A Framework for More Effectively Specifying Secure Policy for Groupware Applications

Rowley, Andrew; Dollimore, Jean

ISSN 1369-1961

Department of
Computer Science

Technical Report No. 743

A Framework for More Effectively Specifying Security Policy for Groupware Applications

Andrew Rowley
Jean Dollimore

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

October 1997

A Framework for More Effectively Specifying Security Policy for Groupware Applications

Technical Report 743

Andrew Rowley
Jean Dollimore
Department of Computer Science, Queen Mary and Westfield College, University of London

October 1997

We believe that many security policies cannot be accurately represented by a conventional access control list (ACL). This applies particularly to policies designed to prevent the malicious or accidental actions of individuals directly and legitimately involved in a group activity. The main problem is that the process of obtaining and changing rights should often be a more dynamic process than is supported by conventional ACLs. We have two techniques for more accurately expressing a policy.

First, the right of an individual to execute some operation will often change according to the state of the task being undertaken. The state of the task is represented by the state of the data objects within the task. In fact the very objects that are being protected. Additionally, our paper also explores another common technique of obtaining rights that is applicable in applications involving groups of participants. We introduce our notion of backing where rights are obtained dynamically through authorisation by one or more other people.

Both of these ideas have been incorporated into an access control framework for groupware applications. We give several examples of policy particularly from the medical field and show how they would be modelled using the framework.

---

1 Introduction

Sensitive information such as medical details need to be held within a computer securely. Access to it needs to be controlled in a way specified by the security policy. Security policy is usually held by a computer system in the form of an access control list (ACL) which contains the identities or roles of those permitted to access and change the information through defined operations. However, we believe that a conventional access control list containing only identities or roles is limited in the policies that it can describe. This could result in the desired policy not being enforced, but instead some compromise. Obviously this is not desirable.

Rights often change as a particular task proceeds. The shared state of the task of course is held within the shared objects of the task and so our access control framework allows this state to be consulted and even updated as part of rights evaluation. This scheme is therefore more flexible when preventing personnel directly involved in a task from corruptly accessing or updating information by allowing an ACL to more precisely express the circumstances under which a person really needs to know or change the information, i.e. that access is in the context of a legitimate activity.

There are examples of policy statements in the proposed BMA guidelines for Security in Clinical Information Systems [AND96] which cannot be expressed conveniently using a conventional ACL. For example it is noted (page 12) that access needs to be granted to roles (e.g. nurses). This is convenient and necessary because the actual identities of individuals are not known when the policy is defined. However it is also stated that extra restrictions need to

be enforced "for example, the group might be any clinical staff on duty in the same ward as the patient". This policy expresses rights in terms of object state, in this case the right to read medical information depends on the state of the duty rota.

We go further than this however. A second example of a policy that a conventional ACL cannot describe is where actions should only be taken or authorised by more than one principal acting together. Often it is the case that certain functions are so sensitive that prior permission has to be obtained from a group, or some proportion of it, before each attempt to invoke the operation. Another medical example highlights this. Often it is the case that multiple sources need to contribute towards a patient's treatment [DRA96] and policy dictates that only the whole group or some proportion of the group can authorise certain actions. Groupware systems aid collaborative working when the participants are remote. If actions, such as prescribing drugs for example, were modelled using a distributed computer system it would be necessary to express the need to obtain backing and enforce it by making participants prove that they have the support of others.

In summary, these examples show that the right to do something may not just depend upon identity but also upon more dynamic properties such as the state of the object being accessed (e.g. the medical record) and other objects (e.g. the duty rota). We refer to this as **state-dependent access control**. Additionally gaining rights can result from negotiation with others in a way that goes beyond simple delegation (as defined by Lampson [LABW92]). We refer to this as gaining **backing**.

There is other work that has attempted to address this problem. The Legion system [LG95] was designed to support distributed collaborative applications and provides a simple approach to security by allowing access to be controlled by methods provided by the application programmer. Simply, the method 'May-I' is automatically called by the access control system before invocations. As a method belonging to the object being protected, it could consult the object's state. However we believe that separating access control from applications is important for easy alteration if policy should change. Incorporating policy into application code therefore is not a good idea.

The authentication service CARDS [HT] provides a more elaborate approach, by allowing a task to be divided into phases. Different sets of rights are assigned to the principals and roles within the task at the different phases. Rights then change automatically as the task progresses through these phases. No consultation of object state is catered for however.

The descriptions in this report are at the level of shared objects used by the application programmer. Some of these objects, such as the ones being protected and others that we introduce, are shared between the various participants in the group activity. We are not directly concerned here about how this sharing is implemented. This paper would apply equally to server based schemes [DWX93] [DABW95] or to a more distributed scheme such as we have described in previous work [RD95]. However we will say that we believe that the decentralised and scaleable advantages of a distributed replicated approach to be our ideal. For this reason we have included an appendix which covers some issues relating solely to such an implementation.

The following diagram clarifies the level at which we are concerned here. We don't make any assumptions about the nature of the secure communication layer other than it must provide authentication and optional secrecy with an appropriate model of trust for groupware applications. It could be secure point-to-point such as the secure sockets layer, or a suitable secure group communication system [RD96a] [RD96b].

