a separate data structure that stores the time period of validity. The TLA structure should either periodically remove expired entries or check for validity each time an access is requested. This again might increase the turn-around time for an access request but on the other hand, it can be very useful since it can help avoid frequent calls to time consuming decision pointers.

[2] TLA could also stand for Three-Letter Acronym

CHAPTER 8
CONCLUSION

The access control model described here is designed for the Distributed Conferencing System version 2 (DCS v.2). However, this model can be used in any distributed system that requires highly available access control services and user driven decision making capabilities. We have formally specified the model and analyzed its various advantages with respect to existing models. We have also provided algorithms to map a state in a DCS system to an equivalent HRU state and *vice versa* and proved that the access right leakage problem is decidable in NP time. The software project example illustrates the usefulness of this mode. As seen in chapter four, this model satisfies many interesting properties of access control systems and supports user-defined voting templates. The presence of decision templates allows the members of the group to decide on their own group policies and vote on important decisions. This democratic approach ensures autonomy for the groups and provides greater flexibility in terms of access control.

We have shown that the model is provably safe in terms of the access rights leak problem. Since some of the access control decisions are made through voting, the security of the system depends on our trust in the voters. This leads us to the three approaches to trust that are mentioned here. The Ad-Approach is simplistic and the analysis is very straight forward. The Pay-As-You-Go approach is slightly more complicated and the Honest-Politician approach is the most difficult of the three to analyze. In all three approaches, we reduced the problem to some form of graph reachability and have provided algorithmic solutions to the same.

This type of analysis can help us prepare against multi-user collusion attacks and is heavily dependent on accurately assigning trust-levels to all the subjects.

In reality, a hybrid of these three approaches is likely to be used and this work provides a starting point for further work in this area. In conclusion, the model presented here is scalable, useful, provably secure, and very flexible.

REFERENCES

[1] B. W. Lampson, "Protection," in *Proceedings of 5th Princeton Symposium on Information Sciences and Systems*. Princeton, 1971, pp. 437–443.

[2] G. Scott Graham and Peter J. Denning, "Protection–principles and practice," in *Proceedings of Spring Joint Computer Conference*. AFIPS, 1972, pp. 417–429.

[3] Steven J. Greenwald, "A new security policy for distributed resource management and access control," in *Proceedings of the 1996 Workshop on New Security Paradigms*. 1996, pp. 74–86, ACM Press.

[4] Markus Lorch and Seth Proctor and Rebekah Lepro and Dennis Kafura and Sumit Shah, "First experiences using XACML for access control in distributed systems," in *Proceedings of 2003 ACM Workshop on XML security*. 2003, pp. 25–37, ACM Press.

[5] Ian Foster and Carl Kesselman and Gene Tsudik and Steven Tuecke, "A security architecture for computational grids," in *Proceedings of the 5th ACM Conference on Computer and communications security*. 1998, pp. 83–92, ACM Press.

[6] M. Webb and D.L. Wilson R.E.Newman-Wolfe and C.L.Ramirez and H. Pelimuhandiram and M. Montes, "A brief overview of the dcs distributed conferencing system," in *Proceedings of Summer Usenix Conference*, Nashville, TN, 1991, USENIX, pp. 437–452.

[7] András Belokosztolszki and Ken Moody, "Meta-policies for distributed role-based access control systems," in *Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002, pp. 106–115.

[8] Anek Vorapanya, *Large-Scale Distributed Services*, Ph.D. dissertation, University of Florida, 2000.

[9] James Gosling and Bill Joy and Guy L. Steele Jr. and Gilad Bracha, *The Java Language Specification*, Addison-Wesley Professional, Boston, MA, third edition, 2005.

[10] Herbert Schildt, *Java: The Complete Reference, J2SE 5 Edition*, McGraw-Hill Osborne Media, 2004.

[11] Ashish Bhalani, "Implementation of conference control service in distributed conferencing system," M.S. thesis, University of Florida, 2002.

[12] Swati Shukla, "Notification services in a distributed conferencing system," M.S. thesis, University of Florida, 2000.

[13] Amit V. Date, "Implementation of distributed database and reliable multicast for distributed conferencing system version 2," M.S. thesis, University of Florida, 2001.

