Towards a Configurable Security Architecture

Martin S Olivier

Department of Computer Science, Rand Afrikaans University, PO Box 524, Auckland Park, Johannesburg, South Africa, e-mail:molivier@rkw.rau.ac.za

Abstract

Current security systems are either an integral part of the system they are protecting or a separate module external to the system. This paper describes an architecture for a security system that allows security components to be developed separately from the system they have to protect and then integrated into the system at a later stage. This means an organisation may select the required components based on its needs and 'plug' these components into their systems.

The paper describes the proposed architecture and demonstrates how it may be used to support various forms of discretionary and mandatory authorisation.

Key words: Security, access control, configurable access control, component-based security

1 Introduction

A number of models exist to support security in computerised systems. They include the basic mandatory, discretionary, role-based and other systems, as well as a number of more specialised and esoteric models. The intention of this paper is to describe a model that supports customisation. Stated differently, the model in this paper may be adapted to suit an organisation's needs by simply modifying a number of simple components. If implemented in practice, it is envisaged that vendors will supply various modules or components that may be 'plugged into' the system to be protected. It may therefore become possible to buy an operating system or database from one vendor and buy an appropriate security module from another vendor.

The potential benefits of such an approach are enormous. In the first place it means that a user (or organisation) is not forced to use the security system that is supplied as standard with a product. It also means that vendors may construct a single version of a product with different security modules for 'normal' users and for those requiring highly trusted systems. In fact, it may allow database vendors to focus on data management issues and security vendors to focus on security issues. Finally, an organisation that wants to employ uniform security features on a range of products from various vendors, may do so by installing the same security module on all the products that they use. Many of the benefits cited above will obviously only become viable once the notion has been proven, accepted and standardised.

This paper proposes a possible architecture for such security components and an infrastructure with which such security components may be used. An object-oriented environment is assumed (and is essential to the operation of the described components). It is demonstrated how multilevel security, discretionary security and role-based components may be used with the components. It will also be clear that it is possible to construct components for other security models.

For ease of reference the architecture described in this paper will be referred to as ConSA (*Configurable Security Architecture*).

The remainder of this paper is structured as follows: Section 2 contains background information. Section 3 describes the proposed architecture. Section 4 illustrates the application of the architecture with a role-based example. Section 5 contains a number of remarks about implementation and related issues. Section 6 concludes the paper.

2 Background

Computer security has to ensure the confidentiality, integrity and availability of computing resources—especially of information. The resources to be protected are usually referred to as *objects*; to avoid ambiguity in the object-oriented context, we will refer to the resources that may be protected as *entities*. Users (and other originators of requests in the system) are referred to as *subjects*. The manner in which a subject accesses an entity is referred to as the *access mode*. Examples of access modes include reading and writing. Since objects in an object-oriented system can only be used by sending messages to the objects, the methods supported by an object form the access modes for the object.

Access control is one aspect of security. An access control policy specifies which subjects are allowed to access which entities using which access modes. The access control mechanism has to enforce the policy in a given system. Many access control models have been proposed. Most have been categorised as either discretionary access control models or mandatory access control models (also known as multilevel security models); some fit into both of these categories.

A *discretionary access control* (DAC) model is an access control model where an entity is owned by a subject; in most cases an entity is owned by the subject that created it. The owner of an entity has the authority to grant access to the entity to other subjects, and to revoke it from them. Access restrictions are viewed conceptually as an *access matrix* where the rows represent the subjects, the columns the entities and the entries in the matrix the allowed access modes for the particular subject and entity. Access matrices are usually implemented with access control lists or with capabilities. An *access control list* (ACL) is a list of user identifiers associated with a protected entity. The list contains the identifiers of subjects allowed to access the entity, as well as the allowed access mode. A *capability* is an unforgeable identifier that a subject may present to gain access to an entity. The allowed access modes are usually encoded in the capability.

In a *mandatory access control* (MAC) model each subject is assigned a clearance level, while each entity is assigned a classification. A subject is authorised to access an entity if its clearance dominates the classification of the entity. Since clearance and classification levels are used in such models they are also often referred to as *multilevel security* (MLS) models. Levels may be fully ordered, such as the simple military model where

$$\textit{Unclassified} < \textit{Confidential} < \textit{Secret} < \textit{Top secret}$$

Categories may also be introduced, where the clearance level of a subject has to dominate the classification of the entity, and the subject has to have access to the category into which the information has been categorised, to gain access. In general, labels are partially ordered, usually forming a lattice. A lattice is a partially ordered set where any subset has a greatest lower bound and a least upper bound. The requirement that a subject's clearance has to dominate an entity's classification to be able to read the entity is usually referred to as the simple security property (from the Bell-LaPadula model [1]). In addition, subjects are not allowed to write information to 'lower' classified containers since that will allow other subjects to access the information—some subjects who were not authorised to access information in its original location. Preventing information to flow from higher classified containers to lower classified containers are referred to as information flow controls. In the Bell-LaPadula model the requirement that information may only be 'written up' is known as the $\star$-property. The system security officer (SSO) is a person or group responsible for assigning clearances and classifications.

Role-based access control (RBAC) models group subjects into roles (based on their role or function in the organisation) and decide which roles need to access which entities based on the jobs the various roles have to perform.

See [12] for more information on basic security models.

The architecture proposed in this paper uses object-oriented concepts to function. To a slightly lesser extent, it is also assumed that the system to be protected is based on object-oriented concepts. The essentials of object-oriented systems as assumed in this paper, are

- Objects are instances of classes;
- Objects encapsulate instance variables and methods;
- Methods are the only way to access objects—messages are sent to an object to accomplish this;
- Classes are also objects—objects are instantiated by sending a `Create` message to the class; and
- Classes may be derived from other classes as subclasses.

See [4,2] for further information about object-orientation. If a system that is not object-oriented needs to be protected, the system may be wrapped in an object-oriented layer. In particular, in the case of a relational database management system, the query language statements (such as SELECT, INSERT, etcetera) will be handled like messages by the wrapper and issued in their native form to the database.

A fair amount of research has been done to investigate the security properties of object-oriented systems, the use of object-oriented systems to enforce security, or both. See [13,11,8,7] for examples.

3 The architecture

Figure 1 depicts one way in which ConSA is intended to be used. A human user accesses the system via some interface. Before allowed access to the system at all, the user has to be authenticated by the authentication module. The user is given some token that uniquely identifies him or her. Let us assume for simplicity during the initial discussion that it is simply a user name (but digitally signed certificates are obviously also one of a number of possibilities).

Once the user has been authenticated the interface sends a logon message to the Subject Management Module (SMM) of ConSA. (The SMM will be discussed in detail below.) The SMM uses the user name, does a table lookup to find the ConSA subject label associated with the given user and associates this label with all subsequent messages, until the user sends a log off message. All messages sent by the user between the logon and log off messages are

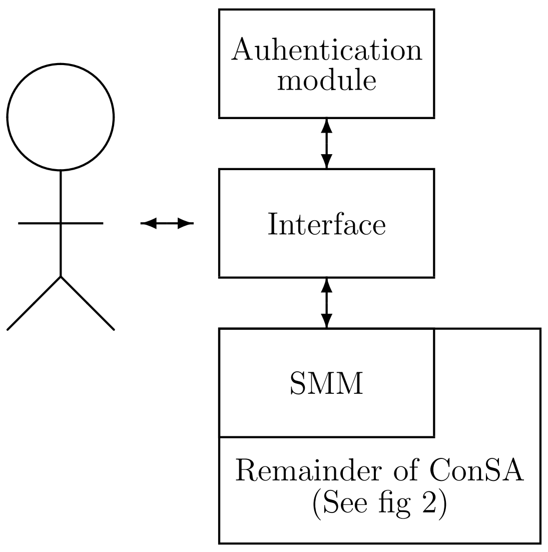


Fig. 1. Using ConSA

relayed by the SMM to the protected applications.

Although the scenario above was described in terms of a human user accessing an entire system, it is also possible that a subset of applications are protected by ConSA. In that case the user (or non-ConSA applications) will access the ConSA-protected applications via some suitable interface.

The proposed architecture contains three orthogonal dimensions. Firstly, access control is enforced. This roughly corresponds to the enforcement of the familiar simple security property (but is obviously not restricted to multilevel security). The second dimension optionally enforces flow control. This roughly corresponds to the $\star$-property, but is again not restricted to multilevel security. The third dimension determines who assigns security labels and how they are assigned.

