Computers & Security, 12 (1993) 285-303

Refereed paper

A distributed capability-based architecture for the Transform model

Ravi S. Sandhu and Gurpreet S. Suri

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

The Transform model is based on the concept of transformation of access rights. It unifies a number of diverse access-control mechanisms such as amplification, copy flags, separation of duties and synergistic authorization. In this paper we describe a distributed architecture for implementing Transform. Our architecture is based on capabilities with identities of subjects buried in them. This ensures unforgeability of capabilities and enables enforcement of non-discretionary controls on propagation of capabilities from one subject to another. The design provides for immediate, selective, partial and complete revocation on a temporary as well as a permanent basis. We also show that Transform has an efficient algorithm for safety analysis of the propagation of access rights (i.e., the determination of whether or not a given subject can ever acquire access to a given object).

Keywords: Distributed systems, Secure architectures, Capabilities, Safety.

1. Introduction

The Transform model unifies a variety of access-control mechanisms which deal with diverse security issues. These mechanisms are mostly taken from the existing literature. Some have been implemented in actual systems. They all have merit and should certainly be supported, in one form or another, by any protection system which claims to be of general applicability. However, considered in isolation these mechanisms are diverse and most have been proposed independently of each other. Simply lumping them together would result in a complex ad hoc model in totality. This is not only inelegant but also casts doubts about prospects for safety analysis (i.e., for determining whether or not a particular subject can obtain a specific right for some given object).

The unifying concept of *transformation of access rights* was proposed in [27] to abstract the common foundation of these mechanisms. Transformation of rights takes place in two different ways:

(1) *Self transformation* or *internal transformation* allows a subject who possesses certain rights for an object to obtain additional rights.

(2) *Grant transformation* or *external 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 right for that object to another subject.[1]

[1]If a subject can grant transformed rights to itself external transformation implies internal transformation. In most applications there are additional controls to prevent such "self granting."

Internal transformations allow us to express consistency in access-control policies such as the requirement that write access implies append access. The well-known technique of amplification [3, 31] for supporting abstract data types and protected subsystems is another instance of internal transformation. The case for abstract data types and protected subsystems is well argued in several classic papers [5, 14, 21, 31]. More recently it has been argued [2] that the "access control triple", which is essentially similar in concept, is necessary for support of integrity policies.

Grant transformations allow us to accommodate various kinds of integrity controls. For instance, we can distinguish the ability to access an object from the ability to grant access to that object. This distinction has been suggested as an essential part of "commercial" access-control policies [19] and is implemented in actual systems such as IBM's RACF (Resource Access Control Facility). This distinction of course is one form of separation of duties. Another instance of grant transformations arises when operations on an object are constrained to occur in a specific sequence. This has similarities to the manner in which separation of duties is enforced by transaction control expressions [26].

Our principal objective in this paper is to describe an architecture and design outline for implementing Transform in a distributed environment. Our architecture for Transform is strongly influenced by the identity-based capability architecture proposed by Gong [7]. The concept of embedding the identity of a subject in a capability in distributed systems has been known for some time [4]. It ensures that capabilities cannot be forged or propagated from one subject to another without intervention of trusted software. Gong's architecture is based on the familiar client-server model of services in a distributed system. It includes mechanisms for revocation which were missing in earlier proposals such as [4]. We have extended Gong's proposal to accommodate Transform. In particular the concept of strongly typed subjects and objects, which is essential to Transform, has been incorporated.

The rest of the paper is organized as follows. Section 2 discusses several examples of internal and external transformation in an informal manner. Section 3 develops the Transform model to unify, and make precise, the common theme running through these examples. This formalization in turn suggests additional applications. Section 4 describes our capability-based architecture and general design for implementing Transform in a distributed environment. The protocols involved in creation, propagation and revocation are presented. An example of the implementation is presented in section 5. In section 6 we digress from the main theme of the paper to discuss the safety implications of Transform and show that it has efficiently decidable safety. Section 7 concludes the paper.

2. Applications

The simplest example of transformation of rights arises when one right is treated as stronger than another. Consider the typical read, write and append operations on a file, respectively authorized by the rights r, w and a. From the semantics of these operations it is clear that possession of w should imply possession of a. The ability to obtain a weaker right by virtue of possessing a stronger one allows a subject to work with the least privileges needed at any given moment. In some cases we require the stronger implication that w implies a and both imply r.[2] 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.

We can generalize these examples somewhat by allowing different implication relations for different types of files. For instance, we may define two types of files respectively with the two implication

[2]This is of course appropriate only in situations where nondisclosure is not an issue.

relations discussed above and a third type of file with no implied rights. However, so long as the ability to obtain implied rights is uniformly available to every subject, internal transformation provides only for consistency in authorization.

Significant power is added by restricting internal transformation to certain subjects. The amplification operation in the Hydra system [3] works in such a fashion, as the basis for implementing abstract data types and protected subsystems. To illustrate amplification consider the example of a stack with push and pop operations implemented in terms of a segment with read and write operations. We need to enforce the following policy:

(1) Subjects other than the type manager for stacks can only possess push and pop rights for a stack.

(2) The type manager for stacks receives the right to push (or pop) a stack when a subject executes the push (or pop) operation. The manager amplifies the push (or pop) right to obtain r and w rights for the segment containing the stack.

(3) Only the type manager for stacks can do such internal transformation.

Predicating the ability to amplify on the type of subject doing the internal transformation enables implementation of abstract data types. Pursuing the example further, we may have stacks implemented in terms of lists which in turn are implemented in terms of segments. Now we have two levels of internal transformation. The first level from push or pop rights (i.e., stack operations) to the head, tail or cons rights (i.e., list operations) can only be done by the type manager for stacks. The second level from head, tail or cons rights (i.e., list operations) to r and w rights (i.e., segment operations) can only be done by the type manager for lists.

Next consider grant transformations. A simple form of grant transformation occurs with the copy flag, which distinguishes between the ability to access an object and the ability to grant access for that object to another subject. The concept goes back to the earliest abstract models for access control [8, 12] and is a fundamental aspect of discretionary access controls. The idea is that possession of a right x authorizes access to the object, whereas possession of xc authorizes the ability to grant access to that object to another subject. The xc right is typically made available to the creator of each object. In many models [8, 12, 15, for instance] 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 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.

Let us call the copy flag in the first case the *one-step copy flag*, denoted xc^1, and in the second case the *unlimited copy flag*, denoted xc^*. Both these copy flags were proposed in the original access-matrix papers [8, 12]. The interpretation is that xc^* can be transformed to xc^*, xc^1 or x during a grant, whereas xc^1 can only be transformed to x. This idea can easily be generalized to allow for *n-step copy flags* by allowing the grant transformation of xc^n to any one of xc^{n-1}, xc^{n-2},...,xc^1 or x. The interpretation of copy flags can also be made to depend on the types of subjects and objects involved in a grant operation. For instance, the copy flag can be interpreted as a one-step flag for sensitive documents, whereas for non-sensitive documents it is an unlimited flag. As another example, say we distinguish members of a department from outsiders with the policy that the copy flag for grants between members is transformed as an unlimited flag, whereas for grants from a member to an outsider it is transformed as a one-step flag. These are very reasonable policies. It is clear that the possibilities are endless, particularly in large systems with lots of subject and object types.