[14] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison-Wesley, Boston, MA, 1994.

[15] Maydene Fisher and Jon Ellis and Jonathan Bruce, *JDBC API Tutorial and Reference*, Addison-Wesley Professional, Boston, MA, third edition, 2003.

[16] Luke Welling and Laura Thomson, *MySQL Tutorial 1/e*, MySQL Press, Uppsala, Sweden, 2004.

[17] Korry Douglas and Susan Douglas, *PostgreSQL*, Sams, Indianapolis, IN, 2003.

[18] Aparna Kakarparti, "Decision support service for distributed conferencing system," M.S. thesis, University of Florida, 2002.

[19] Vijay Manian, "Access control for distributed conferencing system," M.S. thesis, University of Florida, 2002.

[20] Ravichandran Aringunram, "Secure communication services gambetta trustfor distributed conferencing system," M.S. thesis, University of Florida, 2002.

[21] Philip S. Yeager, "A distributed file system for distributed conferencing system," M.S. thesis, University of Florida, 2003.

[22] Vijaya Rangadham Komaragiri, "Mother of all concurrent editors version 2.0," M.S. thesis, University of Florida, 2004.

[23] Charles P. Pfleeger, *Security in Computing*, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.

[24] D. Bell and L. LaPadula, "Computer security model: Unified exposition and multics interpretation," Tech. Rep. MTR-2997, MITRE Corporation, 1976.

[25] K. J. Biba, "Integrity consideration for secure computer systems," Tech. Rep. MTR-3153, MITRE Corporation, 1977.

[26] E. Ellmer and G. Pernul and G. Quirchmayr and A. Tjoa, "Access control for cooperative environments," in *Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work (CSCW'94).*, 1994.

[27] Michael A. Harrison and Walter L. Ruzzo and Jeffrey D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, August 1976.

[28] R. S. Sandhu, "The typed access matrix model," in *Proceedings of the IEEE Symposium on Security and Privacy*, Washington, DC, 1992, pp. 122–136, IEEE Computer Society.

[29] Ravi S. Sandhu and Gurpreet S. Suri, "Implementation considerations for the typed access matrix in a distributed environment," in *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, 1992, pp. 221–235.

[30] P.E. Ammann and Ravi S. Sandhu, "Safety analysis for the extended schematic protection model," in *Proceedings of IEEE Symposium on Research in Security and Privacy*, Washington, DC, 1991, pp. 87–97, IEEE.

[31] Ravi S. Sandhu, "The schematic protection model: Its definition and analysis for acyclic attenuating schemes," *Journal of ACM*, vol. 36, no. 2, pp. 404–432, 1988.

[32] Patrick H. Wood and Stephen G. Kochan, *UNIX System Security*, Hayden Books, Indianapolis, 1985.

[33] Æleen Frisch, *Essential System Administration*, O'Reilly & Associates, New York, 1996.

[34] David Ferraiolo and Richard Kuhn, "Role-based access control," in *Proceedings of 15th National Computer Security Conference*, Baltimore, MD, 1992, pp. 437–443.

[35] David F. Ferraiolo and Janet A. Cugini and D. Richard Kuhn, "Role-based access control (rbac): Features and motivations," in *Proceedings of 11th Annual Computer Security Application Conference*, New Orleans, LA, 1992, pp. 241–48.

[36] David F. Ferraiolo and Ravi Sandhu and Serban Gavrila and D. Richard Kuhn and Ramaswamy Chandramouli, "Proposed NIST standard for role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[37] Roshan K. Thomas, "Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments," in *Proceedings of the second ACM Workshop on Role-based access control*, New York, NY, 1997, pp. 13–19, ACM Press.

[38] Gail-Joon Ahn and Ravi S. Sandhu, "Role-based authorization constraints specification," *Information and System Security*, vol. 3, no. 4, pp. 207–226, 2000.

[39] Steven Jon Greenwald, *The Distributed Compartment model for Resource Management and Access Control*, Ph.D. dissertation, University of Florida, 1994.

[40] Irene Greif and Sunil Sarin, "Data sharing in group work," in *CSCW '86: Proceedings of the 1986 ACM Conference on Computer-Supported Cooperative Work*, New York, NY, 1986, pp. 175–183, ACM Press.