ConSA is outlined in figure 2. At the bottom the *Labels* module defines the label classes on which simple access decisions are based (dimension 1). On top of that the *Information Flow Module* (IFM) handles flow control (dimension 2). The *Authorisation Control Module* (ACM) controls *who* is authorised to authorise other subjects to access entities (dimension 3). The *Subject Management Module* (SMM) is used by the security manager to initialise and manage the security system. Amongst others, it determines the user interface used for management. These components are used to enforce security in one or more systems, be it an application, database, operating system, other system or a combination of these. For ease of reference, this system will be referred to as the *application*. ConSA prescribes two facilities in the protected application. Firstly, the message dispatcher is expected to send particular messages to ConSA modules before and after any application message is sent. The idea is that any message sent by the application to some application method is sent to

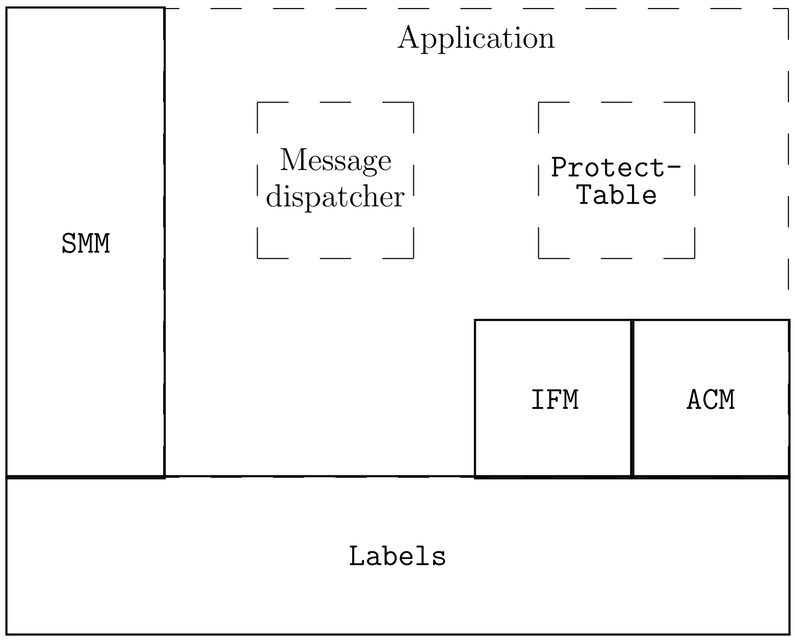


Fig. 2. ConSA

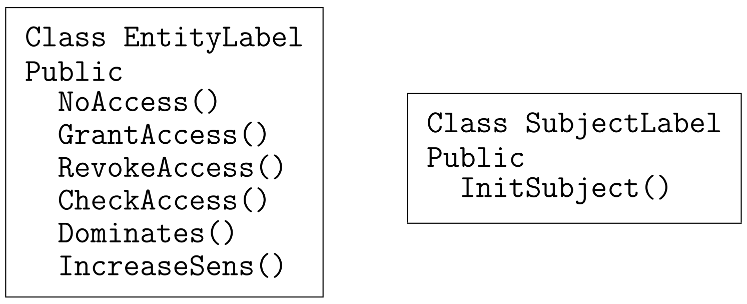
the message dispatcher which redirects it to the appropriate code. When the activated method terminates, it also returns to the activating method via the message dispatcher. Messages to read or write variables are also channelled via the message dispatcher. Secondly, the message dispatcher has to provide a facility (`ProtectTable`) where ConSA modules may register how entities in the application have to be protected. Such 'registrations' influence the parameters of messages sent to ConSA modules by the message dispatcher.

ConSA will be described by describing the operation of each of these modules.

3.1 The fundamentals

As mentioned earlier, ConSA uses object-oriented concepts. The fundamental notion to implement access control is that of two label classes. The structures of entity and subject labels are depicted in figure 3. The methods associated with the labels will be explained below. Do note that we will demonstrate that these methods are sufficient to support a wide range of actual security models.

Entity labels are associated with protected entities, while subject labels are associated with subjects. In some cases a system will also maintain a number of subject labels as templates that are not associated with any particular subject. The internal operation of the labels is not shown and is indeed adaptable. This is the key to the operation of ConSA: by modifying only the implementation of these two classes (and the other ConSA modules to be discussed later), the entire nature of the security system may be tailored for specific requirements.



```
Class EntityLabel
Public
  NoAccess()
  GrantAccess()
  RevokeAccess()
  CheckAccess()
  Dominates()
  IncreaseSens()
```

```
Class SubjectLabel
Public
  InitSubject()
```

Fig. 3. Entity and subject labels

The following notation will be used to facilitate the description of ConSA. Let S be the set of subjects, $\mathcal{S}$ be the set of subject labels and $\mathcal{E}$ the set of entity labels. If a particular subject label L_s will grant an associated subject access to an entity protected by an entity label L_e, this will be denoted by writing $L_s \succeq L_e$. The definition of $\succeq$ depends on the implementation. Let $auth_0(L_s) \subseteq S$ be the set of subjects directly associated with a subject label L_s. The notion of a subject being *directly associated* will be described later, but refers to subjects that may use a subject label because it has been explicitly (versus implicitly) granted permission to do so. Let $subj(L_e) \subseteq S$ be the set of all subjects authorised to access an entity protected with $L_e \in \mathcal{E}$. Clearly

$$subj(L_e) = \{s \in S | s \in auth_0(L_s), L_s \succeq L_e\} \tag{1}$$

Sometimes associating a subject with some subject label implies further access rights. For example, protecting an entity with a secret label label in a military environment not only grants access to users with a secret clearance, but also implies access for subjects with a top secret clearance. For this reason, we will define $auth(L_s)$ as the set of subjects associated with L_s or any 'implied' subject labels. To do this formally, define $ent(L_s) \subseteq \mathcal{E}$ as the set of all entity labels for which access will be granted if presented with the subject label L_s. Clearly $ent(L_s) = \{L_e | L_s \succeq L_e\}$. Further, say that a subject label L_{s1} dominates another subject label L_{s2} if the former allows access to all entities that the latter allows, and perhaps more. Formally

$$L_{s1} \geq L_{s2} \iff ent(L_{s1}) \supseteq ent(L_{s2}) \tag{2}$$

Then $auth(L_s)$ is defined as follows:

$$auth(L_s) = \{s \in S | s \in auth_0(L_{s2}), L_{s2} \geq L_s\} \tag{3}$$

Informally this definition may be illustrated by pointing out that, in the military model, if access is granted at a given clearance level, access is also implied to subjects cleared 'higher' in the hierarchy. If access to L_e is granted to *secret* then $auth_0(L_e) = \{secret\}$ and $auth(L_e) = \{secret, top\ secret\}$.

The intention of each of the methods in the label classes depicted in figure 3 is given below. Remember that we assume that security components will protect some ‘application.’ The security components have, amongst others, to implement the label classes and hence the methods associated with the label classes. The protected application will then send appropriate messages to instances of these classes and expect a certain behaviour of these objects. The description below describes the behaviour expected by the application; obviously the label classes also have to demonstrate that they implement the desired security policy correctly.

Before a more formal description of the methods is given, the idea behind each method will be described: `NoAccess` is used to initialise an entity label such that no subject will be allowed any access to an entity protected with this label. `GrantAccess` may now be used to add subjects (actually subject labels) to the list of those who will be permitted access to an entity protected with the concerned entity label. Note that models where an ‘owner’ automatically gets access to any newly created object, a `NoAccess` message will have to be sent — prohibiting even the owner from accessing the object — followed by a `GrantAccess` message. Detail on how these messages may be combined and isolated in a component will be given in section 3.4.

`RevokeAccess` is the inverse of `GrantAccess`. `CheckAccess` accepts a subject label and compares it to the concerned entity label; it replies with a *yes* or *no* to indicate whether access should be allowed or not. `Dominates` and `IncreaseSens` are only required if flow control is to be implemented; `Dominates` checks whether one subject label is ‘more protective’ than another while `IncreaseSens` adjusts an entity label such that it is as protective as it was, but also to be at least as protective as the label sent as an argument. The notion of protection (‘more,’ ‘at least as,’ etc) will be described below. `InitSubject` is used to associate subject labels with subjects.

From this informal description it is clear that the protected application will use `CheckAccess` to verify that the user is authorised to access any given entity and only access it if permitted. Note that the message dispatcher acts as a reference monitor: since all messages are directed via it, it may call the appropriate `CheckAccess` method, before it activates the requested action. `NoAccess`, `GrantAccess` and `RevokeAccess` are used by the appropriately authorised individual(s) to grant and revoke access rights. We will show below that it is simple to configure the systems such that the *individual(s)* may be the SSO, the ‘owner’ of the concerned entity or whoever else the implemented security model wants to allow. `NoAccess`, `GrantAccess` and `RevokeAccess` may, if required, also be used by `CheckAccess` if a more dynamic security model is being implemented. Since `CheckAccess` is called whenever an access is made, the implementation of `CheckAccess` may send appropriate messages to `Dominate` and `IncreaseSens` to ensure that flow control is properly implemented.

