Proc. IEEE Symposium on Research in Security and Privacy, Oakland, California, May 1992, pages 148-161.

Non-Monotonic Transformation of Access Rights

Ravi S. Sandhu and Gurpreet S. Suri

Center for Secure Information Systems
&
Department of Information and
Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444

Abstract

The concept of transformation of access rights was recently introduced in the literature by Sandhu. It has been previously shown that monotonic transformations unify a number of diverse access control mechanisms such as amplification, copy flags, separation of duties and synergistic authorization. In this paper we demonstrate the importance and expressive power of non-monotonic transformations. A formal model called Non-Monotonic Transform (NMT) is defined. A distributed implementation of NMT is proposed using a client-server architecture. The implementation is remarkably simple and modular in concept. It is based on access control lists and allows for a variety of powerful revocation operations.

1 Introduction

The concept of transformation of access rights was recently introduced by Sandhu in [14]. It was shown in [14] that this concept unifies a surprising variety of access-control mechanisms found in the literature. If these various mechanisms were to be lumped together, the result would be a complex ad hoc model in totality. Instead it was demonstrated that a few basic concepts, viz., strong typing, grant transformations and internal transformations, suffice to express all these mechanisms and more.

The work in [14] was limited to *monotonic transformations*, i.e., transformations which only add access rights in the system but do not remove previously existing rights. It was shown in [14] that monotonic transformations encompass diverse access control mechanisms such as amplification, copy flags, synergistic authorization and some common forms of separation of duties.

In this paper we extend the concept of transformation to include *non-monotonic transformations*, i.e., transformations which not only add access rights in the system but in the process also remove previously existing rights. We demonstrate the importance and expressive power of non-monotonic transformations by means of a number of examples.

A central contribution of this paper is the formulation of a model called NMT (for Non-Monotonic Transform). In particular we show that NMT has a remarkably simple implementation in a distributed environment, using a client-server architecture. The implementation is based on access control lists and allows for a variety of powerful revocation operations. Moreover NMT has decidable safety analysis (i.e., the determination of whether or not a given subject can ever acquire a particular access right to a given object).

The proposed implementation is "object-oriented" in the sense that each server acts not only as the reference monitor for objects that it manages, but is also responsible for exporting semantically correct abstract operations on these objects. In other words the servers are part of the TCB (Trusted Computing Base), rather than subjects running on the TCB. This view has considerable benefits for integrity-oriented applications.

The rest of this paper is organized as follows. Section 2 discusses the basic concepts of monotonic and non-monotonic transformations in an informal manner. The NMT model is motivated and formalized in section 3. Two detailed security policies expressed in NMT are discussed in section 4. The proposed implementation of NMT is given in section 5. The implementation is further illustrated in context of a specific security policy in 6. Section 8 concludes the paper.

2 Transformations

Transformation of access rights involves two basic operations.

1. Internal transformation allows a subject who possesses certain rights for an object to obtain additional rights. In the course of doing so the subject may lose one or more rights previously possessed by the subject.

2. Grant transformation occurs in the granting of access rights by one subject to another. The general idea is that possession of a right for an object by a subject allows that subject to give some other rights for that object to another subject. Again, in the course of this process the subject may lose one or more rights previously possessed by the subject.

In general these transformations are *non-monotonic* in that they add some new rights to the system, as well as delete previously existing rights. In the special case where there is no deletion of rights, we say the transformations are *monotonic*.

2.1 Monotonic Transformations

Let us first briefly review some monotonic transformations from [14]. The simplest example of monotonic internal transformation of rights arises when one right is treated stronger than another. Consider the typical read, write and append operations on a file. From the semantics of these operations it is clear that possession of write should imply possession of append. The ability to obtain a weaker right by virtue of possessing a stronger one allows a subject to work with the least privileges needed. In some cases we require the stronger implication that write implies append and both imply read. The motivation is one of integrity in that a subject who writes a file should be able to check whether the writing has been carried out properly, which requires he be able to read the file. This is of course appropriate only in situations where non-disclosure is not an issue.

A more interesting application of monotonic internal transformation arises in situations described as synergistic authorizations [11]. For instance consider a situation where a scientist needs approval from a security-officer and a patent-officer before he can release a document for publication. Say these two approvals are respectively signified by possession of the a_s and a_p rights. A scientist then needs to be the owner of a document and must possess the two approvals before he can obtain the right to release the document. The synergy in this internal transformation occurs only if we can guarantee that the a_s and a_p rights are obtained from two independent sources. As we will see this can be achieved by grant transformations. This example is discussed in greater detail in section 4.

Next consider grant transformations. A simple form of grant transformation occurs with the copy flag (c). The concept goes back to the earliest abstract models for access-control [7] and is a fundamental aspect of discretionary controls. The xc right (a privilege x with a copy flag c appended to it) is typically made available to the creator of each object. In many access control models the ability to grant access is treated as stronger than the ability to perform access, that is possession of xc implies possession of x. Let us for the moment make this assumption, which of course is another example of monotonic internal transformation. Now consider the following policies.

1. A user who possesses the xc right for an object can grant the x right for that object to another user.

2. A user who possesses the xc right for an object can grant the xc or x right for that object to another user.

These are both examples of grant transformations. In the first case the xc right is transformed to the x right as part of the grant operation. In the second case there is a choice in the transformation, presumably at the volition of the subject doing the granting. The choice is between the identity transformation of xc to itself or an attenuating transformation of xc to x.

Numerous additional examples of monotonic grant and internal transformations are given in [14]. These include amplification for abstract data types, n-step copy flags, separation of duties, and variations of the simple-security and $\star$-properties.

2.2 Non-Monotonic Transformations

It is evident that monotonic transformations are a powerful and expressive concept in access control. Nevertheless, monotonic transformations have substantial limitations. For example, transfer-only privileges (e.g., ownership) and countdown privileges are inherently non-monotonic. It is a well known security principle to have only one owner of each object. This facilitates implementation of auditing and accountability procedures in a system. A situation may arise where a user needs to delegate authority to another user to work on an object on his behalf. To achieve this, the creator of the document needs to grant the other user the own right for that object. By making this grant the creator is stripped of the ownership privilege, and the other user becomes the owner of that object. A similar concept can be found in the UNIX operating system. The "chown" command, transfers the ownership to the new user with the user issuing the request losing the privilege for the object. Another example of a transfer-only privilege is a write privilege in a situation where we wish to have mutually exclusive writing to an object.

