To appear in *Foundations of Security Analysis and Design*,
R. Focardi and R. Gorrieri (eds), LNCS 2172, Springer-Verlag.

Access Control: Policies, Models, and Mechanisms

Pierangela Samarati[1] and Sabrina De Capitani di Vimercati[2]

[1] Dipartimento di Tecnologie dell'Informazione – Università di Milano
Via Bramante 65 – 26013 - Crema (CR) Italy
samarati@dsi.unimi.it
http//homes.dsi.unimi.it/∼samarati

[2] Dip. di Elettronica per l'Automazione – Università di Brescia
Via Branze 38 – 25123 Brescia - Italy
decapita@ing.unibs.it
http//www.ing.unibs.it/∼decapita

Abstract. Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied. The access control decision is enforced by a mechanism implementing regulations established by a security policy. Different access control policies can be applied, corresponding to different criteria for defining what should, and what should not, be allowed, and, in some sense, to different definitions of what ensuring security means. In this chapter we investigate the basic concepts behind access control design and enforcement, and point out different security requirements that may need to be taken into consideration. We discuss several access control policies, and models formalizing them, that have been proposed in the literature or that are currently under investigation.

1 Introduction

An important requirement of any information management system is to *protect data and resources* against unauthorized disclosure (*secrecy*) and unauthorized or improper modifications (*integrity*), while at the same time ensuring their availability to legitimate users (*no denials-of-service*). Enforcing protection therefore requires that *every access to a system and its resources be controlled and that all and only authorized accesses can take place.* This process goes under the name of *access control*. The development of an access control system requires the definition of the regulations according to which access is to be controlled and their implementation as functions executable by a computer system. The development process is usually carried out with a multi-phase approach based on the following concepts:

Security policy: it defines the (high-level) rules according to which access control must be regulated.[1]

[1] Often, the term policy is also used to refer to particular instances of a policy, that is, actual authorizations and access restrictions to be enforced (e.g., Employees can read bulletin-board).

Security model: it provides a *formal* representation of the access control security policy and its working. The formalization allows the proof of properties on the security provided by the access control system being designed.

Security mechanism: it defines the low level (software and hardware) functions that implement the controls imposed by the policy and formally stated in the model.

The three concepts above correspond to a conceptual separation between different levels of abstraction of the design, and provides the traditional advantages of multi-phase software development. In particular, the separation between policies and mechanisms introduces an independence between protection requirements to be enforced on the one side, and mechanisms enforcing them on the other. It is then possible to: *i)* discuss protection requirements independently of their implementation, *ii)* compare different access control policies as well as different mechanisms that enforce the same policy, and *iii)* design mechanisms able to enforce multiple policies. This latter aspect is particularly important: if a mechanism is tied to a specific policy, a change in the policy would require changing the whole access control system; mechanisms able to enforce multiple policies avoid this drawback. The formalization phase between the policy definition and its implementation as a mechanism allows the definition of a formal model representing the policy and its working, making it possible to define and prove security properties that systems enforcing the model will enjoy [54]. Therefore, by proving that the model is "secure" and that the mechanism *correctly implements* the model, we can argue that the system is "secure" (w.r.t. the definition of security considered). The implementation of a correct mechanism is far from being trivial and is complicated by the need to cope with possible security weaknesses due to the implementation itself and by the difficulty of mapping the access control primitives to a computer system. The access control mechanism must work as a *reference monitor*, that is, a trusted component intercepting each and every request to the system [5]. It must also enjoy the following properties:

- *tamper-proof*: it should not be possible to alter it (or at least it should not be possible for alterations to go undetected);
- *non-bypassable*: it must mediate all accesses to the system and its resources;
- *security kernel*: it must be confined in a limited part of the system (scattering security functions all over the system implies that all the code must be verified);
- *small*: it must be of limited size to be susceptible of rigorous verification methods.