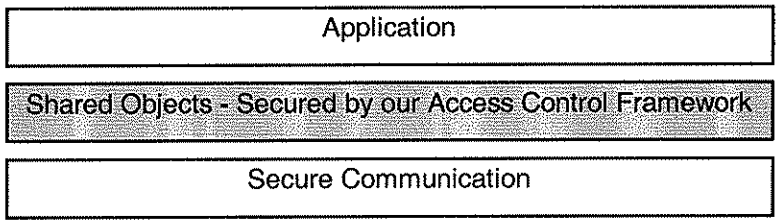

Fig. 1.1 The access control framework is at the level of shared objects, below the application but above some secure communication layer.

The level of the secure shared objects encapsulates the ACLs and of course the shared object data. The application programmer however is only concerned with the objects and associated methods, i.e. security is transparent to the application layer and in the most part to the user as well. The access control checks are applied invisibly in the shared objects layer by a guard which has access to the ACL and intercepts all attempted accesses. The following diagram expands on the central (highlighted) layer of fig. 1.1 to include the major components of the layer.

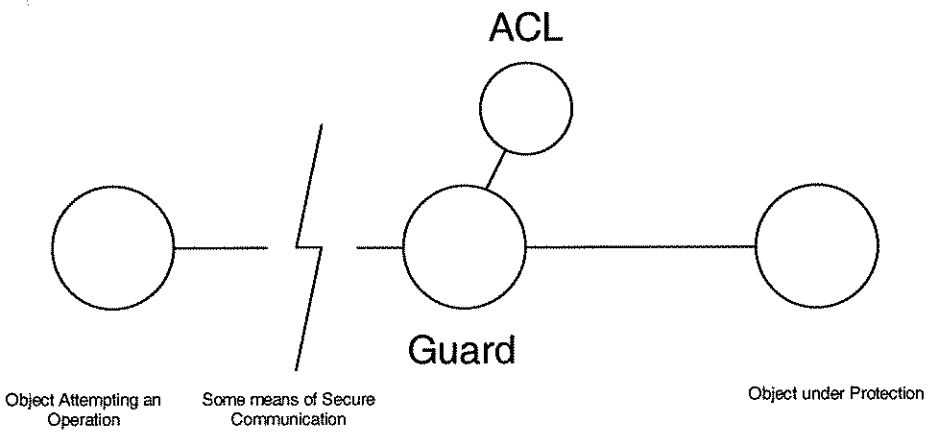

Fig. 1.2 In the Shared Objects Layer the guard has access to the ACL and filters out illegal attempts to access the object that it is protecting.

Our framework also incorporates the notion of roles (explained further in section 2) which is a security concept that the user must be aware of [LABW92]. It is necessary that the user can chose which role is currently being asserted. Hence security is not entirely transparent to the user. Also our notion of backing requires the exchange of security information with the user. Therefore if we do wish security to be transparent to the application then some way of bypassing the application layer is needed in order for the secure object layer to gain information from the user. We facilitate this, as does PerDis [CDKR97], with a security shell.

We envisage the default appearance of the shell as being a simple GUI showing available and current roles, although it is worth stating that its function could be overridden by an application if there is a way of representing protection in a manner more suited to that application. One example is the Mushroom system [TK96] which provides a room metaphor for collaboration. Protection could be modelled in a manner consistent with this metaphor, for example modelling the tightening of the protection of an object in a public room by dragging it into a private room.

Central to our work is our proposed format and semantics for ACLs. The following section looks at a conventional ACL and shows how it might be augmented to facilitate the consultation of data. The work described here adopts a task-oriented framework for structuring access control similar to that of Coulouris and Dollimore [CD94]. We summarise the relevant

issues of the task framework in section 3. Sections 4 and 5 describe in more detail how state-dependent access control and backing might be implemented in a task oriented framework. In section 6 we summarise our conclusions and indicate how we feel future work should proceed. Finally in an appendix we take a brief look at some issues that relate solely to implementing this framework in a system where the shared data is replicated.

2 Augmenting Access Control Lists for the Expression of State-Dependent Policy

Security policy is most often expressed using an Access Control List. A conventional ACL is a list of permitted principals associated with an operation. In an object-oriented setting the operations are methods of objects. The list can only be changed through the intervention of a permitted authority and so generally rights remain static.

Lampson [LABW92] includes the notion of roles. Roles are generic categories such as Physician, Manager and Lecturer for example. Naming a role in an ACL rather than a specific principal simplifies the process of writing an ACL, but it does mean that a method of granting the right for a principal to take on a role must be implemented together with a means of checking these rights. This would generally be achieved, as it is in the TAOS operating system [WABL94] for example, through some authority signing certificates stating a principal's right to act in some role. These certificates have a certain life-time during which they can be used which makes rights difficult to revoke. Therefore merely having roles mentioned on an ACL is still relatively inflexible as it doesn't allow for rights to evolve without intervention as a task progresses. So although roles remain very important in the framework, we believe that additions need to be made to the structure of ACLs in order to allow the consultation of some state that forms part of the task.

It helps our explanation to regard an ACL as a boolean expression. Take the following example of a list expressing the principals entitled to perform an operation *Op1*:

Op1: "Dr X", "Dr Y", "Dr Z"

Which expresses the policy:

"Only Dr X, Dr Y and Dr Z can invoke Op1"

If the identifier *principal* represents the identity of the individual attempting to perform the operation, then the ACL could be expressed in a form that is similar to a boolean expression such as those forming part of a conditional statement in a programming language. If we use the symbol "==" to represent an equality operator, then *principal* has the right to perform *Op1* if:

(principal == Prin1) or (principal == Prin2) or (principal == Prin3)

Which of course expresses the same policy.