The expressive power of a non-monotonic model can be further appreciated in a situation where students submit their assignments electronically. Once a student has submitted an assignment for grading the student should lose the write privilege for it. Here the grant transformation from the student to the professor for grading the answer sheets is a non-monotonic operation in which the student loses the write privilege for the answer-sheets. We will return to this example in section 4.

3 The NMT Model

In this section we formally define and motivate the *Non-Monotonic Transform* (NMT) model. The protection state in NMT can be viewed in terms of the familiar access matrix [7]. There is a row for each subject in the system and a column for each object. In NMT the subjects and objects are disjoint. NMT does not define any access rights for operations on subjects, which are assumed to be completely autonomous entities.

NMT consists of a small number of basic constructs and a language for specifying the commands which cause changes in the protection state. For each command we have to specify the authorization required to execute that command, as well as the effect of the command on the protection state. We generally call such a specification as an *authorization scheme* (or simply scheme) [12].

A scheme in the NMT model is defined by specifying the following components.

1. A set of access rights R.

2. Disjoint sets of subject and object types, TS and TO respectively.

3. A collection of NMT commands. Each command specifies the authorization for its execution and the changes in the protection state effected by it.

The scheme is defined by the security administrator when the system is first set up and thereafter remains fixed. Each component of the scheme is discussed in turn below.

3.1 Rights

Each system has a specified set of rights R. It is important to understand that R is not specified in the model but varies from system to system. We will generally expect R to include the usual rights such as own, read, write, append and execute. But this is not required by the model. We also expect R to generally include more complex rights, such as review, grade-it, release etc. The meaning of these rights will be explained wherever they are used in our examples.

The access rights serve two purposes. Firstly, presence of a right, such as r, in the $[S, O]$ cell of the access matrix may authorize S to perform, say, the read operation on O. Secondly, presence of a right, say o, in $[S, O]$ may authorize S to perform some operation which changes the access matrix, e.g., by entering r in $[S', O]$. In other words, S as the owner of O can change the permissions in the access matrix so that S' can read O. The focus of NMT is on this second purpose of rights, i.e., the authorization by which the access matrix itself gets changed.

3.2 Types of Subjects and Objects

The notion of type is fundamental to NMT. All subjects and objects are assumed to be strongly typed. Strong typing requires that each subject or object is created to be of a particular type which thereafter does not change. The concept of type is a familiar one in computer science, and has a well-established tradition in the security arena. Its application in NMT is to group together subjects and objects into classes (i.e., types) so that instances of the same type have the same properties with respect to the authorization scheme.

Strong typing is analogous (but not identical) to tranquility in the Bell-LaPadula style of security models [2], whereby security labels on subjects and objects cannot be changed. The adverse consequences of unrestrained non-tranquility are well known [4, 9, 10]. Similarly, non-tranquility with respect to types has adverse consequences for the safety problem [15].

NMT requires that a disjoint set of subject types, TS, and object types, TO, be specified in a scheme.

For example, we might have TS={user, security-officer} and TO={user-files, systems-files} with the significance of these types indicated by their names.

3.3 NMT Commands

There are four kinds of commands in NMT.

1. Object Creation commands.
2. Grant Transformation commands.
3. Internal Transformation commands.
4. Revocation commands

These commands are described in detail below.

3.3.1 Creation Commands

First the formalism to create objects by subjects is presented.[1] Object creation is specified by a finite number of creation commands, each of which has the following format.

CREATE(S: u, O: o)
  create object O;
  enter Y **into** [S, O];
end

This command is interpreted as saying that a subject S of type u can create an object O of type o. The effect of creation is that the creator gets $Y = \{y_1, \ldots, y_m\}$ rights for the newly created object. In terms of the access matrix, a new column for O is created with all cells empty except for [S,O] which contains the rights $\{y_1, \ldots, y_m\}$.

Each **CREATE** command can be invoked, much like a procedure call, by substituting actual parameters for the formal parameters. However the command is executed only if the types of the subject and the object involved match the types of the formal parameters. It is further required that object O does not exist in the system, so that each object is created with a unique identity. In specifying the **CREATE** commands the security-administrator of the system can only specify the variables Y, and the subject and object types u and o respectively. The general structure of the **CREATE** procedure cannot be changed, and is a fixed component of NMT.

For example, consider a scheme where subjects of type user are authorized to create objects of type file, and upon creation of a file the creator gets the own right for it. This is specified by the following NMT command.

CREATE(S: user, O: file)
  create object O;
  enter {own} **into** [S, O];
end

The same scheme might also include the following command.

CREATE(S: user, O: doc)
  create object O;
  enter {own, read, write} **into** [S, O];
end

This authorizes users to create objects of type doc, and as a result of creation get the own, read and write rights for the object. Further examples of the **CREATE** command will be given in section 4.

3.3.2 Grant Transformation Commands

A grant transformation command has the general format given below.

GRANT_{X} (S1: u, S2: v, O: o)
  if $X \subseteq$ [S1, O] **then**
    enter Y **into** [S2, O];
    delete Z **from** [S1, O];
end

Here X, Y, and Z are subsets of R, with Z required to be a subset of X. A grant transformation is interpreted as follows: subject S1 of type u can grant Y $= \{y_1, ..., y_m\}$ rights for an object O of type o to subject S2 of type v provided S1 has $X = \{x_1 ..., x_n\}$ rights for O, but in the transformation S1 will lose the $Z = \{z_1 ..., z_i\}$ rights for O.

GRANT_{X} is a conditional procedure, where the possession of X rights in the domain of the subject granting the rights is checked prior to executing the **enter** and **delete** primitives. The security administrator can choose X, Y, Z, u, v and o, while specifying each **GRANT** command. The rest of the structure of a **GRANT** command is fixed as part of NMT and cannot be altered.

The transfer-only privilege of ownership, discussed in the previous section, can be easily expressed by the following **GRANT** command.

GRANT_{own} (S1: user, S2: user, O: file)
  if {own} $\subseteq$ [S1, O] **then**
    enter {own} **into** [S2, O];
    delete {own} **from** [S1, O];
end

[1] We have employed a procedural notation for defining NMT command in preference to a set theoretic notation. This is due to our focus in this paper on implementation issues, which are easier to relate to the formalism with a procedural notation.

Transfer of ownership of objects in a system is a common occurrence, and underscores the importance of non-monotonic operations.

A monotonic grant transformation is a special case of our **GRANT** command, in which Z is empty. In such cases we will omit the **delete** primitive from the body of the command. For example, the following is a monotonic command

GRANT_{own} (S1: user, S2: user, O: file)
  if {own} $\subseteq$ [S1, O] **then**
    enter {read} **into** [S2, O];
end

which allows the owner of a file to grant read access for that file to another user, without affecting ownership of the file.