Even the definition of access control policies (and their corresponding models) is far from being a trivial process. One of the major difficulty lies in the interpretation of, often complex and sometimes ambiguous, real world security policies and in their translation in well defined and unambiguous rules enforceable by a computer system. Many real world situations have complex policies, where access decisions depend on the application of different rules coming, for example, from laws, practices, and organizational regulations. A security policy must capture all the different regulations to be enforced and, in addition, must also consider possible additional threats due to the use of a computer system. Access control policies can be grouped into three main classes:

Discretionary (DAC) (authorization-based) policies control access based on the identity of the requestor and on access rules stating what requestors are (or are not) allowed to do.

Mandatory (MAC) policies control access based on mandated regulations determined by a central authority.

Role-based (RBAC) policies control access depending on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles.

Discretionary and role-based policies are usually coupled with (or include) an *administrative* policy that defines who can specify authorizations/rules governing access control.

In this chapter we illustrate different access control policies and models that have been proposed in the literature, also investigating their low level implementation in terms of security mechanisms. In illustrating the literature and the current status of access control systems, of course, the chapter does not pretend to be exhaustive. However, by discussing different approaches with their advantages and limitations, this chapter hopes to give an idea of the different issues to be tackled in the development of an access control system, and of good security principles that should be taken into account in the design.

The chapter is structured as follows. Section 2 introduces the basic concepts of discretionary policies and authorization-based models. Section 3 shows the limitation of authorization-based controls to introduce the basis for the need of mandatory policies, which are then discussed in Section 4. Section 5 illustrates approaches combining mandatory and discretionary principles to the goal of achieving mandatory information flow protection without loosing the flexibility of discretionary authorizations. Section 6 illustrates several discretionary policies and models that have been proposed. Section 7 illustrates role-based access control policies. Finally, Section 8 discusses advanced approaches and directions in the specification and enforcement of access control regulations.

2 Basic concepts of discretionary policies

Discretionary policies enforce access control on the basis of the identity of the requestors and explicit access rules that establish who can, or cannot, execute which actions on which resources. They are called discretionary as users can be given the ability of passing on their privileges to other users, where granting and revocation of privileges is regulated by an administrative policy. Different discretionary access control policies and models have been proposed in the literature. We start in this section with the early discretionary models, to convey the basic ideas of authorization specifications and their enforcement. We will come back to discretionary policies after having dealt with mandatory controls. We base the discussion of the "primitive" discretionary policies on the access matrix model.

2.1 The access matrix model

The access matrix model provides a framework for describing discretionary access control. First proposed by Lampson [53] for the protection of resources within the context of operating systems, and later refined by Graham and Denning [41], the model was subsequently formalized by Harrison, Ruzzo, and Ullmann (HRU model) [44], who developed the access control model proposed by Lampson to the goal of analyzing the complexity of determining an access control policy. The original model is called access matrix since the authorization state, meaning the authorizations holding at a given time in the system, is represented as a matrix. The matrix therefore gives an abstract representation of protection systems. Although the model may seem primitive, as richer policies and languages have been investigated subsequently (see Section 6), its treatment is useful to illustrate some aspects to be taken into account in the formalization of an access control system.

A first step in the development of an access control system is the identification of the *objects* to be protected, the *subjects* that execute activities and request access to objects, and the *actions* that can be executed on the objects, and that must be controlled. Subjects, objects, and actions may be different in different systems or application contexts. For instance, in the protection of operating systems, objects are typically files, directories, or programs; in database systems, objects can be relations, views, stored procedures, and so on. It is interesting to note that subjects can be themselves objects (this is the case, for example, of executable code and stored procedures). A subject can create additional subjects (e.g., children processes) in order to accomplish its task. The creator subject acquires control privileges on the created processes (e.g., to be able to suspend or terminate its children).