When *principal* is instantiated then the expression will evaluate to true (access permitted) or false (access denied). This is also so for an ACL that contains roles. For example:

Op2: Physician, Manager

Policy: "Only Physicians and Managers can invoke Op2"

A programmer writing a boolean expression could be able to include in the expression calls to functions in his program if necessary, so the evaluation of the expression could take into account the state of objects within the program. Allowing an ACL to contain object method calls in a similar way would enable rights to depend on object state. So for example, if there

was a function that returned true if the patient was under eighteen years old, then the right to perform an operation could depend upon this:

Op3: Physician and MedicalRecord.patientOverEighteen( )

Policy: "Physicians can invoke Op3 if the patient is over eighteen"

Where *MedicalRecord* names some object which is the target of the *patientOverEighteen()* method call. This example also introduces the use of the boolean operator *'and'* -which is used in the context of access control by Lampson [LABW92].

In section 5 we expand on this idea to enable the need to obtain backing to be expressed, but first it is necessary to more clearly explain rights within a task-oriented framework.

3 Tasks for Structuring Access Control

The work described in this report is fundamentally based upon the task-oriented framework of Coulouris and Dollimore [CD94] which is subsequently being worked into the PerDis system [CDKR97]. This section explains the elements of this task framework that we utilise.

The previous section explained Lampson's [LABW92] ideas for allowing roles to be entered in an ACL as well as specific principals as a means of simplifying the process of specifying rights. In any system that requires access control there are by its nature a number of principals that are potentially involved. This number could be large, hence making generalisations about the principals' roles is an obvious way of simplifying the construction of ACLs.

In an object-oriented groupware system there are not only potentially large numbers of principals but also large numbers of objects that need ACLs. Generalising about the objects too can ease the specification of rights.

The task-oriented framework that we use combines the two notions of generalisation. A generalised task will have a set of objects and a set of roles that are common to all specific instances of the task. Take for example a GP's task of administering to a patient. In general each patient will have a set of objects comprising his or her medical record (such as notes, prescription and referral details etc.). Every patient has these objects and the rights to access them (when expressed using generic roles) are the same for each instance of a task. Expressing them once in a generic **Security Template** is more practical than having to do it for every patient. An extremely simple (and far from ideal -as we will go on to explain) example might be as follows:

Task: Administering to Patient

Roles \ Objects	Notes	Referrals	Prescriptions
Patient's GP	read/write	read/write	read/write
Nurse	read	read	read
Receptionist	-	read	read

Fig. 3.1 The table shows the rights of roles within the task "Administering to Patients" to access three types of objects used by all instances of the task.

Some tasks might have many instances of a type of object, such as the patient notes objects in fig 3.1. Here again it is possible to take further advantage of generalisation and specify rights to access categories of objects, i.e. all objects in the same category share an ACL. Again this is the approach taken by the designers of the PerDis system.

When a task comes into existence, the details that need to be available to the participants such as the ACLs and the roles which take the form of signed certificates are copied from the security template into a shared object called the **Task Object**. This is a shared object just like those that form part of the application. Changes to the ACLs and the actual principals entitled to take on the roles will be available to all. The task object therefore can then satisfy queries regarding the current membership of a role. This is import for our notion of backing as will become clear.

In our expansion of the framework it is necessary for ACLs to be able to refer to specific objects, roles and principals within a task. Therefore objects and roles are referred to through their tasks, such as the patient notes in the specific task instance of administering to a patient X. Tasks themselves could be named by universal resource locators (URLs) as are other Internet abstractions, although that is beyond the scope of this paper.

In summary then, protection is specified in a security template. This contains details of the shared object categories that form part of the task, ACLs for these objects and the roles that will participate in the task -but generally not actual principals. When an actual instance of the task is created, actual objects are created too and references to them are placed in a shared task object. The task object also holds the ACLs that are created according to the security template and the identities of the actual principals that are assigned to the roles in the task.

4 State-Dependent Access Control

We start our look at state-dependent access control by examining some security policies.

4.1 State-Dependent Security Policies

The simplest kind of state-dependent policy is one where a right depends upon the state of the same object that is being protected, i.e. the ACL on one of the object's methods only contains calls to other functions of the same object. For example, another policy taken from Ross Anderson's report [AND96] suggests that the right to delete medical records might only occur after a specific period after the death of the patient. Hence the guard must consult the date-of-death state.

Another policy might state that a General Practitioner's receptionist can only read records of patients that have an appointment to see a doctor. This policy is an example where rights depend on some object other than the one being protected.

Both these policies are examples of attempts to constrain the circumstances in which a principal can access an object to reduce the damage that a corrupt individual can do. They are aimed at ensuring that the access is in the context of a legitimate activity. These types of policy are very common outside computer systems, however inside are often left unenforced because of a lack of mechanism.

Both these examples also depend upon some environmental state, in this case time-related. We have only encountered policies that refer the current time and date.

In the introduction we gave another example of a security policy in which rights depended upon the state of the duty rota. The policy stated that the right to perform some operations, such as prescribing drugs for example, may require that the doctor is not only on duty at that time, but also on duty in the same ward as the patient. This policy for protecting the medical record relies upon the state of a duty rota object and of the current time.