[41] Clarence A. Ellis and Simon J. Gibbs and Gail Rein, "Groupware: some issues and experiences," *Commun. ACM*, vol. 34, no. 1, pp. 39–58, 1991.

[42] Adrian Bullock and Steve Benford, "An access control framework for multi-user collaborative environments," in *GROUP '99: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, New York, NY, USA, 1999, pp. 140–149, ACM Press.

[43] Joerg M. Haake and Anja Haake and Till Schummer and Mohamed Bourimi and Britta Landgraf, "End-user controlled group formation and access rights management in a shared workspace system," in *CSCW '04: Proceedings of the 2004 ACM Conference on Computer supported cooperative work*, New York, NY, 2004, pp. 554–563, ACM Press.

[44] HongHai Shen and Prasun Dewan, "Access control for collaborative environments," in *CSCW '92: Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work*, New York, NY, 1992, pp. 51–58, ACM Press.

[45] Joon S. Park and Junseok Hwang, "Role-based access control for collaborative enterprise in peer-to-peer computing environments," in *SACMAT '03: Proceedings of the Eighth ACM Symposium on access control models and technologies*, New Your, NY, 2003, pp. 93–99, ACM Press.

[46] Dieter Fensel, *Ontologies: A silver Bullet for Knowledge Management and Electronic Commerce*, Springer Verlag, 2003.

[47] Trent Jaeger and Atul Prakash, "Requirements of role-based access control for collaborative systems," in *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*. 1996, p. 16, ACM Press.

[48] Margaret Levi and Laura Stoker, "Political trust and trustworthiness," *Annual Review of Political Sciences*, vol. 3, pp. 475–507, 2000.

[49] Roderick M. Kramer, "Trust and distrust in organizations: Emerging perspectives, enduring questions," *Annual Review of Psychology*, vol. 50, pp. 569–598, 1999.

[50] Alfarez Abdul-Rahman and Stephen Hailes, "Supporting trust in virtual communities," in *HICSS-33: Hawaii International Conference on System Sciences*, Washington, DC, 2000, IEEE.

[51] Morton Deutsch, "Cooperation and trust: Some theoritical notes," in *Nebraska Symposium on Motivation*, M. R. Jones, Ed. 1962, Nebraska University Press.

[52] Diego Gambetta, Ed., *Trust*, Oxford: Basil Blackwell, 1990.

[53] Girish Suryanarayana and Richard N. Taylor, "A survey of trust management and resource discovery technologies in peer-to-peer applications," Tech. Rep. UCI-ISR-04-6, Institute for Software Research, University of California, Irvine, 2004.

[54] Peter A. Dinda, "Addressing the trust asymmetry problem in grid computing with encrypted computation," in *LCR '04: Proceedings of the 7th Workshop on Workshop on languages, compilers, and run-time support for scalable systems*, New York, NY, USA, 2004, pp. 1–7, ACM Press.

[55] D. Gritzalis, Ed., *Secure Electronic Voting*, Kluwer Academic Publishers, 2003.

[56] S. Marsh, *Formalising Trust as a Computational Concept*, Ph.D. dissertation, University of Stirling, 1994.

[57] V. Shmatikov and C. Talcott, "Reputation-based trust management," in *Proceedings of Workshop on Issues in the Theory of Security (WITS)*, 2003.

[58] "Ebay feedback scheme," http://pages.ebay.com/help/feedback/evaluating-feedback.html (July 10, 2005).

BIOGRAPHICAL SKETCH

Vijay Manian was born in Chennai, India, in 1977, the son of Padma and S.V.S. Manian. He has two elder sisters, Chithra and Vidya. After completing his primary schooling in Chennai, he attended Ida Scudder School, Vellore, India, where he completed his X. He completed high school from Padma Seshadri Bala Bhavan Junior College, Chennai, in 1995. He received a Bachelor of Engineering degree in computer science and engineering from the University of Madras in 1999. He then decided to pursue his master's degree in computer science from the University of Florida's Computer and Information Sciences and Engineering Department in 1999. After obtaining a Master of Science degree in 2002, he continued on to the Ph.D. program. He looks forward to a hopefully successful career in the software industry.