In the access matrix model, the state of the system is defined by a triple (S, O, A), where S is the set of subjects, who can exercise privileges; O is the set of objects, on which privileges can be exercised (subjects may be considered as objects, in which case $S \subseteq O$); and A is the access matrix, where rows correspond to subjects, columns correspond to objects, and entry $A[s, o]$ reports the privileges of s on o. The type of the objects and the actions executable on them depend on the system. By simply providing a framework where authorizations can be specified, the model can accommodate different privileges. For instance, in addition to the traditional read, write, and execute actions, *ownership* (i.e., property of objects by subjects), and *control* (to model father-children relationships between processes) can be considered. Figure 1 illustrates an example of access matrix.

Changes to the state of a system is carried out through *commands* that can execute *primitive* operations on the authorization state, possibly depending on some conditions. The HRU formalization identified six primitive operations that

	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Fig. 1. An example of access matrix

describe changes to the state of a system. These operations, whose effect on the authorization state is illustrated in Figure 2, correspond to adding and removing a subject, adding and removing an object, and adding and removing a privilege. Each command has a conditional part and a body and has the form

command $c(x_1, \ldots, x_k)$
    if r_1 in $A[x_{s_1}, x_{o_1}]$ and
      r_2 in $A[x_{s_2}, x_{o_2}]$ and
      .
      .
      r_m in $A[x_{s_m}, x_{o_m}]$
    then op_1
        op_2
        .
        .
        op_n
end.

with $n > 0, m \geq 0$. Here $r_1, ..., r_m$ are actions, $op_1, ..., op_n$ are primitive operations, while $s_1, ..., s_m$ and $o_1, ..., o_m$ are integers between 1 and k. If m=0, the command has no conditional part.

For example, the following command creates a file and gives the creating subject ownership privilege on it.

command CREATE(creator,file)
    create object file
    enter Own *into* A[creator,file] **end.**

The following commands allow an owner to grant to others, and revoke from others, a privilege to execute an action on her files.

command CONFER$_a$(owner,friend,file)
    if Own in A[owner,file]
      then *enter* a *into* A[friend,file] **end.**

OPERATION (op)	CONDITIONS	NEW STATE ($Q \vdash_{op} Q'$)
enter r into $A[s,o]$	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $A'[s,o] = A[s,o] \cup \{r\}$ $A'[s_i,o_j] = A[s_i,o_j] \quad \forall (s_i,o_j) \neq (s,o)$
delete r from $A[s,o]$	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $A'[s,o] = A[s,o] \setminus \{r\}$ $A'[s_i,o_j] = A[s_i,o_j] \quad \forall (s_i,o_j) \neq (s,o)$
create subject s'	$s' \notin S$	$S' = S \cup \{s'\}$ $O' = O \cup \{s'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S, o \in O$ $A'[s',o] = \emptyset \quad \forall o \in O'$ $A'[s,s'] = \emptyset \quad \forall s \in S'$
create object o'	$o' \notin O$	$S' = S$ $O' = O \cup \{o'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S, o \in O$ $A'[s,o'] = \emptyset \quad \forall s \in S'$
destroy subject s'	$s' \in S$	$S' = S \setminus \{s'\}$ $O' = O \setminus \{s'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S', o \in O'$
destroy object o'	$o' \in O$ $o' \notin S$	$S' = S$ $O' = O \setminus \{o'\}$ $A'[s,o] = A[s,o] \quad \forall s \in S', o \in O'$

Fig. 2. Primitive operations of the HRU model

command REVOKE$_a$(owner,ex-friend,file)
  if Own in A[owner,file]
   then *delete a from* A[ex-friend,file] **end.**

Note that here a is not a parameter, but an abbreviation for defining many similar commands, one for each value that a can take (e.g., CONFER$_{\text{read}}$, REVOKE$_{\text{write}}$). Since commands are not parametric w.r.t. actions, a different command needs to be specified for each action that can be granted/revoked.