Next we introduce a new kind of copy flag, called the *separation copy flag*, by dropping the assumption that possession of xc implies possession of x. In this way we draw a clear separation between the ability to grant access and the ability to perform access. This separation has been suggested by Moffett and Sloman [19] as a fundamental aspect of "commercial" access-control policies. They note such separation is implemented in actual systems, citing the example of IBM's RACF. In our framework this separation is easily achieved as an instance of grant transformation where xc can only be transformed to x. Now if a subject is allowed to grant to itself the intent of the separation is defeated, since then possession of xc implies possession of x by a grant to onself. We can prevent this by predicating the grant transformation on the types of subjects involved. Say we distinguish security-officers from users. The transformation of xc to x is allowed in a grant from a security-officer to a user. However, in a grant from a security-officer to a security-officer the transformation is from xc to null. This is the policy suggested in [19]. There is the further question of how the ability to grant is obtained in the first place by security-officers. Following [19], this itself can be obtained by grant transformation. The idea is that some user owns the object in question. By possessing the own right for that object the user is authorized to grant xc (by transformation) to a security-officer. That is, the owner of an object can delegate the ability to grant access to security-officers. We can play this game again and ask how ownership is acquired. It should be clear by now that this in turn can be achieved by grant transformation if so desired. Alternatively it can be tied to creation of the object or be determined at system initialization.

More general notions of separation of duties can also be viewed as examples of grant transformations to some extent. These relate to sequences of operations on an object which must occur in a prescribed order and must be executed by different types of subjects. For example, consider a policy in which a check is prepared by a clerk, approved by a supervisor and issued by a cashier. This is separation of duties in that the different steps are to be executed by users with different roles (types). We can enforce this policy by transforming the prepare right into an approve right in the clerk-to-supervisor grant, and again transforming the approve right to an issue right in the supervisor-to-cashier grant.[3]

3. The Transform model

It is apparent from the foregoing discussion that there is a common theme underlying the several examples we have seen. Our objective in this section is to make this intuition precise by means of a formal model called Transform.

The notion of type is fundamental to most examples we have considered. In fact much of the power of transformation derives from predicating the ability to transform on the types of subjects and objects involved. We therefore assume that subjects and objects are classified into types. Object types identify classes of objects which are treated differently for transformation of rights. Subject types similarly identify classes of subjects which have varying ability to transform rights. Subject types also abstract the concept of roles often used in the literature [19, 26, 30, for instance].

[3]Note that separation of duties achieved in this way is limited to separation among roles. Consider the modified policy that the check be issued by a clerk, rather than a cashier, with the stipulation that the issuing clerk be different from the one who prepared the check. Controls based solely on types of subjects and objects cannot handle such cases. See [26, 28] for a mechanism which also deals with intra-type separation.

We define the sets TS and TO for subject types and object types respectively. Each subject is an instance of some subject type and each object an instance of some object type. We assume strong typing in that the type of subject or object is determined when it is created and does not change thereafter.

Before considering transformation of rights let us first deal with creation. It is clear subjects need to create objects. There are two issues involved in creation. Firstly, subjects need authorization to create objects. Secondly, the rights obtained as a result of creation also need to be specified.

In Transform we authorize creation of objects by means of a can-create function as follows:[4]

cc: TS $\longrightarrow 2^{TO}$

The interpretation of $cc(u)=\{o_1,o_2,\ldots,o_k\}$ is that subjects of type u are authorized to create objects of types $o_1,o_2,\ldots,o_k$.

The effect of creation is defined by create-rules of the following form, where R is the set of rights:

cr: TS $\times$ TO $\longrightarrow 2^{R}$

The interpretation is that when subject U of type u creates an object O of type o the creator U obtains the rights cr(u,o) for O. For example, if cc(user) = {file} and cr(user,file) = {own,r,w} the creator of a file gets the own, r and w rights for it. For readability we usually drop the set parentheses around singleton sets, for instance cc(user) = file.

Now consider the authorization for internal transformation. As discussed earlier, internal transformation of rights for an object in a subject's domain involves consideration of their types. So what we need is an internal transformation function of the following form:

itrans: TS $\times$ TO $\times$ R $\longrightarrow 2^{R}$

The interpretation of *itrans*(u,o,x) = $\{x_1,\ldots,x_n\}$ is that a subject of type u who has the x right for an object of type o can obtain the $x_1,\ldots,x_n$ rights for that object by internal transformation. For example, the policy that write implies append and both imply read can be stated in either of the following ways:

(a) *itrans*(user,file,w) = {a,r}

itrans(user,file,a) = r

itrans(user,file,r) = ϕ

(b) *itrans*(user,file,w) = a

itrans(user,file,a) = r

itrans(user,file,r) = ϕ

In (a) the transformation from w to r is achieved directly, whereas in (b) it is done indirectly in two steps. We allow for either formulation in the model. The amplification example of a stack implemented by a list which in turn is implemented by a segment can be specified as follows:

itrans(stack-manager, stack, pop) = {head, tail}

itrans(stack-manager, stack, push) = cons

itrans(list-manager, stack, head) = {r, w}

itrans(list-manager, stack, tail) = {r, w}

itrans(list-manager, stack, cons) = {r, w}

All other values of *itrans* are empty

Here the ability to amplify push and pop to head, tail or cons is restricted to the stack manager; while amplification from head, tail and cons to r and w is restricted to the list manager. Realistically of course these would be fragments of a larger specification involving additional types.

[4]The notation 2^X denotes the power set of X, i.e., the set of all subsets of X. In other words cc is a function which maps each subject type to a subset of the object types.

The internal transformation function generalizes in an obvious way as follows to amplify sets of rights (as opposed to single rights):

itrans: $TS \times TO \times 2^R \longrightarrow 2^R$

The interpretation of *itrans*$(u,o,\{x_1,\ldots,x_n\}) = \{y_1,\ldots, y_m\}$ is that a subject of type u who has *all* the x_i rights specified on the left-hand side for an object of type o can obtain the rights $y_1,\ldots,y_m$ for that object by internal transformation. This is useful in situations described as synergistic authorization in [17] and as command authorization in [9]. For instance, consider a situation where a scientist (abbreviated as sci) needs approvals from a security-officer and a patent-officer before he can release a document (abbreviated as doc) for publication. Say these two approvals are respectively signified by possession of the a_s and a_p rights. We can express this policy as follows:

itrans$(\text{sci},\text{doc},\{\text{own},a_s,a_p\}) = \text{release}$

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.