The requirements that ConSA places on the various methods are given below as postconditions for entity labels. How it may be demonstrated that a given security model satisfies these postconditions will be shown in section 4.

`NoAccess` initialises the entity label such that no subject will be allowed to access an entity protected with it. If this message has been sent to a label L_e (where L'_e indicates the label after the message has been processed) then

$$subj(L'_e) = \emptyset$$

After a label has been thus initialised it may be manipulated with, amongst others, `GrantAccess` and `RevokeAccess`.

`GrantAccess` (sent to an entity label) accepts a subject label as a parameter; it makes the necessary modifications that, if a subject presents the concerned subject label to an entity protected with the concerned entity label, access will be granted. To illustrate, consider a DAC system: Here an entity label may consist of an access control list, while a subject label may merely consist of a user identifier. `GrantAccess` may then simply add the user identifier to the ACL.

In the case of an MLS system both labels may consist of numbers, with the higher number indicating a higher sensitivity or clearance. Here the implementor may choose to increase the number contained in the entity label to that contained in the subject label. This obviously will allow access to 'owners' of the given entity label.

Formally, if a `GrantAccess` message is sent to an entity label L_e (with L'_e again the label after the message has been handled) the effect of this will be

$$subj(L'_e) = subj(L_e) \cup auth(L_s)$$

`RevokeAccess` is the mirror image of `GrantAccess`: It modifies the given labels such that access will not in future be allowed to subjects presenting the concerned subject label to entities protected with the particular entity label. One problem that has to be catered for is the fact that authorisations often imply other authorisations. In a typical military model, removing access from those that are cleared secret, will have no effect if the label still allows access to subjects who are unclassified. This means that, if one wants to revoke access from those cleared secret, access also has to be revoked from all lower cleared subjects. To handle this, two versions of `RevokeAccess` are supported, selected by a parameter.

The first version of `RevokeAccess`, denoted by `RevokeAccess*` will definitely revoke access: if the subjects identified by the subject label in the `RevokeAccess*` are still authorised by other subject labels, access via such labels also has to be revoked. The second version of `RevokeAccess`, denoted by `RevokeAccess0`, only revokes access from those directly associated with the subject label specified in the `RevokeAccess0` message. Those subjects for which access is implied via other labels will still have access. To illustrate, in the simple military model, if access to some entity is allowed for confidential subjects, it is implied for secret and top secret subjects. If access is revoked for confidential users with either version of `RevokeAccess`, access will still be allowed for secret and top secret subjects. If access is revoked for secret subjects with `RevokeAccess*` then access will also be revoked for confidential subjects—otherwise the fact that confidential subjects may still access via the label, will imply that secret users may also still access via the label. On the other hand, `RevokeAccess0` will have no effect in this case—since all secret subjects still have access via the confidential label.

Formally, with the same assumption as for `GrantAccess`, the effect of a `RevokeAccess*` message is

$subj(L'_e) \subseteq subj(L_e)$,

$subj(L'_e) \cap auth_0(L_s) = \emptyset$, and

$(\forall K \in \mathcal{E})(subj(K) \subseteq subj(L_e) \wedge subj(K) \cap auth_0(L_s) = \emptyset \Rightarrow subj(K) \subseteq subj(L'_e))$

The first requirement stems from the fact that, by removing access, no other subject should be granted access. The second requirement states that the new label should not allow access to anyone from whom access has just been revoked. The final requirement states that the new label should be 'maximal': no access should be revoked unnecessarily.

Formally, with the same assumption as for `GrantAccess`, the effect of a `RevokeAccess0` message is

$$subj(L'_e) = subj(L_e) - [auth(L_s) - \bigcup_{L_{s2} \in \mathcal{S}, L_{s2} \succeq L_e, L_{s2} \neq L_s} auth(L_{s2})]$$

The logic behind this specification is that access is effectively only revoked from those subjects who have no 'other' authorisation to access entities protected with L_e.

Note that, if no subject authorisations are implied, in other words if

$$(\forall L_s \in \mathcal{S})(auth_0(L_s) = auth(L_s))$$

the two versions of `RevokeAccess` are identical. We omit the proof.

`CheckAccess` compares the presented subject label with the particular entity label; if access should be allowed `CheckAccess` returns a true, otherwise a false.

Formally, `CheckAccess` returns true when

$$L_s \succeq L_e$$

where L_e is the target entity label and L_s is the subject label sent as parameter.

`Dominates` compares two entity labels; it returns *true* or *false*, depending on whether the target label dominates the label sent as a parameter of this message. Note that this method is only required if flow control is required (see section 3.3). Also note that it is possible to implement this method even if the label classes do not implement traditional multilevel labels. We postpone the discussion of this point to section 3.3.

Formally, if L is the target label and M the label sent as parameter, `L:Dominates M` (also written as $L \geq M$) is defined as follows

$$L \geq M \iff subj(L) \subseteq subj(M)$$

The motivation for this definition is that an entity e_L protected with L may be considered as more sensitive than an entity e_M protected with M if only a subset of those subjects authorised to access e_M are authorised to access e_L (see [11]).

`IncreaseSens` is also only required if flow control is required. It will be used when information e_M protected with a label M is somehow combined with (or 'added to') information e_L protected with a label L. In such a case flow control requires that e_L is now only accessible to subjects who were previously authorised to access both e_M and e_L.

Formally (in the ideal case), `L:IncreaseSens M` should reflect the combined sensitivity of both labels in L' such that

$$subj(L') = subj(L) \cap subj(M)$$

If a label L' exists that satisfies this condition, this label will be denoted by $\lceil L, M \rceil$. If it is not possible to construct such a label L', a label L'' has to be constructed such that

$$subj(L'') \subset subj(L) \cap subj(M)$$

where L'' is selected to be as large a subset (in some sense) as possible. This latter case is not considered further in the current paper. See section 3.3 for further information on the use of `IncreaseSens`.

`InitSubject` is used to initialise subject labels. The parameter sent to it depends on the particular security model implemented and may be a user identifier (for traditional DAC models), a level (for traditional MLS models), a role (for RBAC models), and so on. We will later argue that the potential diversity of this parameter does not restrict the generality of ConSA.

Formally, if L_s is the subject label, and $\sigma \subseteq S$ somehow represents a set of one or more subjects, then the effect of L_s`:InitSubject` σ is

$$auth_0(L_s') = \sigma$$

where L_s' is the initialised subject label.

An example of how the classes described above may be used to implement a traditional security model will be given in section 4.

3.2 Dimension 1: Access control

Given the classes described above, it is simple for a system based on ConSA to enforce access control: Whenever a protected entity is to be accessed, the system simply has to identify the entity label associated with the entity, the subject label associated with the subject and then send a `CheckAccess` message to the entity label with the subject label as parameter.

If the notation $l_e(\mathrm{E})$ is used to indicate the entity label of an entity E and $l_s(\mathrm{M})$ to indicate the subject label of the subject who sent message M, then M is allowed to proceed to access E if

$$l_e(\mathrm{E}){:}\texttt{CheckAccess}\ l_s(\mathrm{M})$$

returns true.

It is expected that the protected system is able to associate labels with all objects, methods and instance variables. The message dispatcher component of the application is responsible for checking that access control checks as above are performed whenever an entity associated with a label is accessed. See section 5 for remarks on the implications this may have on efficiency.

3.3 Dimension 2: Flow control

Flow control has to ensure that a subject authorised to access information, cannot copy it to a location where a perviously unauthorised subject is now able to access it. The classical example is the *-property of the Bell-LaPadula model. Flow control is, however, not restricted to multilevel models such as the military model [11].

To enforce flow control, either content or container-based security may be used. (Hybrid policies are also possible.) In the case of content-based security, the label associated with a variable is determined by the contents of the variable; if information is copied from one variable to another, the label is also copied. This obviously ensures that information cannot be disclosed by copying it. The fact that a variable's label changes may, however, be used as a covert channel: The fact that a variable which was previously accessible to a subject is no longer accessible may tell the subject something about the contents of the variable — particularly if the subject is familiar with (or able to infer) the labelling policy used.

The alternative to content-based labelling is container-based labelling. Here the variable is labelled and if information is copied from a variable x to a variable y, it is necessary to ensure that everybody who is able to read y was able to read x anyway.