The final kind of policy that we have encountered isn't dependent upon the state of any objects, but is dependent upon the state of the parameters of the attempted operation. Such a policy might specify for example that one of a team of architects working on a shared plan for a building might be able to move a door, with a proviso that it not be moved more than 5

centimetres. Moving it a larger distance might require the authority of another role such as the manager for example because this may conflict with other participants' activities.

As a second example of a parameter-dependent policy taken from the financial field: bank security procedure as detailed by Kusner and Anterpol [KA81] states that bank tellers should not be able to process transactions that involve themselves. This policy relies upon the parameters to the operation and on the identity of the teller.

A medical example might be that certain medical staff could prescribe only certain drugs, or certain maximum quantities of drugs.

Summarising these types of state-dependent policy then we have:

1. Policy dependent upon the state of the object being protected by the ACL.
2. Policy dependent upon other protected objects.
3. Policy dependent upon environmental state such as time.
4. Policy dependent upon the parameters of the operation being attempted.

Allowing ACLs to contain method calls, references to parameters and environmental values, combined with boolean operators in a manner similar to a boolean statement in a programming language will cater for all the policies that we have encountered.

Now we go on to show how such policies could be expressed in ACLs in our framework.

4.2 Expressing State-Dependence in an Access Control List

When method calls are included in an ACL, it is necessary of course to specify which object the methods are to be called on. When the target of the call is the same object as the one being protected by the ACL, then in our syntax for ACLs we prefix the call with 'this' -a keyword with similar meaning to 'this' in Java or 'self' in Smalltalk. If the object is different from the one being protected by the ACL then it must be named.

When objects are created within a task, objects are assigned to categories. This is how they obtain their initial versions of their ACL. It is a copy of the template ACL for that category. As explained in section 3, the category is a means of generalising about objects -so that an ACL only has to be supplied for the category and not every individual instance. Medical Records for example could be a category, because there are many of them in the task of running a practice. Each record would have the same ACL (initially at least) when rights are expressed for roles.

This means that objects from categories with more than one instance could not be named in an ACL entry when the ACLs are compiled, because they are not known to exist before the task is instantiated. The ACLs of course exist in template form before any instances of the tasks. This may at first seem restricting, but it is our view that no security policy would depend on these objects -since the policy too of course exists in advance of any individual instances of tasks or objects. It seems unlikely that a security policy for running a practice, for example, would rely upon the state of a specific patient's medical record.

In order that the integrity of an ACL can be verified immediately that it has been written, it is necessary that categories with only one object are marked as such in the security template.

The temporal information needed in the policy are expressed in ACLs using the following identifiers. Other policies that we have not encountered could conceivably need others.

today.year	The current year.
today.date	The date in some format.

Others might include: today.time, today.month, today.hour, today.minute, etc. All of which could be obtained from the operating system by the guard.

Additionally the examples showed that certain information has to be available about the identity of the principal. The extent of information that could be referred to in a security policy is wide ranging. For example policies could potentially refer to phone numbers, postal addresses For this reason we do not propose a scheme along the same lines as for the temporal information. We assume that there must be application level objects that will return the requested information given the identity of the principal attempting access. This identity is referred to in the ACL as *principal* and contains a string uniquely representing the principal.

There would be no further problems if all applications were written in the same object-oriented programming language. Ideally the ACL will be independent of the language that the objects are written in -thus allowing the objects to be used by a variety of applications written in different languages. However this causes a potential problem with the types that we can compare in the ACLs since not all language support the same types. Hence if many languages are to be catered for then it will be necessary to restrict the types allowed to be included in an ACL to some basic types found in most languages, such as integers, characters, strings, etc. and then force all languages to supply methods for converting these into their own representation. These methods would be called by the guard before any methods contained within the ACL.

Below are example policies representing the four kinds previously identified. In the absence of actual applications we have proposed some likely methods and their parameters which are probably over-simplified, but demonstrate the possibilities.

Policy:	Physicians can only delete a record entry of patients deceased for 10 years.
Source:	[AND96]
Type:	1 and 3
Roles involved:	Physician
Object Type:	Medical Record
Method:	Any delete method
ACL Entry:	Physician and (time.year - this.yearOfDeath( )) > 10

Policy:	Physicians can only prescribe to patients on wards where (s)he is currently on duty.
Source:	[AND96]
Type:	1, 2 and 3
Roles involved:	Physician
Object:	Medical Record
Method:	Prescribe(drug)
ACL Entry:	Physician and rota.onDuty(principal.surname, this.ward( ), today.date, today.time)

Policy:	Bank tellers can't make transactions that involve themselves.
Source:	[KA81]
Type:	2 and 4
Roles involved:	Teller
Object:	Transaction Record
Method:	addTransaction(sourceAccNum, destAccNum, amount)
ACL Entry:	Teller and Accounts.getIdentity(sourceAccNum) != principal

4.1 A table showing example state-dependent policies and how they would be expressed in ACLs in our framework.

Many applications do not enforce adequate access control because the mechanisms simply are not there to implement them. However this does not mean that the applications are bad. It

is our intention that our framework could be integrated into existing applications with a minimum of disruption. We believe that many applications will already have the necessary methods that are needed for inclusion in the ACLs. Take the second example policy given in fig. 4.1 above. This suggested that the right to prescribing drugs for may require that the doctor is on duty at that time and also on duty in the same ward as the patient. Even before any mechanisms existed for the enforcement of this policy -it must have been possible for the doctor to look up his duty schedule. Hence there is likely to be some method for this.