Grant transformations can be modeled as a *grant* function of the following form:

grant: $TS \times TS \times TO \times R \longrightarrow 2^R$

The interpretation of *grant*$(u,v,o,x) = \{x_1,\ldots,x_n\}$ is that a subject of type u who has the x right for an object of type o can grant *one or more* of the $x_1,\ldots,x_n$ rights for that object to a subject of type v. The unlimited copy flag xc* and the one-step copy flag xc^1 of section 2 can then be specified as follows:

grant(user, user, file, xc*) $= \{xc^*, xc^1, x\}$

grant(user, user, file, xc^1) $= x$

grant(user, user, file, x) $= \phi$

The extension to *n*-step copy flags is obvious. There are actually several ways of specifying even this rather simple policy. For instance, we could combine grant and internal transformations to achieve the same net effect as follows:

grant(user, user, file, xc*) $= \{xc^*, xc^1\}$

itrans(user, file, xc*) $= xc^1$

grant(user, user, file, xc^1) $= x$

grant(user, user, file, x) $= \phi$

This property of multiple equivalent specifications appears to be inevitable in any sophisticated security model. We cannot realistically hope to have a unique, or even a best, specification for a particular policy in a general model.

The separation copy flag of section 2 is also easily specified as follows:[5]

grant(user, security-officer, file, own) $=$ xc

grant(security-officer, user, file, xc) $=$ x

itrans(security-officer, file, xc) $= \phi$

That is, a user who owns a file can delegate the authority to grant access to that file to a security-officer. The security-officer can grant access to that file to other users but cannot himself access it.

Next let us go back to the example of a scientist who needed multiple approvals for releasing a document for publication. We had mentioned that consideration of grants is required for a complete statement. One possibility is shown below:

grant(sci, security-officer, doc, own) $=$ review

grant(sci, patent-officer, doc, own) $=$ review

[5]We generally assume that all values of *grant* and *itrans* which are not explicitly defined are empty. In this case we have explicitly shown that *itrans* (security-officer, file, xc) $= \phi$ because of its importance in specifying this policy.

grant(security-officer, sci, doc, review) = a_s

grant(patent-officer, sci, doc, review) = a_p

itrans(sci, doc, {own, a_s, a_p}) = release

As the owner of a document a scientist can request it be reviewed by a security-officer and a patent-officer by granting them the review right. In turn they can grant the scientist who gave them the review right appropriate approval rights. Finally the scientist can internally transform these rights to acquire the release right.

Consider a slight modification to the above policy. Say that we require further separation of duties regarding release of a document. A scientist is responsible for gathering the necessary approvals. The actual release, however, must be done by a librarian who is responsible for cataloging information about the document before releasing it. To achieve this we can replace the internal transformation above by the following grant transformation:

grant(sci, librarian, doc, {own, a_s, a_p}) = release

To do so we can generalize *grant* as follows in the same way that *itrans* was generalized:

grant: $TS \times TS \times TO \times 2^R \longrightarrow 2^R$

The interpretation of *grant*(u,v,o,{$x_1,\ldots,x_n$}) = {$y_1,\ldots, y_m$} is that a subject of type u who has *all* the x_i rights specified on the left-hand side for an object of type o can grant *one or more* of the rights $y_1,\ldots,y_m$ for that object to a subject of type v.

To summarize, we have the following definition for Transform.

Definition 1 A policy for transformation of rights is stated in Transform by specifying the following (finite) components:

(1) Disjoint sets of subject types TS and object types TO.

(2) A set of rights R.

(3) A can-create function *cc*: $TS \longrightarrow 2^{TO}$.

(4) Create-rules *cr*: $TS \times TO \longrightarrow 2^R$.

(5) An internal transformation function *itrans*: $TS \times TO \times 2^R \longrightarrow 2^R$.

(6) A grant transformation function *grant*: $TS \times TS \times TO \times 2^R \longrightarrow 2^R$.

This completes our description of Transform.

4. Implementation of Transform

In this section we describe an architecture and design outline for implementing Transform in a distributed environment. Our architecture is capability based. We begin with a brief review of distributed capability systems, following which we describe our architecture and protocols in detail.

4.1. Distributed capability systems

Capability-based architectures have had a strong appeal ever since the concept was first proposed [5]. They are viewed as providing a sound and common basis for providing both reliability and security [14]. In the context of conventional centralized systems a number of such machines have been built [13]. Some even achieved moderate commercial success. Nevertheless today's popular CPUs are not capability based. In retrospect one can argue that using capabilities to solve the memory protection problem is an overkill. The marginal advantages of capabilities over memory segmentation and protection rings, which are available in the latest generation of microprocessors such as the Intel 80386, do not justify the extra costs and performance penalties. In other words the initial application of capabilities was at too low a level.

It is expected by many researchers that in the 1990s distributed operating systems will dominate the computing environment. These systems will appear to users as a single centralized system with complete location transparency. To achieve this, reliability and security will have to be addressed as part of the basic design of these systems. Attempts to graft security features later in the design cycle will surely fail much as they are failing in conventional centralized systems. The capability-based framework continues to offer an attractive approach to these problems. In a distributed operating system capabilities are introduced at a much higher level than memory addressing. Capabilities need to be incorporated into the remote procedure call mechanism rather than the memory addressing mechanism. This offers the hope that the additional overhead will not kill performance. Capabilities can moreover be integrated into the basic client-server structure of distributed systems to provide transparency.

There are three basic issues which must be confronted by the designer of a distributed capability-based system. These issues are complicated relative to conventional centralized capability-based systems because capabilities are dispersed in individual workstations and can no longer be assumed to be under tight control of a centralized security kernel.

(1) *Unforgeability.* It must be guaranteed that capabilities cannot be modified or manufactured by subjects. This requires some form of cryptographic sealing.

(2) *Propagation.* It must be guaranteed that capabilities cannot be copied from one subject to another. This requires some means of embedding the identity of a subject in a capability.

(3) *Revocation.* It must be guaranteed that capabilities which have been granted can be withdrawn or revoked in a timely manner. This requires some means of invalidating existing capabilities and accounting for cascaded revocation.

Various solutions to one or more of these problems have been proposed in the literature. For instance, Amoeba [20] uses "sparse capabilities" with cryptographic protection to ensure unforgeability. Unfortunately Amoeba does not address propagation or revocation. Davies [4] discusses mechanisms to embed the identity of a subject in a capability. This ensures that capabilities cannot be forged or propagated from one subject to another without intervention of trusted software. Davies, however, does not address the revocation issue. Gong's proposed architecture [7] is the first attempt to address all three issues in a distributed context. It is based on the familiar client-server model of services in distributed systems and therefore is a suitable foundation for us to build upon. However, Gong does not incorporate the notion of types which is basic to Transform. His architecture therefore needs to be extended for this purpose.