Hybrid policies have also been proposed, where a content-based policy is essentially used, but where the labels that may be associated with data in a given container is restricted to be within some predetermined range [8].

In an object-oriented system, data is copied by methods. To complicate matters, data is often not simply copied, but 'collected' from several variables, combined in some way and then written to a destination variable.

In many secure object-oriented models the method activation is also viewed as a container of information and very often content-based security is used for this container (although it is obviously not the only possibility).

A secure object-oriented system therefore has to know (1) when data 'flows into' a method activation; and (2) when data flows from a method activation to a variable (*ie* when the method writes to the variable). The first case may be further subdivided into the cases where

1.1) A method reads a value from a variable;
1.2) A method sends a value to another method it activates; and
1.3) A method receives a return value from another method it has invoked.

These items are numbered 1.1 etc because they refine point 1 above.

It is clear that even from these simple choices, a variety of possible flow control policies exist, for example

- Use content-based labelling for the method activation and container-based labelling for variables;
- Use content-based labelling for both method activation and variable labelling; or
- Use container-based labelling for both method activation and variable labelling.

In addition, the (initial) label of a method activation needs to be decided on. (If content-based labelling is used for method activation, this decision determines the initial label; if container-based labelling is used, this decision determines the label for the duration of the activation's existence.)

Finally, information flow from one method activation to another needs special attention: Obviously, if values are sent or returned as parameters (or return values), the appropriate content or container-based rules have to be enforced. However, even if no parameters (values) are transferred, the mere fact that a method has been called may be used as a covert channel and information deemed do have flowed. Similarly, a method that does not return any value, may use time (before it returns) as a timing channel. These facts may be more or less of a concern in a particular environment and influence the particular flow control policy adopted.

Rather than using a simple label to protect the method activation, an Information Flow Management (IFM) object will be used.

Remember (from section 3.1) that `CheckAccess` is to be called by the message dispatcher in the protected application whenever an entity is to be accessed. This includes method activation and access to any variables. We now extend this requirement by expecting the dispatcher to also call one of the following methods from the concerned IFM object when appropriate:

- `ActivateFlow(Meth, Parameters)`: If a_1 activates a_2 then this message has to sent to the IFM of a_1. On return, `Meth` will contain an IFM to be used for a_2. If $l_E(a_1)$ and $l_E(a_2)$ denote the entity labels managed by the two IFM objects, then the minimum requirement (not catering for covert channels) if information flow control is desired, is

  $$\neg Parameters \vee subj(l_E(a_2)) \subseteq subj(l_E(a_1))$$

  This follows from the requirement that, information should not flow to any location where it can be accessed by subjects not authorised to access the information in its original location.
- `ReturnFlow(Meth, ReturnVal)`: If control returns from a_2 (managed by `Meth`) to a_1 and `ReturnVal` is a Boolean indicating whether any values are returned from a_2 to a_1, then the minimum requirement is

  $$\neg ReturnVal \vee subj(l_E(a_1)) \subseteq subj(l_E(a_2))$$

- `ReadFlow(Var)`: If a reads a variable v (managed by `Var`, then information flow requires that

  $$subj(l_E(a)) \subseteq subj(l_E(v))$$

- `WriteFlow(Var)`: If a writes to a variable v (managed by `Var`), then information flow requires that

  $$subj(l_E(v)) \subseteq subj(l_E(a))$$

The IFM makes use of the notion of *sessions*, but the particular definition of a session depends on the implementation of the specific IFM. A session may last for execution of a single method, a single transaction, from the point when a user logs onto the system until the user logs off, or even for the entire time that a user is known to the system. For this reason the IFM also supplies the following 'checkpoint' methods:

- `LogOn( subj )` is to be called when a new subject logs onto the system. This is a class method, that returns an instance of an IFM object to handle subsequent messages.
- `LogOff` is to be called when the subject logs off the system. It will typically destroy the IFM object that handled flow control for the period logged on.
- `NewQuery` has to be called by the message dispatcher whenever it receives a new message from the user; a subsequent `NewQuery` (or `LogOff`) indicates to the IFM object that the previous query has been completed.
- `Register( subj, parm )` and `DeRegister( subj, parm )` will be discussed below.

It is now possible to implement a wide range of information flow policies. A typical container-based flow control policy, but with content-based activation records (similar to that used by SODA [8] and many others) may be implemented as described in figure 4. Note that `All` is assumed to be a subject label associated with all subjects. [1]

This example not only illustrates that it is indeed simple to precisely specify a common information flow policy, but also makes it possible to point out a number of (subtly) different alternatives. For example, by changing `WriteFlow` to `Var:IncreaseSens( actLabel )` implements a content-based policy.

A different variation is to initialise the activation label at the (clearance) level of the logged-in subject, rather than at the 'system low' level used in figure 4. It is also possible to regulate a user's ability to manually move information to an inappropriate location by keeping track of a user's actions over multiple queries.

The two methods not yet discussed, `Register` and `DeRegister` are intended to support policies that depend on the subject who is performing the actions; a trusted subject may, for example, be allowed to indeed copy information to an otherwise 'inappropriate' location. `Register` is intended to 'register' a subject with the IFM. `DeRegister` removes this subject again. The format of the parameters of `Register` and `DeRegister` are, similar to those of `InitSubj`, not precisely specified. `Register` and `DeRegister` are only intended to be used by the SMM. It will be argued later that this potential variety does not affect the generality of the model. Also note that the existence of the `Register` and `DeRegister` methods make it possible to specify flow control policies that span

[1] If `All` does not exist in a given environment, alternative mechanisms do exist that will enable us to achieve the same goal; we omit a discussion.

```
class IFM;
  class method LogOn():IFM {
    return new IFM();
  }

  method LogOff() {
    destroy( this );
  }

  EntityLabel actLabel = new EntityLabel;
    // This is the activation label that will be used to
    // protect the method activation managed by this IFM instance

  method NewQuery() {
    actLabel:NoAccess;     // Make the activation label
    actLabel:Grant( All ); // accessible to All since it does
  }                        // not contain any information yet.

  method ActivateFlow(Meth, Parameters) {
    Meth = new EntityLabel;
    Meth:NoAccess();
    Meth:Grant( All );
    if (Parameters)
      Meth:IncreaseSens( actLabel )
  }

  method ReturnFlow(Meth, ReturnVal) {
    if (ReturnVal)
      actLabel:IncreaseSens( Meth.actLabel );
    Meth = null;  // The label for the returning method is no
  }               // longer required

  method ReadFlow( Var ) {
    actLabel:IncreaseSens( Var.entityLabel );
  }

  method WriteFlow( Var ) {
    if actLabel:Dominates( Var.entityLabel )
      abort;
  }
```
Fig. 4. Container-based variable with content-based activation information flow management

multiple queries issued by a given subject.

3.4 Dimension 3: Authorisation

In traditional mandatory security models, labels are assigned centrally, by a system security officer (SSO) or other responsible party. In traditional DAC models, owners of entities have the right to grant access to their entities to other subjects. The right to grant access may usually also be granted. One typically becomes the owner of an entity by creating the entity.

In ConSA the right to grant access to an entity E is determined by the right to send a *GrantAccess* message to the label that protects E (denoted by l_e(E)). Since labels (both entity and subject) are objects themselves, they are also protected by entity labels. Protecting labels with other labels does not necessarily lead to an infinite progression of labels: it is possible to have a 'root' label that not only protects other labels, but also protects itself. This situation is similar to classes in Smalltalk [4] also being objects and therefore having classes themselves. Our solution for labels is similar to Smalltalk's solution for such (meta-)classes. More details about this 'root' entity label will be given later.

To illustrate how authorisation is controlled, assume that a traditional MAC system is to be implemented where only the SSO is allowed to grant and revoke access. Consider some entity E that needs to be protected.