However, if there isn't an appropriate method then the policy cannot be enforced accurately unless a method is added. Modifying application code once to provide methods for accessing state on which policy depends is far less severe than forcing application code to be modified every time there is a policy change. Our scheme offers a significant advantage over systems such as Legion [WWK95] because policy is represented entirely in ACLs.

We have only considered policies that need to inquire the state of objects. We have not considered the inclusion in ACLs of methods that update the state of objects. None of the policies that we have encountered require that state be updated. Generally policies are expressed in terms of what can and cannot be done and do not mention actions that are to be taken as a result of an attempted access to an object. Although security auditing is a possible exception.

5 Backing

Sometimes it is the case that operations are too security sensitive to be performed by one person alone. Splitting responsibility between two or more principals, so that the consent of all or some proportion of the group is required in order that the operation be performed is a common way of ensuring that a single corrupt principal cannot alone do damage.

Policies that we have seen which are stated in terms of a need to obtain backing are for one-off operations, i.e. others give backing for you to do something once only. In this way backing differs from delegation which is for a period of time. Also it differs in that backing potentially comes form many other principals.

As in the previous section, we start by a look at some example security policies.

5.1 Security Policies that Specify a Need to Obtain Backing

There are many examples of this type of security policy outside the computing environment, particularly in the financial world where such policies are used to protect institutions from embezzlement by corrupt employees.

Splitting responsibility between more than one person or 'segregation of duty' as it is referred to in financial literature is one such example where backing can be used. The overriding concept is that no one person can see through a financial transaction from start to finish without the involvement of some other person, who at least must check the work and possibly apply a signature (hand-written) before the person can continue. Hence it would take the corruption and collusion of more that one employee to cause an irregularity. The principal of segregation of duty is described by Gray and Manson [GM89] as it is used in financial auditing and in relation to information systems by Koning [KON].

Kusner's [KA81] lengthy descriptions of banking procedures naturally cover security extensively and give many examples of policy that requires those involved to obtain backing. One policy example states that a teller must seek permission to adjust a special-purpose bank account for reconciling differences between the actual amount of money taken and the recorded amount (should such a discrepancy occur).

There are examples where the computerisation of healthcare tasks would require backing if existing safeguards are to be continued inside a computer system. Draper [DRA96] discusses

the treatment of mental health patients as being a collaborative process, i.e. the backing of some proportion of a group is required to support decisions. Decisions are taken collectively as a safeguard in cases where the patient's consent is not needed. If the participating physicians were remote and communicating electronically then our idea of backing could ensure that appropriate levels of agreement were achieved before actions where taken. Draper stresses the importance of having a "multidisciplinary team being available to readily access, communicate and share patient information." This seems to advocate a groupware system to alleviate the need for the team to be co-located, which would require a medical record accessible to all participants, but protected appropriately. Such a system would be particularly useful considering Draper's comment that such healthcare professionals have to be available around the clock.

Ting [TIN90] also considers the electronic implementation of patient records and gives some example security polices for protection of the information. One example combines both a requirement for backing and state-dependent access control. The policy states that a parent's permission must be given for certain accesses to a patient's record if the patient is under eighteen years old. Although the paper doesn't give any idea how the parent's permission would be represented and be provable in the system[1]. If a patient is older then the patient's permission must be given, i.e. backing must be obtained.

As a final example, Greif and Sarin [GS86] consider a real-time computer conferencing system. They do not mention security directly, but suggest voting as a means of making decisions. However, if the conference were set up for the discussion of security sensitive issues then this voting system would need to be secured. The operations associated with these decisions could be protected with an ACL that required a person to obtain the backing of a majority of the group.

The examples above can be separated into two types. We call these absolute quantity and proportional backing:

1. Absolute Quantity: Where the backing of a specific number of role members is required, e.g. 2 Physicians.

2. Proportional: Where the backing of greater than some proportion of a group of role members is required, e.g. a majority of the board of directors.

5.2 Expressing Backing in an Access Control List

Our scheme for enabling an ACL to express the need to obtain backing is an extension of the method used in the previous section. We simply provide specialised backing methods to be called from within the ACL. However the methods are obviously not implemented by any of the programming level objects that comprise the task, but instead are implemented by the guard itself.

To avoid having to add any additional confusing syntax into our ACLs the backing methods do not actually specify the target of their call, i.e. the default target is the guard.

We introduce two specialised backing methods. One for absolute quantity backing and one for when the proportion of the group is required. We call these *atLeast* and *proportionally*. Each take two arguments, the quantity of backing required (either an exact quantity or a proportion) and a role from whose members the backing must come. This is all we need in order to express all of the policies introduced above. The following table demonstrates their use:

Policy:	Tellers must obtain a manager's permission to balance the accounts.
Source:	[KA81]
Type:	Absolute Quantity

[1] Montgomery [MON97] describes the use of smartcards for involving patients electronically.