4.2. Basic architecture for Transform

We assume that objects are encapsulated within object servers. The basic computation model is that of remote procedure calls involving the following sequence of events: (i) a client sends a request to a server to manipulate one or more objects, (ii) the server accepts and services the request, and (iii) the server sends back a reply. The object server runs on a trusted host which guarantees that the server cannot be bypassed. For ease of exposition we visualize each object server as running on a separate host. However, we allow multiple object servers on the same trusted host provided the security kernel on the host can enforce separation among these servers. If we have sufficient confidence in the security kernel we can also allow untrusted clients to coexist with object servers on a single trusted host.

Each object server acts as the reference monitor (or access mediator) for the set of objects it manages. In other words the object server is part of the Trusted Computing Base [6]. The object server is responsible not only for access mediation but also for ensuring semantic correctness of the objects with respect to the abstract operations exported from the server. The object server itself has the ability to access all objects within its control. We emphasize that the object server is not a subject in the system but is rather a part of the Trusted Computing Base.

For simplicity, we require that each object server manage exactly one type of object. In practice this rule would probably be relaxed to allow a single server to manage multiple object types, particularly if they are closely related. On the other hand the same type of object may be managed by multiple object servers. For instance, a given system may have numerous file servers. An individual file server manages some subset of the total collection of files in the system. We assume there is no replication of files; that is, each file resides at exactly one file server.

Finally we assume there is an Access Decision Facility which can be consulted by object servers to determine the security policy. In the context of Transform the Access Decision Facility will be consulted by object servers for finding out appropriate values of *cc*, *cr*, *grant* and *itrans*. Pieces of the Access Decision Facility may actually reside at each object server while other pieces are remotely accessed. The reason for this is to allow quick local access to well-established and relatively static aspects of the policy while at the same time allowing for new types etc. to be introduced.

4.3. Identity and type

Each subject or object in the system has a globally unique identifier. Each subject or object also has a unique type which is determined when that subject or object is created. Thereafter the type cannot change. We assume the type of a subject or object is embedded in its identifier. Henceforth we refer to a subject identifier by *sid* and an object identifier by *oid*. These identifiers have the following structure:

type	identifier

The type field denotes the type of the object while the identifier field uniquely identifies each subject or object among instances of the same type. Note that sid's and oid's can be generated at will by users and have no guarantee of unforgeability. We refer to the individual fields of a sid by sid.type and sid.identifier, and similarly for oid's. Note that the sid's and oid's must be globally unique, for which purpose it suffices that their identifier fields are unique within instances of the same type.

4.4. Capability seeds

A capability seed is a secret random number associated with each oid. The seed is known only to the object server which manages the object identified by oid. We can visualize this association by the following pair:[6]

oid	seed

The purpose of the seed is to facilitate revocation and prevent against replay of revoked capabilities, as will be discussed later.

4.5. Capabilities

A capability has the following structure:

oid	rights	seal

where the seal is computed using a publicly known one-way function f as follows:

seal = f(sid, oid, rights, seed)

The oid and rights components of a capability are exactly as one would expect, even in a conventional centralized system. The seal cryptographically embeds the subject identifier (sid) in the capability using the secret capability seed for that purpose.

[6]Gong [7] calls this pair an "internal capability." We feel the name "internal capability" is a misnomer and prefer to call the secret random number a capability seed, because its principal use is in cryptographically sealing capabilities exported from the object server.

4.6. Access mediation

Access mediation 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. For this purpose, we assume that each subject has the means to place its digital signature on every RPC communication to an object server. The RPC also carries within it the relevant capabilities for the operation being requested. The object server first verifies that the sid on each capability is authenticated by the digital signature, otherwise the RPC is immediately rejected. Then the object server looks up the capability seed for oid, computes the seal using the above formula and compares the computed seal with the seal submitted by the subject. If these match, the capability is known to be authentic and the operation is performed, provided the rights are sufficient to authorize it.

Digital signatures for the reverse communication from object servers to subjects can also be incorporated. The details of these protocols are beyond the scope of this paper and can readily be found in the standard literature [1]. We envisage an implementation similar to the interface function box of Amoeba [20] which is placed between each processor module and the network.

4.7. Creation

For object creation the object server consults the Access Decision Facility to determine whether or not such creation is authorized by *cc*(sid.type). If the creation is authorized a new object is created with a new oid and a new capability seed. The rights to be entered on the capability are determined from *cr*(sid.type, oid.type). Finally the capability is sealed and returned to the subject.

4.8. Internal transformation

Let subject sid request the following internal transformation for object oid:

$itrans(u,o,\{x_1,\ldots,x_n\}) = \{y_1,\ldots,y_m\}$

The object server must, of course, be a manager for objects of type o. The server checks that sid.type = u and oid.type = o. It also checks that the RPC request includes a capability (or capability list) for object oid with the rights $x_1,\ldots,x_n$. This check is performed by comparing the computed seal with the seal on the capability as discussed in section 4.6. Finally the object server creates a new capability sealed for sid with rights $x_1,\ldots,x_n$, $y_1,\ldots,y_m$. This capability is returned to the subject sid. Note that the original capability continues to be valid. It is, however, redundant and can be discarded by the subject.

4.9. Grant transformation

Let subject sid1 request the following grant transformation for object oid to subject sid2:

$grant(u,v,o,\{x_1,\ldots,x_n\}) = \{y_1,\ldots,y_m\}$

The object server should again be a manager for objects of type o. The server checks that sid1.type = u, sid2.type = v and oid.type = o. It also checks that the RPC request includes a capability (or capability list) for object oid with the rights $x_1,\ldots,x_n$. If the check is successful the object server creates a new capability sealed for sid2 with rights $y_1,\ldots,y_m$. This capability is returned to the subject sid1, who can then pass it on to subject sid2. (Alternatively it can be communicated to sid2 directly.)

4.10. Revocation

Revocation has always been a problem in capability-based systems. In distributed systems the problem is compounded, since the subjects are completely autonomous, with no centralized authorities enforcing security. There are various issues with respect to which implementations of revocation can be compared [29]:

(1) Partial or Complete: whether it is possible to revoke a specific right or whether all rights in a capability 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?

We provide revocation by a revocation list and a count field appended to the seed as shown below:

oid	seed	count	revocation list

The revocation list contains entries of sids for whom the rights for that particular oid have been revoked. The list specifies for each sid which of its rights have been revoked. When the validity of the capability is checked during access mediation, the revocation lists are checked in parallel as well. Since access mediation is performed on every operation revocation is immediate. The owner of an oid always has the option to revoke partially or completely the capability of a sid for that oid. Partial or complete revocation of a sid in no way interferes with the access rights of other sids.

The count is a measure that determines the number of valid capabilities for that seed. The count is incremented during creation and propagation, but decremented during complete revocation (i.e., when all the rights of a subject for that object are revoked). Temporary or permanent revocation is carried out, depending on the value of the count. If the count is smaller than a threshold the object server goes ahead with permanent revocation. The server deletes the seed associated with that oid, computes a new one and sends new recomputed capabilities to other associated sids. This of course requires that the object server keep a log of propagation of capabilities. However, if the count is above the threshold the object server goes ahead with temporary revocation. In this case the object server appends the revocation information onto the revocation list associated with that oid. The value of the threshold is set by the system administrator.