The example of a mutually exclusive write privilege, discussed in section 2, can be expressed in NMT as follows.

GRANT_{write} (S1: user, S2: user, O: file)
  if {write} $\subseteq$ [S1, O] **then**
    enter {write} **into** [S2, O];
    delete {write} **from** [S1, O];
end

Here user S1 can give away the write access to O but only by losing it in the process. The motivation is one of safety so that both users do not end up writing to the file concurrently.

3.3.3 Internal Transformation Commands

An internal transformation command has the general format given below.

ITRANS_{X} (S: u, O: o)
  if X $\subseteq$ [S, O] **then**
    enter Y **into** [S, O];
    delete Z **from** [S, O];
end

As in grant transformation Z is a subset of X, and X and Y are subsets of R. The interpretation of this command is that a subject S of type u who has the $X = \{x_1 ..., x_n\}$ rights for an object O of type o can obtain the $Y = \{y_1 ..., y_m\}$ rights for O by internal transformation, but in the process S will lose the rights $Z = \{z_1 ..., z_i\}$ for O.

This is also a conditional procedure where the system checks for the presence X rights in the [S,O] cell of the matrix, before executing the command. The security-administrator specifies X, Y, Z and the types of the subject and object involved, i.e., u and o respectively, for each **ITRANS** command.

With this notation in place it is easy to express the n-step copy flag discussed in section 2. To understand the **ITRANS** command given below we define the xgive privilege first. It basically allows a user to do a one time grant of the x access to another user. This is expressed by the following non-monotonic grant command.

GRANT_{xgive} (S1: user, S2: user, O: file)
  if {xgive} $\subseteq$ [S1, O] **then**
    enter {x} **into** [S2, O];
    delete {xgive} **from** [S1, O];
end

Having defined xgive, the n-step copy flag policy is expressed by the following sequence of n **ITRANS** commands.

ITRANS_{xc^n} (S: user, O:file)
  if {xc^n} $\subseteq$ [S, O] **then**
    enter {xc^{n-1}, xgive} **into** [S, O];
    delete {xc^n} **from** [S,O];
end

ITRANS_{xc^{n-1}} (S: user, O:file)
  if {xc^{n-1}} $\subseteq$ [S, O] **then**
    enter {xc^{n-2}, xgive} **into** [S, O];
    delete {xc^{n-1}} **from** [S,O];
end

.
.
.

ITRANS_{xc} (S: user, O:file)
  if {xc} $\subseteq$ [S, O] **then**
    enter {xgive} **into** [S, O];
    delete {xc} **from** [S,O];
end

The above policy simulates a countdown copy flag access right, in which S by virtue of possessing xc^n for O can grant x for O exactly n times to other users.

If the Z set of rights is an empty set, the internal transformation reduces to its monotonic case. In such cases the **delete** command is omitted from the body of **ITRANS**. For example, consider the policy that write implies both append and read with no deletion of rights. This is expressed as shown below.

ITRANS_{write} (S: user, O:file)
  if {write} $\subseteq$ [S, O] **then**
    enter {append} **into** [S, O];
end

ITRANS_{write} (S: user, O:file)
  if {write} $\subseteq$ [S, O] **then**
    enter {read} **into** [S, O];
end

3.3.4 Revocation Commands

In distributed systems implementing revocation is a major problem, since the subjects are completely autonomous with no centralized authorities enforcing security. There are various issues with respect to which the implementation of revocation can be compared [16].

1. Partial or Complete: Whether it is possible to revoke a specific right or whether all rights have to be revoked to get any sort of denial of access in the system?
2. Immediate or Delayed: If the implementation executes revocation immediately or it comes into force only the next time the subject tries to access the object?
3. Selective or General: Does the revocation process affect all users or a select group of users having access over the object?
4. Temporary or Permanent: Is access to be denied permanently or if once it is revoked, is it retrievable?

A major advantage of a client-server architecture with a stateless server is that revocation takes effect immediately. Hence our implementation proposes to have stateless servers providing for immediate revocation. On the other aspects enumerated above our proposal provides for each of the possibilities identified.

We propose a simple revocation policy in NMT: only the owners of objects can revoke rights of other subjects for it. If an owner of an object grants a right to a second user and the second user in turn grants the right to a third user, only the owner is able to revoke the right for the second or third users. We provide revocation by using a dynamic ACL. When revocation is requested, the revoked rights are simply deleted from the list.

The command for Partial revocation is given below.

REVOKE(S1, S2, O, Z: set of rights)
  if {**own**} $\subseteq$ [S1, O] **then**
    delete Z **from** [S2, O];
end

Z is the only variable in this command which can be specified by the subject invoking the command. This command is interpreted as saying that if subject S1 is the owner of object O it can revoke Z = $\{z_1 ..., z_i\}$ rights of a subject S2 for the object O. This is a conditional procedure as well where the presence of own right is checked in [S1, O] before deleting Z rights from [S2, O]. Note that typing is not a relevant factor in this owner-based revocation.

Our syntax for **REVOKE** is slightly different that the syntax for the **ITRANS** and **GRANT** because the nature of **REVOKE** command is somewhat different. In particular Z is a variable here which can take on any value as determined by the subject invoking the command, whereas in the **GRANT** and **ITRANS** commands Z is fixed by the security policy and therefore is not a argument to the command.

The owner of an object has the flexibility either to have complete revocation (i.e. to revoke all the rights of other subjects for the object) or to invoke partial revocation (i.e. to revoke some subset of the rights). For partial revocation the server only deletes the rights $\{z_1 ..., z_i\}$ from the ACL that are specifically mentioned in the **REVOKE** command. For complete revocation the owner needs to specify all the rights a subject possesses or use the **REVOKE-ALL** command discussed below. The **REVOKE-ALL** command for complete revocation has the following format.

REVOKE-ALL(S1, O)
  if {**own**} $\subseteq$ [S1, O] **then**
    forallS2$\neq$S1
      do [S2, O] := ϕ;
end

This command is more powerful than the previous **REVOKE** command in that it eliminates all accesses to the object O by subject S2 other than the owner S1 in a single execution.

Our implementation has facilities for total denial of access as required in the TCSEC [5]. Total denial of access is indicated by the null right represented by the symbol $\perp$. This is a special right, whose occurrence in the ACL invalidates all other rights for that subject. So long as a subject "possesses" the $\perp$ right in an object's ACL, all access rights the subject may acquire through **GRANT** or **ITRANS** commands will not take effect, i.e., denial of access holds for these additional rights as well. This is a feature of permanent revocation.