Firstly, an entity label is required to protect E. We will refer to this label as `AccessRight`. We assume that some subject associated with a subject label S is to be granted access; if more subjects are to be granted access, this is accomplished similar to granting this subject access. ConSA expects that the protected system implements a (possibly virtual) table, `ProtectTable`, that maintains a list of all protected entities, with the entity label that protects it. The system is expected to `CheckAccess` with the appropriate entity label before allowing access to any entity in the system. More details about `ProtectTable` will be given later. `AccessRight` is then created as follows.

```
AccessRight = EntityLabel:Create
AccessRight:NoAccess
AccessRight:GrantAccess S
                   -- Authorise subject S to enter via AccessRight
ProtectTable:Protect E AccessRight
                   -- Protect entity with this label
```

After creating `AccessRight`, it is necessary to ensure that only the SSO is able to modify it (in other words, to grant additional access for E, or to revoke access for E). To do this, another label is created that will be referred to here as `ControlRight`. Suppose that `SSO` is a subject label only associated with the SSO, then `ControlRight` may be implemented as follows:

```
ControlRight = EntityLabel:Create
ControlRight:NoAccess
ControlRight:GrantAccess SSO
                  -- Allow SSO access via ControlRight
ProtectTable:Protect AccessRight.NoAccess ControlRight
ProtectTable:Protect AccessRight.GrantAccess ControlRight
ProtectTable:Protect AccessRight.RevokeAccess ControlRight
ProtectTable:Protect AccessRight.IncreaseSens ControlRight
                  -- Only allow SSO to modify access right
```

In the sequel `Modify` will be used as a shorthand for all methods that may modify the concerned entity label. In other words, when we write

```
 ProtectTable:Protect AccessRight.Modify ControlRight
```

it will have the same intention that the four statements in the previous code fragment had.

It is now necessary to ensure that `ControlRight` is not modified, because that will enable a perpetrator to subvert the security system. Consider the following:

```
 Root = EntityLabel:Create
 Root:NoAccess    -- Allow no access
 ProtectTable:Protect Root.Modify Root
                  -- Disable further modification of Root
 ProtectTable:Protect ControlRight.Modify Root
                  -- Disable further modification of
                  -- ControlRight
```

We will denote the fact that an entity E is protected by a label L, where L grants access to subjects with subject labels $S_1, S_2, \ldots S_n$ as follows:
$\langle E, L \rangle, \qquad L = \{S_1, S_2, \ldots S_n\}$

The effect of the code given above is to establish a series of labels as follows:
$\langle E, \texttt{AccessRight} \rangle, \qquad \texttt{AccessRight} = \{S\}$
$\langle \texttt{AccessRight.Modify}, \texttt{ControlRight} \rangle, \qquad \texttt{ControlRight} = \{\texttt{SSO}\}$
$\langle \texttt{ControlRight.Modify}, \texttt{Root} \rangle, \qquad \texttt{Root} = \{\}$
$\langle \texttt{Root.Modify}, \texttt{Root} \rangle$

Clearly, this will properly implement mandatory security.

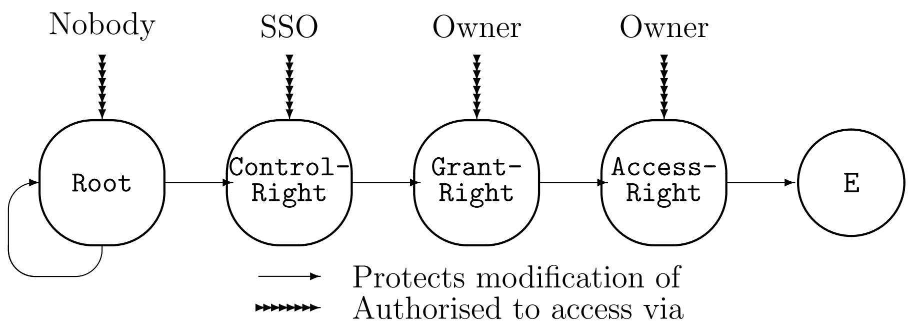


Fig. 5. Discretionary protection system where SSO controls ownership

Note, for the code given above, that it is not necessary to recreate `Root` and `ControlRight` for every entity, and `AccessRight` may even be re-used in some cases.

As a second example, consider traditional discretionary access controls. Suppose a subject with subject label `S` sends a message that creates an entity E. Suppose further that the following labels are established:

$\langle E, \texttt{AccessRight}\rangle$, $\texttt{AccessRight} = \{\texttt{S}\}$
$\langle \texttt{AccessRight.Modify}, \texttt{GrantRight}\rangle$, $\texttt{GrantRight} = \{\texttt{S}\}$
$\langle \texttt{GrantRight.Modify}, \texttt{ControlRight}\rangle$, $\texttt{ControlRight} = \{\texttt{SSO}\}$
$\langle \texttt{ControlRight.Modify}, \texttt{Root}\rangle$, $\texttt{Root} = \{\}$
$\langle \texttt{Root.Modify}, \texttt{Root}\rangle$

The protection system implemented above is graphically depicted in figure 5. Since nobody has the `Root` right, `ControlRight` cannot be modified, and the SSO is (permanently) the only subject who can use `ControlRight`. This means that the SSO is the only subject who can modify `GrantRight`: the SSO will use this authority to grant 'owners' the right to use `GrantRight`. These owners may now use `GrantRight` to modify `AccessRight` to grant other subjects access to `E`.

Now consider the following, where `AccessRight` and `GrantRight` are implemented as in the previous discretionary access control case, but where `Root` is omitted and `ControlRight` is implemented differently:

$\langle E, \texttt{AccessRight}\rangle$, $\texttt{AccessRight} = \{\texttt{S}\}$
$\langle \texttt{AccessRight.Modify}, \texttt{GrantRight}\rangle$, $\texttt{GrantRight} = \{\texttt{S}\}$
$\langle \texttt{GrantRight.Modify}, \texttt{ControlRight}\rangle$, $\texttt{ControlRight} = \{\texttt{S}\}$
$\langle \texttt{ControlRight.Modify}, \texttt{ControlRight}\rangle$

In this version, the owner can grant ownership to another subject, after which any of the owners can revoke access from any owner.

The preceding discussion showed that it is possible to use the underlying infrastructure to implement various approaches to control authorisation. Other possibilities are also possible. However, it is still necessary to argue that the various approaches may be isolated into a few components, that may be interchanged with other components so that an organisation can decide on a particular component that suits their needs and ‘plug’ such a module into their system. Consider the following definition:

```
Object ACM
  SetSSO(SSO)
  EntityCreated(E, Ms, Me)
```

`SetSSO` is normally activated once only to initialise the ACM object such that it ‘knows’ who is the subject ultimately responsible for security management. If such a subject is not defined in a particular environment, a suitable label associated with no subject may be used. `SetSSO` may initialise labels such as `Root` and `ControlRight` described above.

`EntityCreated` is activated when any new protectable entity has been created. It creates (or finds) suitable labels and sends an appropriate message to `ProtectTable` to effect protection of the newly created entity E. Both the subject label of the message (`Ms`), identifying the subject under whose authority the message is functioning, and the entity label (`Me`), currently protecting the contents of the message activation are sent as parameters to `EntityCreated` and may be used by the specific implementation of ACM to protect the newly created entity.

Note that the previous examples of MAC and DAC implementations only serve as examples; `SetSSO` and `EntityCreated` may use any suitable implementation strategy.

3.5 Labelling subjects

Note that all interaction between the protected system and the two label classes, IFM classes and other ConSA objects described above are completely independent from the implementation of the label classes: The implementation details are completely isolated in the two classes. There are, however, three exceptions that have not received attention yet: The parameters sent to `InitSubject`, `Register` and `DeRegister` do depend on the selected access control model. `InitSubject` may, for example, be sent a user identifier if access control lists are used, it may be a number if numeric security levels are used, and so on. However, it is possible to argue that all messages to this method may be isolated in a fourth component (see figure 2). We will refer to this fourth component as the *subject management module* (SMM).

The fact that the SMM forms a layer through which all communication between the external world and the ConSA-protected world occurs has already been mentioned in section 3. It is now necessary to distinguish between two categories of communication with the outside world: We will consider communication between the SSO and ConSA for administrative purposes first. The normal communication between the user and the protected applications (noting that the 'user' may in fact be a program outside the ConSA-protected system) will be considered after that.

One of the primary tasks performed by the SSO is the registration of new users. To this end the SMM provides a table with all user names and corresponding subject label. The SSO can send messages to add, edit or delete entries in this table. How it is ensured that only the SSO can make these changes will be discussed below. Note that the 'user names' may be names of real users, clearance levels for MAC systems, etc. In cases where real names have to be mapped to one of these identifiers, it is the responsibility of the authentication module to provide this mapping. Stated differently: The SSO may use the SMM to enter identifiers such as *restricted*, *secret* and *top secret*. The SSO may interact with the authentication module to ensure that when, say, John logs on, John is provided with a *secret* identifier which is passed to the SMM when his logon is processed by the SMM. Similarly, Jane may be given a *top secret* identifier, which is passed to the SMM when her logon is processed.

When a user logs in from the interface, the user is expected to provide an authenticated user identifier. This is obtained from the authentication module as mentioned in the previous paragraph, but not discussed in detail in the current paper. The SMM maintains a table with these user identifiers and their associated subject labels; the appropriate subject label is associated with all subsequent messages received from the interface until the user logs off. Since the ConSA components are themselves protected by ConSA mechanisms it is straightforward to protect the SMM methods that add, edit and delete subjects such that only the SSO is able to invoke these methods.