5. Implementation of an example

The scientist and the security-officer example discussed earlier in section 3 is illustrated here using the protocols described above. A scientist (say Joe) creates a document (say SDI) on his workstation, but before he can release it he needs to have approval from a security-officer (say Sam) and a patent-officer (say Pat). The following is the sequence of protocols needed to complete the task.

(1) Joe asks the server to create a document called SDI. This RPC is made by the kernel of Joe's workstation to the appropriate daemon responsible for that host's actions. RPC contains the action requested and the sid of the requester together signed under Joe's digital signature. In this case the sid = sci.Joe and the request is to create a new document of type doc with specified contents. On receiving the request, server checks the digital signature to authenticate Joe. The server then checks the *cc* policy, taking into account sid.type. If it is in the affirmative it checks the *cr* policy, by which it determines what rights Joe gets for the document he is creating. The document server generates a new oid for the document being created (say doc.SDI) as well as a new seed (say seed1) for that document. The server sets the count to 1 and the initial revocation list to empty and stores the information in its internal tables with the following association.

doc.SDI	seed1	1	

The revocation list field is empty as there are no entries for it and we shall not show it till it is needed. So from here on it can be assumed that the revocation list, if missing, is empty.

Then the object server manufactures the following capability and sends it to Joe (strictly speaking to the kernel of Joe's workstation):

doc.SDI	own, read	seal1

where seal1 = f(sci.Joe, doc.SDI, {own, read}, seed1).

(2) Now Joe is ready to release the document. His workstation sends the propagation requests to the server on his behalf. The RPC looks like this:

grant(security-officer.Sam, review)

doc.SDI	own, read	seal1

The host, when framing the RPC, appends to it the capability that Joe possesses for SDI and signs the request under Joe's digital signature. The server on receiving the request decrypts the digital signature and authenticates Joe. Then the server checks the validity of the capability by retrieving the seed of SDI (i.e., seed1) from its internal tables, and computing the seal using the one-way function f. Then it extracts seal1 from the capability provided by Joe and if the two seals match the validity of the capability is confirmed. The request is then checked against the *grant* function. When the server determines Joe has sufficient rights (i.e., own) for SDI, it proceeds with the grant. In its internal tables the count is updated to 2, which looks like this:

doc.SDI	seed1	2

The server then computes the capability for the security-officer Sam to have the review right for SDI. The capability

doc.SDI	review	seal2

where seal2 = f(security-officer.Sam, doc.SDI, review, seed1)

is sent to Joe. Joe then forwards this capability to Sam. Sam now has the capability for oid = doc.SDI with the review right. With this capability he can only access the document to review it. If Sam tries to get additional rights by internal transformation, the server will turn down his request because the set of rights, namely review, is an insufficient set for any internal transformation. Sam now reviews the document, and if he approves of the action to release SDI he requests the server to grant Joe the approval (a_s) right.

grant(sci.Joe, a_s)

doc.SDI	review	seal2

The server, as before, updates the count in the internal table to 3:

doc.SDI	seed1	3

and then computes the following capability and sends it back to Sam, who in turn sends it to Joe:

doc.SDI	a_s	seal3

where seal3 = f(sci.Joe, doc.SDI, a_s, seed1).

(3) Exactly similar protocol steps are executed to get the approval (a_p) from the patent-officer Pat. At the end of this session the internal table looks like this:

doc.SDI	seed1	5

And Joe possesses the following capability:

doc.SDI	a_p	seal4

where seal4 = f(sci.Joe, doc.SDI, a_p, seed1).

(4) Now the scientist Joe possesses the capabilities giving him the approval to get the release right by

internal transformation. Joe presents these capabilities to the server with the following request:

	doc.SDI	own, read	seal1
itrans(release)	doc.SDI	a_s	seal3
	doc.SDI	a_p	seal4

Like before, the server carries out the authentication and the validity tests on the capabilities presented to it by Joe. Then the server checks that Joe has the rights own, a_s and a_p for SDI which are required to get the additional release right. Count is not updated during internal transformation. The server sends him a new capability:

doc.SDI	own, read, a_s, a_p, release	seal5

where seal5 = f(sci.Joe, doc.SDI, {own, read, a_s, a_p, release}, seed1).

To exemplify revocation let us augment the *grant* function of the document release example of section 3, which we illustrated above, with

grant(sci, sci, doc, release) = read

That is, the release right allows the scientist to let other scientists read the document. Now assume that Joe grants scientist Jill (sid = sci.Jill) the read right for SDI. The protocols are the same as above. The signed RPC request is as follows:

grant(sci.Jill, read)

doc.SDI	own, read, a_s, a_p, release	seal5

In response to this request the server updates the count in the internal tables to 6:

doc.SDI	seed1	6

And then the server computes the following capability and passes it to Joe, who in turn passes it to Jill:

doc.SDI	read	seal6

where seal6 = f(sci.Jill, doc.SDI, read, seed1).

This capability gives Jill the authorization to read SDI.

Now if at a later time Joe wants to revoke the read privilege of Jill, he requests the server to execute the following action:

rev(sci.Jill, read)

doc.SDI	own, read, a_s, a_p, release	seal5

The server performs the various tests on the capability to check its authenticity and validity. Then the server looks at the value of the count in its internal tables for seed1. Let us assume threshold to determine the type of revocation (i.e., permanent or temporary) is 7. The server compares the value of the count with that of the threshold and decides with permanent revocation, as the value of the count (6) is less than the threshold. For each complete revocation the server decrements the count by one. So in this case the count will decrease to 5, as only one complete revocation is requested. The server then recomputes new capabilities for the doc.SDI with a new seed2 and the old ones are purged. This new association is shown below:

doc.SDI	seed2	5

And the new recomputed capability sent to Joe looks like this:

doc.SDI	own, read, a_s, a_p, release	seal7

where seal7 = f(sci.Joe, doc.SDI, {own, read, a_s, a_p, release}, seed2).

In similar fashion new recomputed capabilities are sent to all subjects which possess a capability for SDI. For this purpose the server maintains a list of all subjects who possess capabilities for SDI. All the seals on these new capabilities will be computed as before, except they will be a function of seed2 instead of seed1. With this all previous capabilities for SDI are invalidated and should be purged. And if Jill tries to access SDI with the capability she possesses her request will fail since the capability she possesses for SDI will fail the validity test.

To illustrate temporary revocation let us assume that the threshold is set at 4 and the count is 6. Joe requests revocation in a similar fashion as before and similarly the server compares the value of the threshold (4) to the count (6). The server decides with temporary revocation as the value of count is greater than the value of the threshold. For temporary revocation, the server adds the sid of the revoked subject with the revoked right to the revocation list in the internal table. The count is not decremented in temporary revocation. The internal table now contains the following association:

doc.SDI	seed1	6	sci.Jill{read}

On all future access mediations by Jill, when the server would check the revocation list for SDI, it will find her sid along with the list of revoked rights and thus deny read access.