REVOKE(S1, S2, O, $\perp$)
  if {**own**} $\subseteq$ [S1, O] **then**
    enter {$\perp$} **into** [S2, O];
end

In short if there is a $\perp$ present it supersedes all other rights that may be present and implies that the object is totally inaccessible to that subject.

3.4 Summary

This completes our definition of the NMT model. To summarize the NMT model, we have the following definition. A scheme for transformation of rights is stated in NMT by specifying the following components.

1. A set of rights R.
2. Disjoint sets of subject types TS and object types TO.
3. A finite set of NMT commands in the following format.
   (a) Object Creation commands.

   CREATE(S: u, O: o)
       create object O;
       enter Y **into** [S, O];
   end

   (b) Grant Transformation commands.

   GRANT_{X} (S1: u, S2: v, O: o)
       if X $\subseteq$ [S1, O] **then**
         enter Y **into** [S2, O];
         delete Z **from** [S1, O];
   end

   (c) Internal Transformation commands.

   ITRANS_{X} (S: u, O: v)
       if X $\subseteq$ [S, O] **then**
         enter Y **into** [S, O];
         delete Z **from** [S, O];
   end

4. The three Revocation commands.
   (a) **REVOKE**(S1, S2, O, Z: set of rights)
       if {**own**} $\subseteq$ [S1, O] **then**
         delete Z **from** [S2, O];
   end

   (b) **REVOKE-ALL**(S1, O)
       if {**own**} $\subseteq$ [S1, O] **then**
         forallS2$\neq$S1
           do [S2, O] := ϕ;
   end

   (c) **REVOKE**(S1, S2, O, $\perp$)
       if {**own**} $\subseteq$ [S1, O] **then**
         enter {$\perp$} **into** [S2, O];
   end

In all these commands X, Y, and Z are subsets of R. Moreover Z is also a subset of X.

Revocation commands are usually not listed in the definitions of our schemes since they are fixed in NMT and do not vary from scheme to scheme.

3.5 Safety Analysis of NMT

Safety analysis determines, for a given scheme and initial state, whether or not it is possible for a given privilege to be acquired by a particular subject. Safety analysis issues were first formalized by Harrison, Ruzzo and Ullman [6] in a model commonly known as HRU. It has been previously shown by Lipton and Snyder [8] that for the case of HRU with absence of subject creation safety is decidable. Now NMT allows only object creation with no provisions for subject creation. It can be shown that NMT is subsumed by this subcase of HRU. This demonstration is omitted here due to lack of space. It does follow that safety is decidable for NMT.[2]

4 Example Schemes

This section provides two detailed schemes in NMT so as to demonstrate the natural expressive power of the model. The examples include both monotonic as well as non-monotonic aspects.

4.1 Grading Example

Consider the example of a class in which a professor gives the students a take-home test, whose solution is to be prepared and submitted electronically. A student creates a document called Answer-Sheet on which he/she writes the answers. As a result of creating Answer-Sheet, the student gets the own, read and write privileges for it. Once the solution has been completed, Answer-Sheet is submitted to the professor for grading. To prevent the student from altering Answer-Sheet once it has been submitted for grading, it is imperative that the student loses the ability to write on Answer-Sheet. In other words, submission of the Answer-Sheet will be by means of a non-monotonic grant transformation. We will arrange that the professor, to whom Answer-Sheet has been submitted for grading, can acquire read and append access to Answer-Sheet by internal transformations.

[2] It should be noted that the decision procedure of [8] has exponential complexity. Because of the simplicity of NMT relative to the general case considered in [8], we are hopeful that efficient safety analysis for NMT can be achieved.

The student will be allowed to retain the read privilege for Answer-Sheet, so as to discuss the grading.

An NMT scheme for this policy is shown below.

1. R = {own, read, write, append, grade-it}
2. TS = {student, faculty}, TO = {answer-sheets}
3. The commands of the scheme are expressed below.

CREATE(S: student, O: answer-sheets)
  create object O;
  enter {own, read, write} **into** [S, O];
end

GRANT_{own, write} (S1: student, S2: faculty, O: answer-sheets)
  if {own, write} $\subseteq$ [S1, O]; **then**
    enter {grade-it} **into** [S2, O];
    delete {write} **from** [S1, O];
end

ITRANS_{grade-it} (S: faculty, O: answer-sheets)
  if {grade-it} $\subseteq$ [S, O] **then**
    enter {read, append} **into** [S, O];
end

The subjects in this system are of the types faculty and student, for professors and students respectively. There is only one object type, viz., answer-sheets.

The student in our scenario, creates Answer-Sheet using the **CREATE** command above, prepares the solution, submits it for grading to the professor by using the **GRANT_**{own,read} command, who then uses the **ITRANS_**{grade-it} command to acquire read and append access to Answer-Sheet. Realistically, this would be a fragment of a larger scheme which included additional object types and perhaps additional subject types to deal with other access control aspects of the overall system policy.

4.2 Document Release Example

Next let us take the case of a scientist who creates a document and consequently gets the own, read and write privileges for it. We stipulate that prior to releasing this document for publication, the scientist needs approvals from two separate and independent sources. The security-officers and the patent-officers of the organization are the two types of users who can grant the scientist each of these separate approvals. They, of course, need to review the document before granting approval.

After preparing the document for publication, the scientist can internally transform the own right to acquire the seek-approval right, which is the necessary authorization to request the security and the patent officers to review the document. This is a non-monotonic transformation in which the scientist gets seek-approval privilege but in the process loses the write privilege for the document. This is done to ensure that the scientist is not able to alter the document once the review process has started. In order to enforce this point of the policy the write privilege of the scientist is deleted once the request for a review of the document has been invoked.[3]

With the seek-approval right for a document, the scientist can request the security-officer and the patent-officer to review the document. Once the security-officer and the patent-officer have reviewed the document, they grant the scientist their respective approvals. And after granting their approvals, it is reasonable to disallow any further attempts by them to review the document. Thus their review privilege for the document is deleted at this point.

Now, having obtained both approvals the scientist can internally transform and acquire the release privilege needed to release the document for publication.

This policy is expressed by the following NMT scheme.

1. R = {own, read, write, seek-approval, review, a_s, a_p, release}
2. TS = {sci, sec-off, pat-off}, TO = {doc}
3. The commands of the scheme are given below.