Let $Q \vdash_{op} Q'$ denote the execution of operation op on state Q, resulting in state Q'. The execution of command $c(a_1, ..., a_k)$ on a system state $Q = (S, O, A)$ causes the *transition* from state Q to state Q' such that $\exists\ Q_1, \ldots, Q_n$ for which $Q \vdash_{op_1^*} Q_1 \vdash_{op_2^*} ... \vdash_{op_n^*} Q_n = Q'$, where $op_1^* \ldots op_n^*$ are the primitive operations $op_1 \ldots op_n$ in the body (operational part) of command c, in which actual parameters a_i are substituted for each formal parameters x_i, $i := 1, \ldots, k$. If the conditional part of the command is not verified, then the command has no effect and $Q = Q'$.

Although the HRU model does not include any buil-in administrative policies, the possibility of defining commands allows their formulation. Administrative authorizations can be specified by attaching flags to access privileges.

For instance, a *copy flag*, denoted $*$, attached to a privilege may indicate that the privilege can be transferred to others. Granting of authorizations can then be accomplished by the execution of commands like the one below (again here TRANSFER_a is an abbreviation for as many commands as there are actions).

command TRANSFER_a(subj,friend,file)
  if a^* in A[subj,file]
    then *enter a into* A[friend,file] **end.**

The ability of specifying commands of this type clearly provides flexibility as different administrative policies can be taken into account by defining appropriate commands. For instance, an alternative administrative flag (called *transfer only* and denoted $+$) can be supported, which gives the subject the ability of passing on the privilege to others but for which, so doing, the subject looses the privilege. Such a flexibility introduces an interesting problem referred to as *safety*, and concerned with the propagation of privileges to subjects in the system. Intuitively, given a system with initial configuration Q, the *safety* problem is concerned with determining whether or not a given subject s can ever acquire a given access a on an object o, that is, if there exists a sequence of requests that executed on Q can produce a state Q' where a appears in a cell $A[s,o]$ that did not have it in Q. (Note that, of course, not all leakages of privileges are bad and subjects may intentionally transfer their privileges to "trusworthy" subjects. Trustworthy subjects are therefore ignored in the analysis.) It turns out that the safety problem is undecidable in general (it can be reduced to the halting problem of a Turing machine) [4]. It remains instead decidable for cases where subjects and objects are finite, and in *mono-operational* systems, that is, systems where the body of commands can have at most one operation (while the conditional part can still be arbitrarily complex). However, as noted in [81], mono-operational systems have the limitation of making create operations pretty useless: a single create command cannot do more than adding an empty row/column (it cannot write anything in it). It is therefore not possible to support ownership or control relationships between subjects. Progresses in safety analysis were made in a later extension of the HRU model by Sandhu [81], who proposed the *TAM* (Typed Access Matrix) model. TAM extends HRU with strong typing: each subject and object has a type; the type is associated with the subjects/objects when they are created and thereafter does not change. Safety results decidable in polynomial time for cases where the system is monotonic (privileges cannot be deleted), commands are limited to three parameters, and there are no cyclic creates. Safety remains undecidable otherwise.

2.2 Implementation of the access matrix

Although the matrix represents a good conceptualization of authorizations, it is not appropriate for implementation. In a general system, the access matrix will be usually enormous in size and sparse (most of its cells are likely to be empty). Storing the matrix as a two-dimensional array is therefore a waste of memory space. There are three approaches to implementing the access matrix in a practical way:

Authorization Table Non empty entries of the matrix are reported in a table with three columns, corresponding to subjects, actions, and objects, respectively. Each tuple in the table corresponds to an authorization. The authorization table approach is generally used in DBMS systems, where authorizations are stored as catalogs (relational tables) of the database.

Access Control List (ACL) The matrix is stored by column. Each object is associated with a list indicating, for each subject, the actions that the subject can exercise on the object.

Capability The matrix is stored by row. Each user has associated a list, called capability list, indicating, for each object, the accesses that the user is allowed to exercise on the object.

Figure 3 illustrates the authorization table, ACLs, and capabilities, respectively, corresponding to the access matrix in Figure 1.