Our examples demonstrate that fairly complicated policies arise in even rather simple situations. The examples have used a few types of subjects and objects. Realistically in large organizations we would have hundreds of types. The complexity will rapidly multiply. We believe that authorization policies will necessarily be formulated in terms of local and incremental considerations of the kind we have discussed. In such situations safety analysis is very important.

6. Safety analysis of Transform

The safety question for access control poses the following question: is it possible for a given subject to ever acquire access to a given object? It is well known that in general this question is undecidable [10], even for monotonic systems [11].

In this section we show that Transform has efficiently decidable safety. We do this by demonstrating that Transform is an instantiation of SPM (Schematic Protection Model) [24]. Our construction establishes that Transform can be simulated in SPM within SPM's efficiently decidable cases for safety.

One difficulty in reducing Transform to SPM is that the SPM copy operation is attenuating, whereas the Transform *grant* and *itrans* operations may be amplifying (i.e., new rights may be created rather than simply being copied from one subject to another). Section 6.1 shows that the *grant* operation in Transform can be assumed, without loss of generality, to be attenuating. Section 6.2 then shows how amplifying *itrans* operations can be simulated in SPM. It also contains a brief review of SPM.

6.1. Attenuating Transform

We now show that amplifying *grant* can be eliminated from the Transform model without any loss of expressive power.

It is clear that internal transformations are useful only if they are amplifying in the sense that new rights are obtained. That is, we can assume without loss of generality,

$$\textit{itrans}(u,o,R_i) = R_j \implies R_j \cap R_i = \phi$$

Now consider the grant transformation $\textit{grant}(u,v,o,R_i) = R_j$. That is, possession of R_i rights enables transfer of R_j rights. Clearly if $R_j \subseteq R_i$ such a grant is attenuating or non-amplifying in that the source subject cannot give away rights that he does not possess. But note that the source subject may be able to internally amplify the R_i rights, so in defining attenuation we need also to consider implied rights. Now implied rights can be obtained directly by one application of *itrans* or indirectly by several applications. This leads us to the following definition.

Definition 2 Let *itrans** be the reflexive transitive closure of *itrans*. A grant transformation is *attenuating* provided

$$\textit{grant}(u,v,o,R_i) = R_j \implies R_j \subseteq \textit{itrans}^*(u,o,R_i)$$

Otherwise it is *amplifying*.

For example, *grant*(user, user, file, x) = x is trivially attenuating. On the other hand for *grant*(user, user, file, xc) = x we need to consider the interpretation of the copy flag. With the assumption that xc is strictly stronger than x, i.e., *itrans*(user, file, xc) = x, the latter grant is attenuating. However for the separation copy flag, where we have *itrans*(user, file, xc) = ϕ, this grant is amplifying. This is clearly consistent with the intuitive concepts of amplification and attenuation.

One can take issue with this definition in that we are ignoring implied rights in the destination domain. That is, what we really need is the following requirement:

$$\textit{grant}(u,v,o,R_i) = R_j$$
$$\implies \textit{itrans}^*(v,o,R_j) \subseteq \textit{itrans}^*(u,o,R_i)$$

Let us call such grants *strictly attenuating*. This requirement is very strong and will not allow for the grants required to support abstract data types or protected subsystems, as illustrated by our stack example. These features are of such fundamental importance that it is clear we cannot limit ourselves to strictly attenuating grants in the framework of Transform.

The question therefore is whether or not we can limit ourselves to attenuating grants. In other words, do amplifying grants add any power not already available with amplifying internal transformations? The answer turns out to be no; that is, grant amplifications can be built out of internal amplifications. To see the redundancy of amplifying grants consider the separation copy flag specified earlier as follows:

grant(user, security-officer, file, own) = xc

grant(security-officer, user, file, xc) = x

These grants are clearly amplifying. An equivalent policy with attenuating grants is achieved by introducing new right symbols as follows:

itrans(user, file, own) = delegate

grant(user, security-officer, file, delegate) = delegate

itrans(security-officer, file, delegate) = xc

itrans(security-officer, file, xc) = cando-x

grant(security-officer, user, file, cando-x) = cando-x

itrans(user, file, cando-x) = x

The two amplifying grants of the original policy are respectively simulated by the two sequences above. The general principle is evident from this example. Each amplifying grant is simulated by an internal amplification at the source, followed by a grant with the trivial and attenuating identity transformation, finally followed by another internal transformation at the destination.

A general construction can be outlined as follows. Let $r \in \textit{grant}(u,v,o,R_i)$ and $r \notin \textit{itrans}^*(u,o,R_i)$. That is, r makes this grant amplifying. Modify the given Transform specification as follows:

(1) Define the new right $\underline{r.u.v.o.R_i}$. The entire symbol signifies a single right. The components in this symbol emphasize that we need a new right for each combination of the components.

(2) Modify *itrans*(u,o,R_i) to include $\underline{\text{r.u.v.o.R}_i}$.

(3) Modify *grant*(u,v,o,R_i) by replacing r with $\underline{\text{r.u.v.o.R}_i}$.

(4) Define *itrans*(v,o,$\underline{\text{r.u.v.o.R}_i}$) = r.

It is clear that r no longer makes this modified *grant*(u,v,o,R_i) amplifying. By repeating this procedure we can therefore get rid of all amplifying grants. Since new rights are introduced for each iteration of this procedure there is no interaction between different amplifications removed in this way. The original amplifying grants are then simulated as follows:

$$r \in \mathit{grant}(u,v,o,R_i)$$

$$\Longleftrightarrow \begin{cases} \mathit{itrans}(u,o,R_i) = \underline{\text{r.u.v.o.R}_i} \\ \underline{\text{r.u.v.o.R}_i} \in \mathit{grant}(u,v,o,R_i) \\ \mathit{itrans}(v,o,\underline{\text{r.u.v.o.R}_i}) = r \end{cases}$$

The correctness of this construction is self-evident. A formal inductive proof can be given showing every reachable state with the former policy has an equivalent counterpart with the modified policy, and vice versa. The details are tedious and shed little insight.

6.2. Reduction of Transform to SPM

The previous subsection has shown that we can assume, without loss of generality, that all grant transformations are attenuating. Attenuating grant transformations are easily simulated by the links and filter functions of SPM. It remains to consider how internal transformations—attenuating or amplifying—can be simulated by SPM copy operations. We first give a brief review of SPM, followed by the reduction of Transform to SPM.

6.2.1. The schematic protection model

We begin with a review of SPM. Our review is necessarily brief and to the point. Motivation for defining SPM in this manner and its resulting expressive power are discussed at length in [23–25].