Roles involved:	Trainee, Manager
Object:	Account
Method :	finalise ( )
ACL Entry	Trainee and atLeast(1, Manager)

Policy:	A majority of a Patient's carers must agree on new treatment.
Source:	[DRA96]
Type:	Proportional
Roles involved:	Carers
Object:	Treatment Record
Method :	updateTreatment(details)
ACL Entry:	Carer and proportionally(1/2, Carers)

Fig. 5.2.1 A table showing example backing policies and how they would be expressed in ACLs in our framework

Of course there is no reason why the two forms of backing shouldn't be combined by the boolean operators for particularly complex policies. A possible (but fictional) policy for protecting some method might require the backing of:

"two doctors and majority of hospital managers."

Expressed as:

atLeast(2, Doctor) and proportionally(1/2, Manager)

Going even further, they could be combined with state-dependent policies:

(Physician and patientAge( ) >= 18), (atLeast(2, Physician) and (patientAge( ) < 18))

Note that in proportional backing, if the requester happens to be a member of the group from which backing is being sought then it is implicit that the requester backs themself. This isn't so for Absolute Quantity. For example:

Physician and proportionally(1/2, Physician)

expresses a policy requiring the requester to gain a majority of a group which includes himself. However:

Physician and atLeast(2, Physician)

requires a Physician to procure the backing of two other Physicians meaning that three Physicians in total agree.

5.3 The Proof of Backing

In this section we will explain that the backing is accumulated in the task object (introduced in section 3), and what is actually collected when a request is sent out are digitally signed certificates of the signers' consent to an individual to perform some operation. Here we explain exactly what these certificates contain. Section 5.4 will suggest how they might be collected.

The example policies show that what a backer actually consents to is:

- To execute some specific method;
- Upon some specific named object;

- To execute it once;
- Within some period of time.

The contents of the certificate must of course reflect this meaning.

The period of time for which backing will be valid is supplied with requests for backing. This must be displayed to the potential backer alongside the request. Circumstances will often change that could result in the backer changing their opinion. Hence backing is never given indefinitely, merely just for a specified period after the initial request. This period is part of the security policy and hence is specified along with the ACL (although we do not show it here). Hence the recipient of the backing must use the signed statements to perform the operation within this duration.

Additionally, there must be information within the backing statement to enable the guard to ensure that the statements are not used more than once. This is perhaps more of a problem than it first appears. Take as an example a policy that requires that two members of a particular role are required to back a principal to perform some method. Suppose there are ten possible backers and all give consent for the method to be performed. In order to ensure that the method is only performed once it is not sufficient just to recognise that a backing statement has been used before. If this was the case, the principal could make five repeated attempts to perform the operation with different pairs of certificates (each from different backers). We must prevent this, i.e. once the operation has been performed, then all the backing statements must be unusable.

Our solution involves the task object holding information about outstanding backing requests. The principal requiring backing creates an **Outstanding Backing Object** which is another shared object referred to by the task object. This is done by invoking a method on the shared task object. The new outstanding backing object has a unique identifier. Potential backers then pick up the details of the request from the task object and if they decide to grant backing they include the request identifier in the signed body of the backing statement. Backing is then granted by the backer creating the signed backing certificate and installing it in the outstanding backing object from where the original requester and everyone else can retrieve it.

When sufficient backing is collected the operation can be attempted. The necessary backing statements can be accessed by the guard along with other necessary certificates. The guard maintains a list of the unique request identifiers that it has received and as a consequence it can ignore any attempts to reuse statements of backing that originated from the same request.

This mechanism also enables backing requests to be outstanding for a large period of time as would be the case for less synchronous applications. Participants starting new sessions pick up the details of all currently outstanding requests from the task object.

Ideally when sufficient backing is collected the Outstanding Backing Objects in the task server should be cancelled by the initiator. In a secure system any apportion of responsibility should be done with care, however it is not essential to the secure running of the system that the outstanding request is cancelled since it will naturally time-out anyway, as will the guards cache of already used backing request identifiers.

There is a problem however which stems from the gap in the two levels of abstraction. Security policies exist outside the computer system at a level which is conceptually above the application. However the policy must be represented at the level of rights to execute methods upon objects. It might be reasonable to expect the person responsible for translating the policy into access control lists to be sure about what methods actually do as the methods of the protected objects in a well designed application would correspond to real-life operations. However it is not so reasonable to expect the participants in the collaborative task to make a decision about whether to grant backing in response to a request which looks like a method call, i.e. "can prin1 perform updateRecord("Dose", 5)?". This might not mean much to the person. It is necessary to bridge the gap between the shared object level of abstraction and the users' perception.

In order to make more obvious exactly what the sought consent is for, we require that a natural language statement be included in an ACL which is distributed along with the request. This may need to have place markers for any parameters to be filled in. Take the following example of an operation whose ACL contains a statement of rights which depend upon backing being sought:

finalise ( )

This could be displayed on a recipients security shell as:

X requests your backing to 'finalise the exam paper'

Where X is filled in with the principal's identity and 'finalise the exam paper' is the natural language statement of the method's function. The request would appear along with the name of the object that the method will be invoked upon and the task in which it forms part. Both of which would be given a similarly meaningful names.

Finally we can now see that the actual contents of the certificate reflects the semantics of backing outlined at the beginning of this section. Each certificate contains:

- The natural language statement of what the backing is for.
- The method name and value of the arguments, the task reference and object reference.
- The unique backing request identifier.
- The expiry date/time after which the backing statement can no longer be used.
- A signature signed with the private key of the backer.

And as a consequence the backer has given consent to perform the specific operation upon a specific method before a certain time and once only.