CREATE(S: sci, O: doc)
  create object O;
  enter {own, read, write} **into** [S, O];
end

ITRANS_{own, write} (S: sci, O: doc)
  if {own, write} $\subseteq$ [S, O] **then**
    enter {seek-approval} **into** [S, O];
    delete {write} **from** [S,O];
end

GRANT_{seek-approval} (S1: sci, S2: sec-off, O:doc)
  if {seek-approval} $\subseteq$ [S1, O]; **then**
    enter {review} **into** [S2, O];
end

[3] It is assumed, and must be ensured by the scheme, that no subject other than the owner in the system can possesses the write privilege for objects of type doc.

GRANT_{seek-approval} (S1: sci, S2: pat-off, O:doc)
    if {seek-approval} $\subseteq$ [S1, O]; **then**
      enter {review} **into** [S2, O];
end

GRANT_{review} (S1: sec-off, S2: sci, O: doc)
    if {review} $\subseteq$ [S1, O]; **then**
      enter {a_s} **into** [S2, O];
      delete {review} **from** [S1, O];
end

GRANT_{review} (S1: pat-off, S2: sci, O: doc)
    if {review} $\subseteq$ [S1, O]; **then**
      enter {a_p} **into** [S2, O];
      delete {review} **from** [S1, O];
end

ITRANS_{a_s, a_p} (S: sci, O: doc)
    if {a_s, a_p} $\subseteq$ [S, O] **then**
      enter {release} **into** [S, O];
end

5 Implementation Of NMT

This section proposes an implementation of NMT in a distributed system, using the typical client-server architecture. Each server acts as a mediator for the set of objects it manages. For simplicity of exposition, we assume that each server manages exactly one type of object. The extension to servers which manage several types of objects is straightforward. We do allow the same type of object to be managed by several servers. Each of these servers will manage a disjoint collection of instances of this object type.

All accesses to an object pass through its server who determines the validity of the request. The object server furthermore is responsible for ensuring semantic correctness of the objects with respect to the abstract operations exported from the server. In other words objects are encapsulated within their servers, for both integrity and information hiding in the sense of data abstraction as well as for access control.

We emphasize that the object servers are not subjects in the system but rather a part of the trusted computing base (TCB). We assume each server is physically secure with integrity and authentication of the client-server communication ensured by using the standard encryption based techniques [3]. If necessary, confidentiality of the communication can also be ensured by encryption.

We assume the server maintains an access control list (ACL) for each of the objects it is required to manage. We depict the access list as follows.

object	
subjects	rights

An ACL is associated with each object, specifying the subjects who can access the object and the access right authorized for each of them. The ACL makes the access to the object dependent on the identity of the subject. Every time a subject makes a request—whether it be for access to the object (e.g., read the object) or for a grant or internal transformation of rights—the request is checked against this list. The access request is valid only if the access requested is authorized by the rights present for the subject in the ACL for the object.

This ACL is dynamic in nature. Every time the server receives a request for creation of a new object the **CREATE** commands are checked. If the **CREATE** commands allow such a creation, the new object is created with an entry in the access list. The access rights for it are added to the list according to the **enter** primitive of the **CREATE** commands. The ACL is similarly checked and modified for the **ITRANS** and **GRANT** commands.

Each server only needs to know about the **CREATE**, **ITRANS** and **GRANT** commands for the object type that it manages. This makes the system modular, in the sense that new object types can be introduced (along with their servers) without any need to inform previously existing servers. Similarly when new subject types are introduced, only those servers whose commands are affected by this introduction need to be informed.

The rest of this section discusses various aspects of the proposed implementation in detail.

5.1 Identity and Type

Every subject and object is assigned a type when it gets created. The typing is strong and cannot be altered thereafter. Moreover each subject or object in the system has a globally unique identifier. Subjects can only be created by the system administrator, who assigns their type and unique name at the moment of creation. Objects are created by object servers in response to requests by subjects. Each object server manages only one type of object, so the type is uniquely determined. The object servers ensure that each object has a globally unique identifier.

We assume the type of a subject or object is embedded in its identifier. Henceforth, we refer to a subject identifier by *sid* and a object identifier by *oid*. These identifiers have the following structure.

type	identifier

The type field denotes the type of the subject or the object. The identifier field uniquely identifies each subject or object among instances of the same type. Uniqueness of oid's reduces to requiring each object to have a unique identifier among instances of the same type. If a particular type is managed by more than one server, uniqueness of oid's can be ensured by having the following structure.

type	server identifier	identifier

Having made this point, we will use the former oid structure in rest of this paper.

5.2 Access Mediation

All accesses to a object are mediated by the server responsible for managing that object. Authentication is also carried out at the time of object access, and must be incorporated into the RPC (Remote Procedure Call) mechanism of the client-server architecture. The object server must authenticate the source of every RPC request. This can be achieved by any of the encryption protocols found in the literature [3]. One method would be to arrange for every subject to place its digital signature on every RPC communication to a object server. Digital signatures for the reverse communication from object servers to subjects can also be incorporated.

5.3 Creation

When a subject requests an object creation the object server checks the **CREATE** commands given below to check the validity of the request.

CREATE (S: u, O: o)
  create object O;
  enter Y **into** [S, O];
end

In this command sid = u.S. When the server receives the object creation request it performs a series of actions listed below.

1. The first step involves authenticating the subject. If the authentication test fails the request is denied.
2. Then the server checks the types of the subject S and object O involved in creation. If the subject S is of type u and object O of type o the request is valid and the server goes on to execute the next two steps. If the request is invalid (the types of the subjects and the objects do not match), the command procedure is aborted.
3. Next the server makes the existence check, i.e., whether the object O already exists in the server or not? If it does the **CREATION** request is refused, otherwise a new object O of type o is created with an empty ACL. The oid = o.O is assigned to this object.
4. And finally, the **enter** primitive of the procedure is checked to determine the rights to be awarded to the subject S for the object O. The Y = $\{y_1..., y_m\}$ rights are added into the ACL under the respective object heading for the subject S.

5.4 Grant Transformation

A grant request from a subject S1 of sid = u.S1 for object O of oid = o.O to a subject S2 of sid = v.S2, is implemented according to the following definition.

GRANT_X (S1: u, S2: v, O: o)
    if X $\subseteq$ [S1, O] **then**
      enter Y **into** [S2, O];
      delete Z **from** [S1, O];
end

When the object server receives the **GRANT** request it performs the various tasks listed below.