The dynamic privileges in SPM are tickets of the form Y/x, where Y identifies some unique entity (subject or object) and x is a right. SPM subjects and objects are strongly typed. The type of a ticket is determined by the type of entity it addresses and the right symbol it carries; that is, type(Y/x) is the ordered pair type(Y)/x. Tickets are acquired in accordance with rules which comprise the scheme, which is defined by specifying the following (finite) components. These are briefly explained below:

(1) Disjoint sets of subject types TS and object types TO. Let T = TS ∪ TO.

(2) A set of rights R. The set of ticket types is thereby T × R.

(3) A can-create function *cc*: TS → 2^T.

(4) Create-rules of the following form for each v ∈ *cc*(u): $cr_p(u,v) = c/R_1 \cup p/R_2$, and $cr_c(u,v) = c/R_3 \cup p/R_4$.

(5) A collection of link predicates {link_i}.

(6) A filter function f_i: TS × TS → $2^{T \times R}$ for each predicate link_i.

There are only two operations in SPM: create and copy. These are controlled by the scheme as follows.

The create operation

Subjects of type u can create entities of type v if and only if v ∈ *cc*(u). Tickets introduced as the side effect of creation are specified by a (different) create-rule for every (u,v) such that v ∈ *cc*(u). Each create-rule has two components shown above, where *p* and *c* respectively denote parent and child and the R_i's are subsets of R. When subject U of type u creates entity V of type v the parent U gets the tickets V/R_1 and U/R_2. The child V similarly gets the tickets V/R_3 and U/R_4. For example, file ∈ *cc*(user) authorizes users to create files. And cr_p(user, file) = *c*/rw and cr_c(user, file) = ϕ gives the creator r and w tickets for the created file.

The copy operation
A copy of a ticket can be transferred from one subject to another, leaving the original ticket intact. SPM has a copy flag built in which we denote as k to distinguish it from the copy flags of Transform. Possession of Y/xk implies possession of Y/x but not vice versa. Let dom(U) signify the set of tickets possessed by U. Let x:k denote x or xk, with multiple occurrences in the same context either all read as x or all as xk. Three independent pieces of authorization are required to copy Y/x:k from U to V:

(1) $Y/xk \in \text{dom}(U)$; that is, U must possess Y/xk for copying either Y/xk or Y/x.

(2) There is a link from U to V. Links are established by tickets for U and V in the domains of U and V. The predicate $\text{link}_i(U,V)$ is defined as a conjunction or disjunction, but not negation, of one or more of the following terms for any $z \in R$: $U/z \in \text{dom}(U)$, $U/z \in \text{dom}(V)$, $V/z \in \text{dom}(U)$, $V/z \in \text{dom}(V)$, and **true**. Some examples of link predicates from the literature are given below:

$$\text{link}_{tg}(U,V) \equiv V/g \in \text{dom}(U) \vee U/t \in \text{dom}(V)$$
$$\text{link}_t(U,V) \equiv U/t \in \text{dom}(V)$$
$$\text{link}_{sr}(U,V) \equiv V/s \in \text{dom}(U) \wedge U/r \in \text{dom}(V)$$
$$\text{link}_u(U,V) \equiv \textbf{true}$$

The first example is from the take-grant model [15], where the t and g rights are respectively read as take and grant. The next example retains only the take right [16]. The fourth example is from the send-receive mechanism [18, 22] where the s and r control rights are respectively read as send and receive. The last case is unique in that it requires no tickets for a link to exist.

(3) The final condition is defined by the filter functions f_i: $TS \times TS \rightarrow 2^{T \times R}$, one per predicate link_i. The value of $f_i(u,v)$ specifies types of tickets that may be copied from subjects of type u to subjects of type v over a link_i. Example values are $T \times R$, $TO \times R$ and ϕ, respectively, authorizing all tickets, object tickets and no tickets to be copied.

In short, Y/x:k can be copied from U to V if and only if

$$Y/xk \in \text{dom}(U) \wedge (\exists \text{link}_i)[\text{link}_i(U,V) \wedge y/x{:}k \in f_i(u,v)]$$

where the types of U, V and Y are respectively u, v and y. Note that f_i determines whether or not the copied ticket can have the copy flag. This completes our review of SPM.

6.2.2. Simulation of Transform in SPM
As noted earlier, attenuating grant transformations are easily simulated by the links and filter functions of SPM. It remains to consider how internal transformations—attenuating or amplifying—can be simulated by SPM copy operations.

Consider $itrans^*(u,o,R_i) = R_j$. Let U be a Transform subject of type u and O a Transform object of type o. These are respectively modeled as SPM subjects of types u and o respectively. Let the possession of O/R_i by U set up a link_{R_i} from O to U. The internal transformation is effected by defining $f_{R_i}(u,o)$ to be o/R_j. The scheme ensures that the only tickets that O can ever possess are tickets for itself, so the copy operation authorized in this manner has precisely the same effect as the internal transformation. The SPM copy flag is irrelevant to the construction and we assume it is allowed to be carried along by every filter function we define.

This construction is formally expressed by the following SPM scheme for a given instance of Transform, which is assumed (without loss of generality) to have attenuating *grant*'s:

(1) $TS' = TS \cup TO$, $TO' = \phi$

(2) $R' = \{x{:}k \mid x \in R\}$

(3) For all $u \in TS$: $cc'(u) = cc(u)$
For all $o \in TO$: $cc'(o) = \phi$

(4) $cr_p'(\mathrm{u,o}) = c/\mathrm{R}_1$, where $cr(\mathrm{u,o}) = \mathrm{R}_1$

$cr_c'(\mathrm{u,o}) = c/\mathrm{R}'$

(5) Define the following link predicates:

$\mathrm{link_u(U,V)} \equiv \mathbf{true}$

$\mathrm{link}_{\mathrm{R}_i}(\mathrm{O,U}) \equiv \mathrm{O/R}_i \in \mathrm{dom(U)}$, for all $\mathrm{R}_i \subseteq \mathrm{R}$

(6) Let $\mathrm{R_jk}$ denote $\{\mathrm{xk} \mid \mathrm{x} \in \mathrm{R_j}\}$

Define $f_{\mathrm{R}_i}(\mathrm{o,u}) = \mathrm{o/R_jk}$, where $itrans^*(\mathrm{u,o,R}_i) = \mathrm{R_j}$

Define $f_{\mathrm{u}}(\mathrm{u,v}) = \{\mathrm{o/R_jk} \mid (\exists \mathrm{R}_i)\ \ grant(\mathrm{u,v,o,R}_i) = \mathrm{R_j}\}$

All other values of the filter functions are empty

The simulation can be summarized as follows:

$itrans^*(\mathrm{u,o,R}_i) = \mathrm{R_j}$

$$\Longleftrightarrow \begin{cases} \mathrm{link}_{\mathrm{R}_i}(\mathrm{O,U}) \equiv \mathrm{O/R}_i \in \mathrm{dom(U)} \\ \qquad f_{\mathrm{R}_i}(\mathrm{o,u}) = \mathrm{o/R_jk} \end{cases}$$

$\mathrm{r} \in grant(\mathrm{u,v,o,R}_i)$

$$\Longleftrightarrow \begin{cases} \mathrm{link_u(U,V)} \equiv \mathbf{true} \\ \quad \mathrm{o/rk} \in f_{\mathrm{u}}(\mathrm{u,v}) \end{cases}$$

An internal transformation is replaced by a subject copying the transformed tickets from the object's domain. For grant transformations we have earlier shown that we can assume $\mathrm{r} \in \mathrm{R}_i$ so they are reduced to copying a ticket over the universal link. Formal correspondence between the original Transform policy and the constructed SPM scheme can be established by a straightforward inductive proof that the reachable states in both cases are equivalent.