5.4 Adapting the Security Shell to Implement the Collection of Backing

The previous section made clear that what the principal who collects backing is actually collecting is signed certificates. However we didn't explain how participants are prompted to given backing. Just as was the case for delegation and role taking, we believe that the application should be able to implement the collection of backing. However if it is not desirable to alter the application then the security shell can be adapted to request the consent of the user as described here.

The security shell bridges the gap between the user and the secure shared objects layer by supplying the layer with information that can only come from the user such as changes to ACLs, delegations to others and the roles currently being asserted. We extend this to include the capability to request and grant backing.

When a principal attempts to invoke a method that requires backing, communication with the guard containing an attempt to perform the operation is not attempted straight away. First the user must be asked if an attempt to go ahead and gather consent should be instigated. This request can be fulfilled by the security shell. The shell will communicate this request to the user and if (s)he decides to go ahead and make the request, then the outstanding backing object is created. Other security shells, which have access to the task object, can now see this request and if it is one to which the user can respond, it will display the natural language request to the user. Users may or may not consent. If they do the certificate is created and installed by them in the outstanding backing object.

Once sufficient backing is collected the security shell of the originator would highlight the operation as an indication that it can now be performed. When the user gives the go-ahead, the method is invoked and ideally the outstanding backing object is cancelled.

6 Conclusions and Future Work

This work has, we believe achieved what it set out to do, which was to allow a principal's rights in a collaborative computerised activity to accurately reflect the kinds of security policy that exist. We can't say that the work is complete because it is not possible to know what every possible security policy could be. However, we believe that allowing a principal's rights to depend upon the state of the data being protected and hence the state of the task being undertaken, together with the notion of dynamically obtaining rights through others has enabled a huge and complex array of rights to be specified. At least, we have been able to express the policies that we have encountered.

State-dependent access control however is limited by the actual access to state that is permitted by the objects, i.e. by what methods are available. If there are no suitable methods then the policy cannot be represented accurately. However we believe that there is no prohibitive reason why objects cannot be extended to include suitable methods. Doing this once so that policy can depend upon a certain aspect of the state is not the same as having security policy actually expressed in application code. Future changes in policy would not necessarily result in further application code modifications, rather modifications are solely confined to the ACL.

Performance is an import issue in interactive groupware. Our ideal implementation of such a framework would be built upon a replicated architecture, with data therefore being local. Hence, the evaluation of rights that depend upon the data would not we believe cause any substantial overheads -although we have no actual implementation to time as yet.

Our framework allows complex policies to expressed and enforced. There is a chance that covert channels may open up if ACLs are written without thought. The fact that whether an attempt to perform an operation depends upon the state of the object, may enable a principal to glean information about its state -even if the attempt to invoke a method fails. This may at first seem worrying.

However security policy initially exists outside the system that will enforce it and that any sensible policy carefully transposed into an ACL will not contain loop-holes. In other words if there are covert channels then it is not our framework that is at fault, but whoever converted that policy into an ACL. The process of transforming high-level policy into rights to invoke methods on objects could be automated which might allow for the checking of covert channels.

Any problems that do arise in the transposing of the policy could be prevented if there was a more efficient way of bridging the gap between the user's perspective of policies and the application interface and our shared-object level of compiling ACLs. We mentioned this problem first in the introduction and again during section 5 in the discussion of making backing requests and the requirement to supply an natural language description of the backing being sought. The security shell is a basic attempt to bridge the shared object to user-level gap, that exists when specifying security, however it is not ideal. It still requires that policy be expressed in terms of rights to perform methods on objects.

More ideal would be a method of specifying policy at a higher level. One possible solution would be the use of policy specifiers. General purpose descriptions of rights that could then be automatically translated into rights to perform methods upon objects. This is not something that we have as yet attempted any work on, but is the next stage.

Appendix: Replicated Implementation Details

In the first section we explained how our access control framework could be applied whatever the underlying architecture. However we believe that for groupware, a replicated architecture is advantageous. For performance reasons replicating data locally is desirable, even at the expense incurred of having to distribute all updates in a secure fashion to all the replicas when ever a change is made.

The secure shared object abstraction can be built on top of a secure group communication system [RD95]. The secure group communication system must work in a suitable model of trust. Groupware applications in which the participants are given different rights, such as ones which may utilise the framework explained in this report, must, in order to be as secure as possible, be built on a group communication system that doesn't rely on the uncorrupted functioning of the software running at any of the participant's machines. Put bluntly, if the participants have to have their access controlled, then they are not fully trusted and so any communications coming from there machines, cannot be fully trusted. Access control is only applied if there is a risk that someone might attempt to do something illegal after all. A suitable secure group communication system is described in our paper [RD96a] and in the more detailed report [RD96b].

The shared objects, built on top of such a secure group communication system could then have access control applied to them using the framework described here. However there are some issues that need further explanation. Complications that arise due to the distributed nature of the architecture.

The guard, when evaluating a principal's right to access the object that it is protecting may, under our state-dependent scheme, be required to consult the state of another object. If the state of this object is not already local, then it must be fetched. This would be achieved as part of a normal group join operation of the group communication system. Where this state comes from is an important security issue. It is actually a problem that lies in the group communication layer. Our solution is to have one trusted member of the group which is responsible for sending state to new members. This member may be located on a more physically secure machine and not play any interactive part in an application, i.e. it is a server, but is a normal member of groups in every other way. Again this issue is dealt with more thoroughly in our report [RD96b]. It is also the case that in order to securely ensure that every recipient receives the same message that all multicasts are directed through this machine.