1. On receiving the **GRANT** request the server authenticates the subject. If the authentication check fails the request is denied.
2. The server then attempts to validate the **GRANT** request by checking it with respect to the authorized **GRANT** commands. For the request to be valid the subject S1 has to be of type u, subject S2 of type v and the object type o. If these types do not match, the request is turned down.
3. Then the server checks that S1 possesses X=$\{x_1..., x_n\}$ rights in the ACL.[4] If these rights are present only then the server proceeds to execute the next two steps, otherwise the request is not serviced.
4. Next the Z=$\{z_1, ..., z_i\}$ rights of subject S1 for the object O are deleted from the ACL. Recall that these deleted rights are a subset of X.

[4] In this interpretation the $\perp$ right negates access rights, such as read and write, used for true access to the object. However, $\perp$ does not negate access rights for the purpose of transformation operations. NMT can, of course, be easily extended to allow for $\perp$ to override existing access rights for this purpose too.

5. The last step is the addition of transformed rights. The server adds to the ACL for the subject S2 the $Y=\{y_1, ..., y_m\}$ rights for the object O.

5.5 Internal Transformation

All internal transformations are interpreted according to the definition given below for the type of subject and object involved. Here the sid = u.S and oid = o.O.

ITRANS_{X} (S: u, O: o)
  if X $\subseteq$ [S, O] **then**
    enter Y **into** [S, O];
    delete Z **from** [S, O];
end

The server on receiving the **ITRANS** request performs a series of actions enumerated below.

1. As before, the server authenticates the subject. If the authentication check fails the request is denied.
2. The validity of the **ITRANS** request is checked by matching the types of subjects and objects involved. The request is valid if the subject S is of type u and the object O of type o, otherwise the request is turned down.
3. The server then checks the access list to see if the subject possesses the $X=\{x_1 ..., x_n\}$ rights for the object O. If this is true then only the server executes the following two steps, otherwise it denies the subject's request.
4. Next the $Z=\{z_1 ..., z_i\}$ rights (a subset of X) of the subject S for that object O are deleted from the access list.
5. And lastly the server adds the to the ACL for the subject S the $Y=\{y_1 ..., y_m\}$ rights for the object O.

5.6 Revocation

Partial revocation is implemented using the command given below.

REVOKE(S1, S2, O, Z: set of rights)
  if {**own**} $\subseteq$ [S1, O] **then**
    delete Z **from** [S2, O];
end

On seeing the **REVOKE** command the server executes the following steps.

1. First the server authenticates the subject. If the authentication fails the request is denied.
2. The server then checks in the ACL if the subject S1 issuing the request has own privilege for the object O. If this is false the request is turned down and if it is true (i.e. S1 is the owner of O) the server executes step 3.
3. The server deletes the $\{z_1 ..., z_i\}$ rights of S2 for O from the ACL.

For complete revocation for all users the following command is invoked.

REVOKE-ALL(S1, O)
  if {**own**} $\subseteq$ [S1, O] **then**
    forallS2$\neq$S1 **do** [S2, O] := ϕ;
end

This command too is implemented as the previous one except that the server purges all entries in the ACL for O except those entries for S1.

Finally for denial of access we have the following command.

REVOKE(S1, S2, O, $\perp$)
  if {**own**} $\subseteq$ [S1, O] **then**
    enter {$\perp$} **into** [S2, O];
end

The implementation of this command is also similar to the **REVOKE** command except that in step 3 instead of deleting the $\{z_1 ..., z_i\}$ rights of S2 for O it adds $\perp$ right into the ACL.

6 Implementation Example

In this section we consider the security and patent officer example discussed in section 4. Our aim is to demonstrate how the policy is implemented using the protocols described in the previous section.

In the scheme of section 4 a scientist (say Tom) creates a document (say TST) on his workstation. In order to release this document Tom needs multiple approvals, one from the security-officer and another from the patent-officer. The following is the sequence of steps is required to achieve the release of TST.

1. Tom requests creation of TST. The kernel of Tom's host, makes a remote procedure call (RPC) to the object server which is responsible for the objects created by Tom. This RPC contains the action requested, and the sid; all signed under Tom's digital signature. In this instance, the sid = sci.Tom.

2. On receiving the request the server authenticates the origin as being Tom. The server then checks the **CREATE** commands to ascertain the validity of the request:

   CREATE(Tom: sci, TST: doc)
       create object TST;
       enter {own, read, write} **into** [Tom, TST];
   end

   After determining the validity of the request it performs the existence check to ascertain whether there exists another object in the server with the name TST. Suppose there exists no other object with name TST. The server then creates a document TST with oid = doc.TST and an initially empty ACL. The ACL is then adjusted as per the **enter** primitive to give us the following.

	doc.TST
sci.Tom	own, read, write

   That is, Tom is the owner of TST and can read and write into it.

3. Now Tom is ready to release the document. He initiates the internal transformation request to the server to get the seek-approval right through the **ITRANS** command. The RPC has information about the action requested and the oid and sids of the subjects and objects involved. The **ITRANS** request is of the following form:

   ITRANS_{own, write} (Tom: sci, TST: doc)

   The server on receiving the RPC authenticates its origin as Tom. Then the server checks the types of the Tom and object TST. As they match with the required types for the above **ITRANS** command invocation, the server proceeds to check in the ACL for TST whether Tom has the own and write privileges for TST. Next the server adds into the ACL for TST the seek-approval right for Tom and deletes the Tom's write right from the ACL. The updated ACL is as follows.

	doc.TST
sci.Tom	own,read,seek-approval

   Note the write is now missing in the ACL, hence Tom's subsequent attempts to write to TST will fail. Tom cannot modify TST even after he gets approvals from both the security-officer and the patent-officer.

4. With the seek-approval privilege for TST, Tom can have TST reviewed by a security-officer, say Sam, and a patent-officer, say Jill. Tom issues two separate **GRANT** commands.

   GRANT_{seek-approval} (Sam: sec-off, TST: doc)
   GRANT_{seek-approval} (Jill: pat-off, TST: doc)

   For each of the above requests the server, as before authenticates the origin of the request as Tom. Then the server checks whether Tom, Sam, Jill and TST are of the type sci, sec-off, pat-off and doc respectively. As the request is valid the server services the request by adding the review rights in the ACL as shown below.

	doc.TST
sci.Tom	own,read,seek-approval
sec-off.Sam	review
pat-off.Jill	review

   No rights are deleted from the ACL as this is a monotonic operation. (There is, of course, no need for these **GRANT**s to be made at the same time. It just makes our exposition briefer. Similarly, the **GRANT**s below do not need to be synchronized.)

5. Now the security-officer and the patent-officer can access TST to review it. After review the officers can send their respective approvals to Tom. They both individually request the server to grant Tom the approval rights. The **GRANT** requests of Sam and Jill are shown below respectively.