For 'normal' user messages, the SMM simply passes the message with the associated subject label on to the message dispatcher of the application. Given these functions of the SMM it is simple for the SMM to send `Logon`, `Logoff` and `NewQuery` messages to the IFM when appropriate. These messages may be sent at the obvious times or, if a different notion of sessions (for the IFM) is required, may be sent at other appropriate times.

Authorisation as described in section 3.3 occurs in terms of subject labels. However, it is often more logical to work in terms of user identifiers. To enable this to happen, the SMM provides a translation service from the external identifier (expressed as a character string) to the corresponding subject label. Suppose, for example, that the owner of some entity wants to authorise some user `X` to access an entity. This subject may then send the message

SubjectLabel = SMM:Translate ‘X’

and then proceed to authorise the subject as described previously.

Note that all interaction between the protected system and the two label classes, IFM classes and other ConSA objects described earlier are completely independent from the implementation of the label classes: The implementation details are completely isolated in the two classes. There are, however, three exceptions that have not received attention yet: The parameters sent to `InitSubject`, `Register` and `DeRegister` do depend on the selected access control model. `InitSubject` may, for example, be sent a user identifier if access control lists are used, it may be a number if numeric security levels are used, and so on. However, it is possible to argue that all messages to this method may be isolated in the SMM. It is further possible to argue that the user name (whether it is a name of a real user, a clearance level or some structured certificate) may be represented as a character string, enabling independence between the SMM and subject label.

Remember that `Register` and `DeRegister` are provided by the IFM to assign special rights to certain users. Exactly what those rights are and how they are interpreted depend entirely on the IFM. However, it is the responsibility of the SMM to inform the IFM when such users are entered into the lookup table of the SMM. Note that these messages may only be sent by the SMM. Again, for generality, we assume that a character string is used to encode the special privileges of such a user. Assume for example that some user is allowed to copy information from a high sensitivity level to a lower level. Suppose the IFM uses the character string “trusted” to indicate this privilege. The role of the SMM in this regard is to allow the SSO to enter this character string when entering new identifiers (or editing existing ones). Whenever a new string is entered, the SMM sends a `Register` message to the IFM, with the appropriate subject label and this string as parameters. When an existing identifier, with which such a string was associated, is deleted (or modified) a `DeRegister` message is sent to the IFM. (Obviously, in the case of modification, the new string, if any, is subsequently sent with a new `Register` message.) This establishes the independence between the SMM and IFM.

3.6 Authentication

A fifth, and final, component is the authentication component. This component enables a user to enter an external identifier and somehow prove this identity. The precise means of authentication is encapsulated in this module and therefore configurable: Any authentication technique may be used; we only assume that this component enables subsequent messages sent by the particular user to be sent under cover of the appropriate subject label. No further details about the authentication module are given in this paper. For this reason it is also not shown in figure 2.

4 A role-based example

This section demonstrates the use of ConSA to implement a role-based access control system.

For this example we will make the following assumptions:

(1) Roles are arranged in a lattice.
(2) Roles higher in the lattice at least have all the rights that roles directly below them have. A manager therefore automatically has all the rights a secretary lower in the lattice has, without the need for the manager to explicitly log in in the secretary role.
(3) Any individual subject is a member of precisely one role.

Note again that a different implementation may make other choices for any or all of these assumptions; ConSA does not rely on any of them.

We will use the following notation to expound this example: r_i will be used to denote a role i. Each role may be viewed as a set of subjects. This set of subjects denotes those subjects directly associated with the role: if $L_s = \{r_s\}$ then $auth_0(L_s) = r_s$. If r_i is a superior of r_j in the role lattice, it will be denoted by $r_i \geq r_j$. R will denote the set of all roles.

A subject label is a set containing a single role in which the associated subjects are authorised to function. An entity label is a set of roles that are allowed to access the protected entity. A subject is allowed to access the protected entity if it is authorised to function in any of these roles or any of its superior roles. Formally, if $L_s = \{r_s\}$ and $L_e = \{r_1, r_2, r_3, \ldots, r_n\}$ then $L_s \succeq L_e \iff (\exists r_i \in L_e)(r_s \geq r_i)$.

Given this choice of notation, it is now necessary to determine the specific meaning of ConSA concepts, such as *subj*, *ent* and *auth* for this example.

It follows from equation 1 that

$$subj(L_e) = \{s \in S | (\exists q \in R)(\exists r \in L_e)(s \in q \geq r)\} \tag{4}$$

Further $ent(L_s) = \{L_e \in \mathcal{E} | (\exists r \in L_e)(r_s \geq r)\}$. Then, if $L_{s1} = \{r_{s1}\}$ and $L_{s2} = \{r_{s2}\}$, it follows (from equation 2) that

$$\begin{aligned} L_{s1} \geq L_{s2} &\iff \{L_e|(\exists r \in L_e)(r_{s1} \geq r)\} \supseteq \{L_e|(\exists r \in L_e)(r_{s2} \geq r)\} \\ &\iff \{L_e|r \in L_e, r \leq r_{s1}\} \supseteq \{L_e|r \in L_e, r \leq r_{s2}\} \\ &\iff r_{s1} \geq r_{s2} \end{aligned}$$

Then it follows from equation 3 that

$$\begin{aligned} auth(L_s) &= \{s \in S|s \in auth_0(L_{s2}), L_{s2} \geq L_s\} \\ &= \{s \in S|(\exists r_s \in L_s)(\exists q \in R)(s \in q \geq r_s)\} \end{aligned} \tag{5}$$

As previously, if an entity label L labels more sensitive information than a label M, this will be indicated by writing $L > M$. If L and M label equally sensitive information, this will be indicated by writing $L \equiv M$. $L \leq M$ means either $L < M$ or $L \equiv M$.

Before the details of the role-based example will be given, it is necessary to define a number of useful concepts.

Let the *upward closure* of any label L be all roles in the role lattice that dominate any role of L. Denote the upward closure of L by $L\!\uparrow$. Informally, $L\!\uparrow$ is then all roles in the lattice that may be reached by starting at the roles in L and moving upward. Formally,

$$L\!\uparrow = \{r' \in R|(\exists r \in L)(r' \geq r)\}$$

Let the *downward closure* of any label L be all roles in the role lattice that are dominated by any role of L. Denote the downward closure of L by $L\!\downarrow$. Informally, $L\!\downarrow$ is then all roles in the lattice that may be reached by starting at the roles in L and moving downward. Formally,

$$L\!\downarrow = \{r' \in R|(\exists r \in L)(r' \leq r)\}$$

Let the *floor* of a label L be the set of roles in L that do not dominate any other role in L. Denote this by $\lfloor L \rfloor$. Informally $\lfloor L \rfloor$ is the roles that remain in L after all roles that have any roles 'below' them in L have been removed from L. Formally,

$$\lfloor L \rfloor = \{r' \in L|\neg[(\exists r \in L)(r < r')]\}$$

Theorem 1 *For any entity label L,*

$$L\!\uparrow \equiv L$$

PROOF. To label equally sensitive information, every subject allowed to access information protected by $L\uparrow$, also has to be able to access information protected by L and *vice versa*, thus it should be proven that

$$subj(L\uparrow) = subj(L)$$

Let $s \in subj(L)$. Then $s \in q$ for some role q and $(\exists r \in L)(q \geq r)$. Let $r' = r$ then, trivially, $(\exists r \in L)(r \leq r')$. Now $r = r' \in L\uparrow$. Since $s \in q \geq r' \in L\uparrow$ it follows that $s \in subj(L\uparrow)$.

Let $s \in subj(L\uparrow)$. Then $s \in q$ for some role q and $(\exists r' \in L\uparrow)(q \geq r')$. From the definition of ($\Uparrow L$)and since $r' \in L\uparrow$ it follows that $(\exists r \in L)(r \leq r')$. Since $s \in q \geq r' \geq r \in L$, it follows that $s \in subj(L)$.

Since $s \in subj(L) \iff s \in subj(L\uparrow)$, it follows that $L\uparrow \equiv L$. $\square$

Theorem 2 *For any label L,*

$$\lfloor L \rfloor \equiv L$$

PROOF. Let $s \in subj(L)$. Then $s \in q$ for some role q and $(\exists r' \in L)(q \geq r')$. Select r' such that $r' \not> r$ for any other $r \in L$ for which $q \geq r$. Then $\neg[(\exists r \in L)(r < r')]$ and hence $s \in q \geq r' \in \lfloor L \rfloor$. Therefore $s \in subj(\lfloor L \rfloor)$.