It remains to argue that this construction establishes that safety is efficiently decidable for Transform. This follows from the result for SPM [24] that safety is decidable provided *cc* is acyclic in the following sense: the directed graph with edges $\{(\mathrm{u,v}) \mid \mathrm{v} \in cc(\mathrm{u})\}$ is acyclic. Since the only edges in this graph for *cc'* are from types in TS to types in TO, *cc'* is trivially acyclic. Moreover this graph for *cc'* is sparse, which guarantees that the decision procedure is efficient [24] (i.e., has low-degree polynomial complexity).

7. Conclusion

To summarize, we have described a wide variety of access-control mechanisms from the literature with the common theme of transformation of access rights. We have unified these mechanisms in a simple model called Transform.

We have described a distributed capability-based architecture for implementing Transform. The architecture is based on object servers who act as access-mediators on any attempt by a subject to create, use, acquire, grant or revoke capabilities. Each object server runs on a trusted host which guarantees that the server cannot be bypassed and therefore is a reference monitor for the objects that it manages. The object server is not a subject in the system but is rather a part of the Trusted Computing Base.

The basic computation model is that of remote procedure calls involving the following sequence of events: (i) a client sends a request to a server to manipulate one or more objects; (ii) the server accepts and services the request; and (iii) the server sends back a reply. We assume a digital signature facility which authenticates the originating subject on each remote procedure call. The capabilities are cryptographically sealed to tie together the identity of the subject, the identity of the object, the rights and a secret cryptographic seed. Strong typing of subjects and objects has also been incorporated.

Finally we have shown that Transform has efficiently decidable safety analysis of the propagation of access rights, that is, the determination of whether or not a given subject can ever acquire access to a given object.

References

[1] S. G. Akl, Digital signatures: a tutorial survey, *Computer*, 16 (2) (1983) 15–24.

[2] D. D. Clark and D. R. Wilson, A comparison of commercial and military computer security policies, *IEEE Symposium on Security and Privacy*, 1987, pp. 184–194.

[3] E. Cohen and D. Jefferson, Protection in the Hydra operating system, *5th ACM Symposium on Operating Systems Principles*, 1975, pp. 141–160.

[4] D. W. Davies, Protection, in B. W. Lampson, M. Paul and H. J. Siegert (eds.), *Distributed Systems: An Advanced Course*. Springer-Verlag, Berlin, 1981, pp. 211–245.

[5] J. B. Dennis and F. C. Van Horn, Programming semantics for multiprogrammed computations, *Commun. ACM*, 9 (3) (1966) 143–155.

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

[7] L. Gong, A secure identity-based capability system, *IEEE Symposium on Security and Privacy*, 1989, pp. 56–63.

[8] G. S. Graham and P. J. Denning, Protection: principles and practice, *AFIPS Spring Joint Computer Conference*, 40, 1972, pp. 417–429.

[9] W. Harkness and P. A. Pittelli, Command authorization as a component of information integrity, *Computer Security Foundations Workshop*, 1988, pp. 219–226.

[10] M. H. Harrison, W. L. Ruzzo and J. D. Ullman, Protection in operating systems, *Commun. ACM*, 19 (8) (1976) 461–471.

[11] M. H. Harrison and W. L. Ruzzo, Monotonic protection systems, in R. A. DeMillo, D. P. Dobkin, A. K. Jones and R. J. Lipton (eds.), *Foundations of Secure Computations*, Academic Press, New York, 1978.

[12] B. W. Lampson, Protection, *5th Princeton Symposium on Information Science and Systems*, 1971, pp. 437–443. Reprinted in *ACM Operating Systems Rev.*, 8 (1) (1974) pp. 18–24.

[13] H. M. Levy, *Capability-Based Computer Systems*, Digital Press, Bedford, MA, 1984.

[14] T. A. Linden, Operating system structures to support security and reliable software, *ACM Computing Surveys*, 8 (4) (1976) 409–445.

[15] R. J. Lipton and L. Snyder, A linear time algorithm for deciding subject security, *J. ACM, 24* (3) (1977) 455–464.

[16] A. Lockman and N. Minsky, Unidirectional transport of rights and take-grant control, *IEEE Trans. Software Eng.*, SE-8 (6) (1982) 597–604.

[17] N. Minsky, Synergistic authorization in database systems, *7th International Conference on Very Large Data Bases*, 1981, pp. 543–552.

[18] N. Minsky, Selective and locally controlled transport of privileges, *ACM Trans. Programming Languages and Systems*, 6 (4) (1984) 573–602.

[19] J. D. Moffett and M. S. Sloman, The source of authority for commercial access control, *IEEE Computer*, 21 (2) (1988) 59–69.

[20] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse and H. van Staveren, Amoeba: a distributed operating system for the 1990s, *IEEE Computer*, 23 (5) (1990) 44–53.

[21] J. H. Saltzer and M. D. Schroeder, The protection of information in computer systems, *Proc. IEEE*, 63 (9) (1975) 1278–1308.

[22] R. S. Sandhu, *Design and Analysis of Protection Schemes Based on the Send-Receive Transport Mechanism*, PhD thesis, Rutgers University, 1983.

[23] R. S. Sandhu and M. E. Share, Some owner based schemes with dynamic groups in the schematic protection model, *IEEE Symposium on Security and Privacy*, 1986, pp. 61–70.

[24] R. S. Sandhu, The schematic protection model: its definition and analysis for acyclic attenuating schemes, *J. ACM*, 35 (2) (1988) 404–432.

[25] R. S. Sandhu, Expressive power of the schematic protection model, *Computer Security Foundations Workshop*, 1988, pp. 188–193.

[26] R. S. Sandhu, Transaction control expressions for separation of duties, *4th Aerospace Computer Security Applications Conference*, 1988, pp. 282–286.

[27] R. S. Sandhu, Transformation of access rights, *IEEE Symposium on Security and Privacy*, 1989, pp. 259–268.

[28] R. S. Sandhu, Separation of duties in computerized information systems, in S. Jajodia and C. E. Landwehr (eds.), *Database Security IV: Status and Prospects*, North-Holland, Amsterdam, 1991, pp. 179–189.

[29] A. Silberschatz, J. Peterson and P. Galvin, *Operating System Concepts*. Addison Wesley, Reading, MA, 1991.

[30] Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS), Bentley College, MA, October 1987.

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