   GRANT_{review} (Sam: sec-off, Tom: sci, TST: doc)
   GRANT_{review} (Jill: pat-off, Tom: sci, TST: doc)

   On receiving these requests the server makes the necessary authentication and validity tests. As per the definitions of the **GRANT** commands, the server updates the ACL in the following way.

	doc.TST
sci.Tom	own,read,seek-approval,a_s, a_p

   Note that the entries for Sam and Jill have been deleted, since the policy requires that once Sam and Jill grant the approval rights to Tom, their review privilege for TST will be deleted.

6. Now the scientist Tom possesses the approvals to get the release privilege by internal transformation. The RPC for the the internal transformation is shown below.

   ITRANS_$\{a_s, a_p\}$ (S: sci, O:doc)

   The server then checks the ACL to see if Tom has the a_s and a_p rights for TST, which are required to get the additional release privilege. As this is in the affirmative the server updates the ACL to provide Tom with the release right.

doc.TST	
sci.Tom	own,read,seek-approval,a_s, a_p,release

   With this release right domain Tom can release TST.

This completes the grant and internal transformation aspect of the example.

To illustrate the implementation of revocation aspect of the model consider a subject (say Jack) of type user who has the own, read and write privileges for a object (say SDI) of the type doc. Also assume there is another user (say Mary) also possesses the read, write and execute privileges for SDI. Our revocation policy allows Jack to revoke the access rights of Mary. The ACL for SDI has the following entries before the **REVOKE** command is invoked.

doc.SDI	
user.Jack	own,read,write
user.Mary	read,write,execute

Revocation is illustrated as follows.

1. Jack being the owner of SDI can revoke any of the rights Mary has for SDI. For Jack to revoke the execute privilege of Mary the following **REVOKE** command is sent to the server.

   REVOKE (Jack, Mary, SDI, execute)

   The server receives the command in the form of an RPC from Jack's workstation.

2. When the **REVOKE** command is invoked, the server invokes the following implementation procedure.

   REVOKE(Jack, Mary, SDI, execute)
   if $\{$**own**$\} \subseteq$ [Jack, SDI] **then**
   delete $\{$execute$\}$ **from** [Mary, SDI];
   end

   Besides making the regular authentication checks, the server verifies the possession of own privilege by Jack for SDI by looking in SDI's ACL. When this fact is confirmed the server removes the execute privilege from Mary's domain. The updated ACL is given below.

doc.SDI	
user.Jack	own,read,write
user.Mary	read,write

3. After this if Mary tries to execute SDI, her request will be denied by the server. For Jack to revoke all the rights of Mary he would have to make additional **REVOKE** request for read and write.

4. If Jack wants SDI to be totally inaccessible to Mary he can "grant" Mary the $\perp$ right.

   REVOKE(Jack, Mary, SDI, $\perp$)
   if $\{$**own**$\} \subseteq$ [Jack, SDI] **then**
   enter $\{\perp\}$ **into** [Mary, SDI];
   end

   The server verifies that Jack is the owner of SDI. When this is confirmed the server enters the $\perp$ right into Mary's entry in the ACL for SDI, as follows.

doc.SDI	
user.Jack	own,read,write
user.Mary	$\perp$,read,write

   Any future accesses by Mary to SDI will be denied by the server. The $\perp$ right overrides any other rights the subject may have for the object. It may be noted that the access rights are not deleted from the ACL in this denial of access request. Only the owner of the object can invoke the denial of access procedure and only the owner can restore services by revoking the null right from the subject's domain.

To summarize, in our implementation any subject who is the owner of an object can revoke the rights of other subjects for the same object. The implementation provides facilities for an efficient revocation with total denial of access service built into it.

7 Revocation

Revocation was not formally defined in the abstract NMT model, because it is usually closely tied to the implementation due to performance reasons. In distributed systems implementing revocation is a major problem, since the subjects are completely autonomous with no centralized authorities enforcing security. There are various issues with respect to which the implementation of revocation can be compared [16].

1. Partial or Complete: Whether it is possible to revoke a specific right or whether all rights have to be revoked to get any sort of denial of access in the system?

2. Immediate or Delayed: If the implementation executes revocation immediately or it comes into force only the next time the subject tries to access the object?

3. Selective or General: Does the revocation process affect all users or a select group of users having access over the object?

4. Temporary or Permanent: Is access to be denied permanently or if once it is revoked, is it retrievable?

A major advantage of a client-server architecture with a stateless server is that revocation takes effect immediately. Hence our implementation proposes to have stateless servers providing for immediate revocation. On the other aspects enumerated above our proposal provides for each of the possibilities identified.

We provide revocation by using a dynamic ACL. When revocation is requested, the revoked rights are simply deleted from the list. We propose a simple revocation policy for our implementation of NMT: only the owners of objects can revoke rights of other subjects for it. If an owner of an object grants a right to a second user and the second user in turn grants the right to a third user, only the owner is able to revoke the right for the second or third users.

Partial revocation is implemented using the command given below.

REVOKE(S1, S2, O, Z: set of rights)
    if own $\in$ [S1, O] **then**
        delete Z **from** [S2, O];
end

Z is the only variable in this command which can be specified by the subject invoking the command.[5] On seeing the **REVOKE** command the server executes the following steps.

1. First the server authenticates the subject. If the authentication fails the request is denied.

2. The server then checks in the ACL if the subject S1 issuing the request has own privilege for the object O. If this is false the request is turned down and if it is true (i.e. S1 is the owner of O) the server executes step 3.

3. The server deletes the $\{z_1..., z_n\}$ rights of S2 for O from the ACL.

Note typing is not an important factor in this owner-based revocation. Anyone can revoke anybody's access privileges provided they are the owner of that object.

The owner of a object can revoke all the rights (complete revocation) of other subjects or he can revoke some subset of the rights (partial revocation). The server only deletes the rights $\{x_1..., x_n\}$ from the ACL that are specifically mentioned in the **REVOKE** command. For complete revocation the owner needs to specify all the rights a subject possesses and consequently the server deletes all the rights. And similarly for partial revocation he needs to specify only those rights he wants deleted. For complete revocation "X" is equal to the set of rights in the revoked subject's domain for a object while in partial revocation "X" is a proper subset of these rights.

We also provide for complete revocation for all users by the following command.

REVOKE-ALL(S1, O)
    if own $\in$ [S1, O] **then**
        forall S2$\neq$S1 **do** [S2, O] := ϕ;
end

In terms of the ACL this command is easily implemented by purging all entries in the ACL for O except those entries for S1.