Let $s \in subj(\lfloor L \rfloor)$. Then $s \in q$ for some role q and $(\exists r' \in \lfloor L \rfloor)(q \geq r')$. Since, from the definition of $\lfloor L \rfloor$, $r' \in \lfloor L \rfloor \Rightarrow r' \in L$ it follows that $(\exists r' \in L)(s \in q \geq r')$. Therefore $s \in subj(L)$.

Since $s \in subj(L) \iff s \in subj(\lfloor L \rfloor)$, it follows that $\lfloor L \rfloor \equiv L$. $\square$

It is obvious that

$$L\downarrow \not\equiv L$$

As previously, the subject label of a subject S will be denoted by $l_s(S)$. We assume that this label only contains the (single) role to which S has been assigned. Similarly, the entity label of an entity E will be denoted by $l_e(E)$. We assume that this label contains the list of all roles who are permitted to access E.

`GrantAccess` will additionally grant members of the specified role access to the protected entity. This is accomplished by adding the role identifier to the list contained in the entity label. Formally, granting subjects in role r (with subject label L_r) access to an entity protected by a label L, modifies L to L'

such that $L' = L \cup \{r\}$. It is obvious that this fulfils the requirement that $subj(L') = subj(L) \cup auth(L_r)$.

`L:RevokeAccess*` L_r, with $L_r = \{r\}$ modifies L to L' such that $L' = \lfloor L\uparrow - L_r\downarrow \rfloor$.

PROOF. Let $N = L\uparrow - L_r\downarrow$. We will prove that N satisfies the requirements for `RevokeAccess*`. Then, since $N \equiv L'$, it follows that those conditions are also satisfied for L'.

The three requirements described for `RevokeAccess` in section 3.2, will be addressed in the order in which they were listed there.

$subj(N) \subseteq subj(L)$: If $s \in subj(N)$ then there exist roles $q, r \in R$ such that $s \in q \geq r \in N$. From the definition of N it follows that $r \in L\uparrow$. Therefore we have $s \in q \geq r \in L\uparrow$, which implies $s \in subj(L\uparrow)$. Since $L\uparrow \equiv L$, it follows that $s \in subj(L)$.

$subj(N) \cap auth_0(L_r) = \emptyset$: Suppose $s \in subj(N) \cap auth_0(L_r)$. Since $s \in auth_0(L_r)$, $(\exists r \in L_r)(s \in r)$. Since it is assumed that $s \in subj(N)$ it follows that $(\exists q \in N)(q \leq r)$. For this to be true, from the definition of N, it cannot be true that $q \in L_r\downarrow$. However, this contradicts the definition of $L_r\downarrow$ since $(r \in L_r)(r \geq q)$.

$(\forall K \in \mathcal{E})(subj(K) \subseteq subj(L) \wedge subj(K) \cap auth_0(L_r) = \emptyset \Rightarrow subj(K) \subseteq subj(N))$: Assume that the predicates to the left of the implication sign are true for some label K.

Let $s \in subj(K)$. Then, from the assumption $s \in subj(L)$. Let q be the role such that $s \in q$. Then $(\exists r' \in L)(q \geq r')$. From the definition of $L\uparrow$, since $q \geq r' \in L$, $q \in L\uparrow$. $s \notin auth(L_r)$ since $subj(K) \cap auth(L_r) = \emptyset$. Thus $(\forall p \in L_r)(q \neq p)$. From the definition of $L\downarrow$ it follows that $q \notin L_r\downarrow$. Since $q \in L\uparrow$ and $q \notin L_r\downarrow$, $q \in N = L\uparrow - L_r\downarrow$. □

`L:RevokeAccess0` L_r, with $L_r = \{r\}$ modifies L to L' such that $L' = \lfloor L\uparrow - L_r \rfloor$.

We omit the proof.

`CheckAccess` compares the role of the supplied subject label (say r_{clear}) with the list of roles in the entity label L for the particular entity E. If r_{clear} or any role below it occurs in L, access is allowed. Access is therefore allowed if and only if

$$(\exists r \in L)(r_{clear} \geq r)$$

`L:Dominates M` returns true if every role in L dominates at least one role in M. This is motivated by the following theorem.

Theorem 3 *For any two labels L and M*

$$L \geq M \iff (\forall r_L \in L)(\exists r_M \in M)(r_L \geq r_M)$$

PROOF. Assume $L \geq M$. In other words, assume $subj(L) \subseteq subj(M)$. Select any $r_L \in L$. Select any $s \in r_L$. Then $s \in subj(L)$. From the assumption it follows that $s \in subj(M)$. From the definition of $subj()$ we know $(\exists r_M \in M)(r_L \geq r_M)$. Since r_L was selected arbitrarily, this proves $L \geq M \Rightarrow (\forall r_L \in L)(\exists r_M \in M)(r_L \geq r_M)$.

Assume $(\forall r_L \in L)(\exists r_M \in M)(r_L \geq r_M)$. Let $s \in subj(L)$. Then $s \in q$ for some role $q \geq r_L \in L$. From the assumption it follows that $(\exists r_M \in M)(r_L \geq r_M)$. Therefore $s \in q \geq r_L \geq r_M \in M$ from which it follows that $s \in subj(M)$. Therefore $subj(L) \subseteq subj(M)$. This proves $(\forall r_L \in L)(\exists r_M \in M)(r_L \geq r_M) \Rightarrow L \geq M$. $\square$

Now for `IncreaseSens`. For `L:IncreaseSens M`, $\lceil L, M \rceil$ exists and has to be returned, with

$$\lceil L, M \rceil = \lfloor L\uparrow \cap M\uparrow \rfloor$$

PROOF. It will be proven that $subj(\lfloor L\uparrow \cap M\uparrow \rfloor) = subj(L) \cap subj(M)$.

Let $K = L\uparrow \cap M\uparrow$. Then $K \equiv \lfloor L\uparrow \cap M\uparrow \rfloor$

Let $s \in subj(K)$. Then $(\exists q, r \in R)(s \in q \geq r \in K)$.
$r \in K \Rightarrow r \in L\uparrow \wedge r \in M\uparrow$.
Therefore $s \in subj(L\uparrow) \wedge s \in subj(M\uparrow)$.
Since $L\uparrow \equiv L \wedge M\uparrow \equiv M, \quad s \in subj(L) \wedge s \in subj(M)$.
Therefore $s \in subj(L) \cap subj(M)$.

Let $s \in subj(L) \cap subj(M)$.
Therefore $s \in subj(L) \wedge s \in subj(M)$.
Then $s \in q$ for some role q and $(\exists r \in L)(q \geq r) \wedge (\exists r' \in M)(q \geq r')$.
From the definition of $L\uparrow$ and the fact that $q \geq r \in L$ it follows that $q \in L\uparrow$.
Similarly $q \in M\uparrow$.
Therefore $q \in L\uparrow \cap M\uparrow$, from which follows that $s \in subj(K)$.

This proves that $K \equiv \lceil L, M \rceil$. Since $K \equiv \lfloor K \rfloor$ it follows that $\lfloor L\uparrow \cap M\uparrow \rfloor \equiv \lceil K, L \rceil$. $\square$

Note that this example also demonstrates that it is possible to build a traditional MLS system: Let the roles be *unclassified*, *confidential*, *secret* and *top secret*. Then restrict information flow in the obvious way. This forms a simple, four level MLS system. A more complex example may be formed by using a more elaborate role lattice. A more efficient treatment of such a system falls outside the scope of the current paper.

5 Discussion

The primary intention of the current paper was to describe the various modules of a ConSA system and to specify the requirements for such modules. However, it is also necessary to show that it is possible to implement such a system such that it may indeed be trusted; further it is necessary to show that it is possible to implement such a system efficiently. It is also necessary to consider other component-based or 'pluggable' security solutions.

5.1 Trusting ConSA

In previous sections of this paper a number of modules have been identified that are used to protect a ConSA system. It is, however, clear that such modules need to be adequately protected themselves in order to trust the ConSA system as a whole. In section 3.4 it has, already, been argued that those methods that may modify an entity label need to be protected by other labels.

Other methods in the protecting system are either protected by labels—as described above—or by considering the origin of any message sent to the method. The latter solution applies to most of the methods: Only the message dispatcher is allowed to activate `CheckAccess` and `Dominates`; only the ACM is allowed to create new entity labels; only the SMM is allowed to `InitSubject` for subject labels and to `SetSSO` for the ACM; and so on.