Related to this there is a performance issue. If the state is already local, then it should not be necessary for the state to be fetched again. Therefore we suggest that a register of all the shared objects that are currently located on a machine be maintained and updated as necessary.

There is a problem concerning the distributed evaluation of certificates, such as role membership certificates or signed statements of backing. In a distributed replicated architecture, certificates are duplicated and sent out to all the parties that need to evaluate rights. This happens on every invocation of a method that updates the state of a shared object. It is important for the consistency of the replicas that all the members of the group either perform a particular update method or reject it. In the most part the group communication system will ensure that this happens by totally ordering the multicasts containing the updates and ensuring that any lost messages are resent and arrive eventually.

However, certificates generally have a time-out. This is definitely the case for backing certificates since principals do not generally grant backing for an indefinite period. Certificates, in a system where no participant cheats would be refreshed long before their expiration time. However in our system of differently trusted principals, we should not rely on this. This leads to the possibility of a certificate failing the expiration test. However, if all the clocks on the machines are not exactly synchronised, then the test might fail at some sites but not others, leading to an update being applied inconsistently. This applies equally to the environmentally dependent rights that depend upon the time of day or date.

The trusted server can solve this problem if we ensure that it stamps all the update messages with the time of day and date. It should be this date that is used to check certificate time-outs and this date that is used for the evaluation of time-dependent rights.

Note that this doesn't work if the shared object is server based however. We can't trust the source of the message to apply their own time stamp because they could cheat. In this case

the guard must use its own idea of what the time is, taken presumably from the its system clock.

8 References

[AND96] Anderson R "Security in Clinical Information Systems" Computer Laboratory, University of Cambridge, January 1996.

[CD94] Coulouris G and Dollimore J "Protection of Shared Objects for Cooperative Work" TR 703, Department of Computer Science, Queen Mary and Westfield College, August 1994.

[CDKR97] Coulouris G, Dollimore J, Kindberg T and Roberts M "A security architecture for PerDis in Java" Position paper for the Workshop on Persistence and Distribution, Portugal, October 1997.

[DWX93] Dollimore J and Wang Xu "The Private Access Channel: A Security Mechanism for Shared Distributed Objects" TOOLS Europe 93, pp. 211-222, March 1993.

[DABW95] van Doorn L, Abadi M, Burrows M and Wobber E "Secure Network Objects" TR 385, Digital Systems Research Center, Palo Alto, California, USA, 1995.

[DRA96] Draper R "Electronic Patient Records: Usability vs Security, with Special Reference to Mental Health Records" Proceedings of Personal Information Security, Engineering and Ethics, Cambridge, June 1996.

[GM89] Gray, I and Manson, S "The Audit Process" ISBN 0-278-00044-4, Van Nostrand Reinhold (International), London, 1989.

[GS86] Greif, I, Sarin, S, "Data Sharing in Groupwork", Proceedings of the first Conference on Computer Supported Cooperative Work (Austin, Texas), ACM New York, pp. 175 - 183, December 1986.

[HT] Holbein R and Teufel S "A Context Authentication Service for Role Based Access Control in Distributed Systems - CARDS" Department of Computer Science, University of Zurich, Winterhurerstasse 190, CH-8057, Zurich, Switzerland.

[TK96] Kindberg T "Sharing Objects over the Internet: the Mushroom Approach" IEEE Global Internet 96, London, November 1996.

[KON] de Koning, WF "Security within Financial Information Systems (as seen from an auditor's point of view)" Paardekooper & Hoffman, Calandstraat 41, PO Box 23123, 3001 KC Rotterdam.

[KA81] Kusner K and Anterpol J "Modern banking checklists : with commentary" 3rd ed, Warren, Gorham & Lamont, Boston, 1981.

[LABW92] Lampson BW, Abadi M, Burrows M and Wobber E "Authentication in Distributed Systems: Theory and Practice" ACM Transactions on Computer Systems, vol. 10, pp. 265-310, November 1992.

[LG95] Lewis M and Grimshaw A "The Core Legion Model", Department of Computer Science, University of Virginia, August 1995.

[MON97] Montgomery, J "Health Care Law" Oxford university Press, 1997.

[RD95] Rowley, A, Dollimore, J "Replicated Secure Shared Objects for Groupware Applications", TR 716, Department of Computer Science, Queen Mary & Westfield College, University of London, March 1995.

[RD96a] Rowley AJ and Dollimore, J "Secure Group Communication for Groupware Applications" Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems, Zinal, Switzerland, March 1997.

[RD96b] Rowley AJ and Dollimore, J "An Implementation of a Secure Group Communication for Groupware Applications" TR 732, Department of Computer Science, Queen Mary & Westfield College, University of London, January 1997.

[TIN90] Ting, TC, "Hospital Records Security", Database Security: Status and Prospects, vol. 3, North Holland, 1990.

[WABL94] Wobber, E, Abadi, M, Burrows, M, and Lampson, B, "Authentication in the TAOS Operating System," ACM Transactions on Computer Systems, vol. 12, no. 1 February 1994.

[WWK95] Wulf W, Wang C and Kienzle D "A New Model of Security for Distributed Systems" Computer Science Technical Report CS-95-34, University of Virginia, August 1995.