Our implementation has facilities for total denial of access as required in the TCSEC [5]. Total denial of access is indicated by the null right represented by the symbol $\perp$. This is a special right, whose occurrence in the ACL invalidates all other rights for that subject. So long as a subject possesses the $\perp$ right in an object's ACL, all access rights the subject may acquire through **GRANT** or **ITRANS** commands will not take effect, i.e., denial of access holds for these

[5] Our syntax for **REVOKE** is slightly different that the syntax for the **ITRANS** and **GRANT** because the nature of **REVOKE** command is somewhat different. In particular Z is a variable here which can take on any value as determined by the subject invoking the command, whereas in the **GRANT** and **ITRANS** commands it is fixed by the security policy and therefore is not a argument to the command.

additional rights as well. This is a feature of permanent revocation. The ACL does not specify negative rights. If a particular access right is not present that particular type of access is refused. And if there is a $\perp$ present, it supersedes all other rights that may be present and implies as the object being totally inaccessible to that subject.

To illustrate the implementation of revocation aspect of the model consider that a subject (say Jack) of type user has the own, read and write privileges for a object (say SDI) of the type doc. Also assume there is another user (say Mary) also who possesses the read, write and execute privileges for SDI. Our revocation policy allows Jack to revoke the access rights of Mary. The ACL for SDI has the following entries before the **REVOKE** command is invoked.

doc.SDI	
user.Jack	own,read,write
user.Mary	read,write,execute

Revocation is illustrated as follows.

1. Jack being the owner of SDI can revoke any of the rights Mary has for SDI. For Jack to revoke the execute privilege of Mary the following **REVOKE** command is sent to the server.

   REVOKE (Jack; Mary; SDI; execute)

   The server receives the command in the form of an RPC from Jack's workstation.

2. When the **REVOKE** command is invoked, the server invokes the following implementation procedure.

   REVOKE(Jack; Mary; SDI; execute)
           if own $\in$ [Jack, SDI] **then**
             delete {execute} **from** [Mary, SDI];
   end

   Besides making the regular authentication checks, the server verifies the possession of own privilege by Jack for SDI by looking in the ACL. When this fact is confirmed the server removes the execute privilege from Mary's domain. The updated ACL is given below.

doc.SDI	
user.Jack	own,read,write
user.Mary	read,write

3. After this if Mary tries to execute SDI, her request will be denied by the server. For Jack to revoke all the rights of Mary he would have to make additional **REVOKE** request for read and write.

4. If Jack wants SDI to be totally inaccessible to Mary he can "grant" Mary the $\perp$ right.

   REVOKE(Jack; Mary; SDI; *)
           if own $\in$ [Jack, SDI] **then**
             enter $\{\perp\}$ **into** [Mary, SDI];
   end

   On seeing the * the server interprets the command as if total denial of access for Mary for the object SDI is being requested. First the server verifies that Jack is the owner of SDI. When this is confirmed the server enters the $\perp$ right into Mary's entry in the ACL for SDI, as follows.

doc.SDI	
user.Jack	own,read,write
user.Mary	$\perp$,read,write

   Any future accesses by Mary to SDI will be denied by the server. The $\perp$ right is considered to override any other rights the subject may have for the object. It may be noted that the access rights are not deleted from the ACL in this denial of access request. Only the owner of the object can invoke the denial of access procedure and only the owner can restore services by revoking the null right from the subject's domain.

To summarize, in our implementation any subject who is the owner of an object can revoke the rights of other subjects for the same object. The implementation provides facilities for an efficient revocation with total denial of access service built into it.

8 Conclusion

In this paper we have demonstrated the importance and expressive power of non-monotonic transformations. We have developed a formal model called NMT (for Non-Monotonic Transform) and shown that it has decidable safety analysis. NMT is based on a few concepts and yet is very expressive and flexible as shown by the examples in this paper.

We have proposed a simple and efficient implementation for NMT in a distributed environment, using a client-server architecture. This implementation is based on access control lists. It provides for efficient and immediate revocation which could be partial, complete, selective, temporary or permanent.

References

[1] Ammann, P. and Sandhu, R.S. "Extending the Creation Operation in the Schematic Protection Model." *Proc. Sixth Annual Computer Security Applications Conference*, 340-348 (1990).

[2] Bell, D.E. and LaPadula, L.J. "Secure Computer Systems: Unified Exposition and Multics Interpretation." MTR-2997, Mitre, Bedford, Massachusetts (1975).

[3] Davies, D.W. and Price, W.L. *Security in Computer Networks.* John Wiley & Sons (1989).

[4] Denning, D.E. "A Lattice Model of Secure Information Flow." *Communications of ACM* 19(5):236-243 (1976).

[5] Department of Defense National Computer Security Center. *Department of Defense Trusted Computer Systems Evaluation Criteria.* DoD 5200.28-STD, (1985).

[6] Harrison, M.H., Russo, W.L. and Ullman, J.D. "Protection in Operating Systems." *Communications of ACM* 19(8):461-471 (1976).

[7] Lampson, B.W. "Protection." *5th Princeton Symposium on Information Science and Systems*, 437-443 (1971). Reprinted in *ACM Operating Systems Review* 8(1):18-24 (1974).

[8] Lipton, R.J. and Snyder, L. "On Synchronization and Security." In DeMillo, R.A., Dobkin, D.P., Jones, A.K. and Lipton, R.J. (Editors). *Foundations of Secure Computations.* Academic Press (1978), pages 367-385 (1978).

[9] McLean, J. "A Comment on the 'Basic Security Theorem' of Bell and LaPadula." *Information Processing Letters* 20(2):67-70 (1985).

[10] McLean, J. "Specifying and Modeling Computer Security." *IEEE Computer* 23(1):9-16 (1990).

[11] Minsky, N. "Synergistic Authorization in Database Systems." *7th International Conference on Very Large Data Bases* 543-552 (1981).

[12] Sandhu, R.S. "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes." *Journal of ACM* 35(2):404-432 (1988).

[13] Sandhu, R.S. "Transaction Control Expressions for Separation of Duties." *Aerospace Computer Security Applications Conference*, 282-286 (1988).

[14] Sandhu, R.S "Transformation of Access Rights" *IEEE Symposium on Security and Privacy*, 259-268 (1989).

[15] Sandhu, R.S. "The Typed Access Matrix Model" *IEEE Symposium on Research in Security and Privacy*, this proceedings (1992).

[16] Silberschatz, A., Peterson, J., and Galvin, P. *Operating System Concepts.* Addison Wesley (1991).

Acknowledgments

We are indebted to Howard Stainer and Sylvan Pinsky for their support and encouragement, making this work possible.