In fact, only the SMM and the labels as described in section 3.4 need to be protected by other labels in general.

How is a system constructed such that a method can verify that it has been activated because a message was sent by an appropriate other method or module? The answer depends on the level of trust required. In a system where not much trust is required, this may be accomplished by examining the system stack. At a higher level of trust, the system may operate in a user and a system mode, where the code in the system mode is trusted. Most of the messages between the modules mentioned earlier are then sent in system mode. The transition between user and system mode will typically occur when the message dispatcher performs an access check; after the check has succeeded and flow control checks have been performed, a transition occurs back to user mode. If more specific checks have to be implemented (that a method a has indeed been activated by a method b), this may be done by dividing the system into segments and use the protection facilities of the hardware (and perhaps operating system, if the ConSA system does not include the operating system) to implement the specific checks. More specific checks may provide a higher level of trust, but at a higher cost. We do not discuss this issue further in the current paper.

5.2 Efficiency

Before efficiency can be considered, it is necessary to consider how the various security modules and the ConSA application may interact in practice. Such interaction depends on how the security components are 'plugged into' the application. On the one extreme the vendor of the application may acquire the security components (in source form) under licence from various security module vendors and simply compile the combined systems to produce various executable versions of the application, each version with an alternative set of security components installed. Such versions will probably be selected according to popular demand; major customers of the vendor will be able to prescribe the security components they want included in their versions.

On the other extreme, the applications and the security components may be shipped separately by the separate vendors and combined (at run time) on the client's system.

Various other possibilities between these two extremes also exist: in particular is it possible for the vendor to incorporate some security components at compile time and allow others to be added at run-time.

The impact of the above alternatives on efficiency is the level to which an optimising compiler may improve efficiency if the code is compiled together: Messages to security methods that do nothing in a particular implementation may be removed; messages to very simple security methods may be replaced by including the code of the target message in-line where the message is sent. The use of an optimising compiler as described here, obviously influences trust in the system, in which case trust and efficiency requirements may have to be prioritised.

In principle indirect addressing may be used by the application, with a table containing the entry points to routines that implement methods (and even reading and writing of instance variables). An instruction to `ProtectTable` may then daisy-chain a call to a routine that will implement the requested check. Such an approach will have little cost impact on entities that are not subject to checks, while checks may be performed efficiently where required.

5.3 A prototype

A prototype of ConSA has been implemented [5,6]. The prototype works on Linux and supports three modes of operation: Firstly, a library has been written that allows one to link individual applications to ConSA modules. While this provides a highly efficient implementation, it is not transparent, since applications have to be aware of the fact that ConSA is used. Secondly, a device driver has been written that allows a protected susbsystem to be accessed via the device driver and where the device is protected by a ConSA system. This provides a level of transparency since the applications that use the subsystem are unaware of how it is protected; they merely have to interact with it via a device driver. Finally, a ConSA system was integrated with the Linux kernel. This obviously provides the greatest level of generality because all resources are then automatically protected by the installed ConSA components. See [5,6] for details.

5.4 Related work

On mainframe computers add-on security packages that control access to resources have been popular for many years. Examples of these systems include RACF, Top Secret and CA-ACF2 [12, p.450]. These systems form a layer between the resources and the application that uses these resources. Similar to ConSA they are intended to be general-purpose security systems usable over a wide range of applications. Not only can the systems be configured via a range of parameters or menu selections, but APIs (*Application Program Interfaces*) are also provided by some so that the security system may be extended by custom programs.

ConSA differs from the existing mainframe (and similar) systems since ConSA is not a system, but an architecture. A wide variety of policies (with an associated variety in cost and/or in trust) may therefore be implemented under the same architecture and be supported by a variety of vendors. While the existing systems may be extended by custom programs, ConSA clearly specifies what the relationships between components in its three dimensions are. Even though the dimensions are orthogonal, given an access control policy, its interpretations for flow control and authorisation are exactly specified. Similarly a policy for one of the other two dimensions has a precise interpretation in the other dimensions.

In other related work the *Object Management Group* (OMG) specification [3] provides for a flexible access control strategy: access control is, in the first place provided by the *Object Request Broker* (ORB) — the “middleware layer’ that enables the components to seamlessly communicate with one another. The ORB security facilities may be extended by server objects. Since such an object is a program, a very flexible mechanism is again provided.

Note that the OMG specification implies various degrees of ‘structuredness’:

(1) ORB-based security is structured and governed by policies;
(2) Security server objects still are very general since the code for their `allow_access` methods may be arbitrary; in addition, since it is known that an `allow_access` method is used the system still has a constrained structure;
(3) Application dependent access control is very general, but has no predetermined structure.

ConSA fits between layers 2 and 3 in this schematisation.

Note that the more structured the approach is, in general, the easier it is to manipulate the object. In particular, in our terminology, structure leads to ease of interaction between policy dimensions. ConSA therefore provides similar generality to layer 2 above, but also affords the benefits associated with the increased structure than the OMG security server objects.

Note that both the mainframe and ORB approaches also address other security issues such as authentication, auditing and non-repudiation — issues that the current paper does not address in detail.

6 Conclusion

Current practise is to have different security systems in different products. Often such systems are incompatible with one another; at best they are tied to the facilities of a particular (common) operating system. If trusted versions of products are available, they are often much more expensive than the less-trusted version; trusted versions also appear in the marketplace long after their less-trusted counterparts have appeared.

This paper presented an approach to separate the security aspects of a system from the data processing aspects of the system. It has been argued that the approach is general enough to appeal to a wide audience. The paper illustrated how the approach may be used to build a role-based security system. It is obvious that the approach is suitable for both traditional multilevel and discretionary access control systems (with support for both capability and access control list approaches in the latter case). The paper illustrated the implementation of a role-based model. We have also used the approach to specify a Chinese Wall security model elsewhere [9].

If an approach such as ConSA is accepted and standardised by industry it will become possible for a vendor to design a system independent from the security components to be used eventually. From the client's side it may potentially make a variety of security components available for any given application. In particular may it become possible for a user to decide on a particular security policy and a set of components that directly supports the policy. The same security components may then be used for all applications. The potential benefits of such an approach are obviously huge.

Acknowledgements

The author would like to thank Alexandre Hardy for his scrutiny of the manuscript which led to the revision of a number of concepts. Any errors that do remain are, however, the sole responsibility of the author.

References

[1] D.E. Bell and L.J. LaPadula, Secure computer system: unified exposition and Multics interpretation, *Rep. ESD-TR-75-306*, MITRE, 1976.

[2] T. Budd, *An Introduction to Object-Oriented Programming* (Addison-Wesley, Reading, 1991).

[3] CORBAservices: Common Object Services Specification, Object Management Group, 1998.

[4] A. Goldberg and D. Robson, *Smalltalk 80: The Language and its Implementation* (Addison-Wesley, Menlo Park, 1983).

[5] A. Hardy, An Implementation and Analysis of the Configurable Security Architecture, M.Sc. Dissertation, Rand Afrikaans University, Johannesburg, South Africa, 1999.

[6] A. Hardy and M.S. Olivier, A Configurable Security Architecture Prototype, in: *Proceedings of the 14th IFIP 11.3 Working Conference on Database Security* (Schoorl, The Netherlands, 2000) 69–84.

[7] P.W. Jansen van Rensburg and M.S. Olivier, A Discretionary Security Model for Object-oriented Environments, in: S.K. Katsikas and D. Gritzalis, eds., *Information Systems Security: Facing the Information Society of the 21st Century* (Chapman & Hall, London, 1996) 306–316.

[8] T.F. Keefe, W.T. Tsai and M.B. Thuraisingham, SODA: A Secure Object-oriented Database System, *Computers & Security*, **8** (1989) 517–533.

[9] F.A. Lategan and M.S. Olivier, An implementation of the Chinese Wall security Model using ConSA, Internal Report, Department of Computer Science, Rand Afrikaans University, Johannesburg, South Africa, 1998.

[10] M.S. Olivier and S.H. von Solms, Building a Secure Database Using Self-protecting Objects, *Computers & Security*, **11**, 3 (1992) 259–271.

[11] M.S. Olivier and S.H. von Solms, A Taxonomy for Secure Object-oriented Databases, *ACM Transactions on Database Systems*, **19**, 1 (1994) 3–46.

[12] C.P. Pfleeger, *Security in Computing* (Prentice-Hall, Englewood Cliffs, 1989).

[13] F. Rabitti, E. Bertino, W. Kim and D. Woelk, A Model of Authorization for Next-Generation Database Systems, *ACM Transactions on Database Systems*, **16**, 1 (1991) 88–131.