Capabilities and ACLs present advantages and disadvantages with respect to authorization control and management. In particular, with ACLs it is immediate to check the authorizations holding on an object, while retrieving all the authorizations of a subject requires the examination of the ACLs for all the objects. Analogously, with capabilities, it is immediate to determine the privileges of a subject, while retrieving all the accesses executable on an object requires the examination of all the different capabilities. These aspects affect the efficiency of authorization revocation upon deletion of either subjects or objects.

In a system supporting capabilities, it is sufficient for a subject to present the appropriate capability to gain access to an object. This represents an advantage in distributed systems since it permits to avoid repeated authentication of a subject: a user can be authenticated at a host, acquire the appropriate capabilities and present them to obtain accesses at the various servers of the system. However, capabilities are vulnerable to *forgery* (they can be copied and reused by an unauthorized third party). Another problem in the use of capability is the enforcement of revocation, meaning invalidation of capabilities that have been released.

A number of capability-based computer systems were developed in the 1970s, but did not prove to be commercially successful. Modern operating systems typically take the ACL-based approach. Some systems implement an abbreviated form of ACL by restricting the assignment of authorizations to a limited number (usually one or two) of named groups of users, while individual authorizations are not allowed. The advantage of this is that ACLs can be efficiently represented as small bit-vectors. For instance, in the popular Unix operating system, each user in the system belongs to exactly one group and each file has an owner (generally the user who created it), and is associated with a group (usually the group of its owner). Authorizations for each file can be specified for the file's owner, for the group to which the file belongs, and for "the rest of the world" (meaning all the remaining users). No explicit reference to users or groups is allowed.

User	Access mode	Object
Ann	own	File 1
Ann	read	File 1
Ann	write	File 1
Ann	read	File 2
Ann	write	File 2
Ann	execute	Program 1
Bob	read	File 1
Bob	read	File 3
Bob	write	File 3
Carl	read	File 2
Carl	execute	Program 1
Carl	read	Program 1

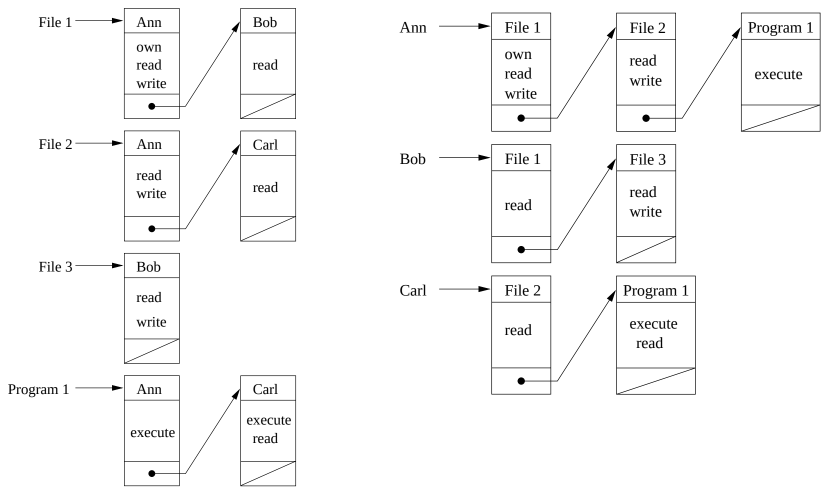

Fig. 3. Authorization table, ACLs, and capabilities for the matrix in Figure 1

Authorizations are represented by associating with each object an access control list of 9 bits: bits 1 through 3 reflect the privileges of the file's owner, bits 4 through 6 those of the user group to which the file belongs, and bits 7 through 9 those of all the other users. The three bits correspond to the read (`r`), write (`w`), and execute (`x`) privilege, respectively. For instance, ACL `rwxr-x--x` associated with a file indicates that the file can be read, written, and executed by its owner, read and executed by users belonging to the group associated with the file, and executed by all the other users.

3 Vulnerabilities of the discretionary policies

In defining the basic concepts of discretionary policies, we have referred to access requests on objects submitted by users, which are then checked againsts the users' authorizations. Although it is true that each request is originated because of some user's actions, a more precise examination of the access control problem shows the utility of separating *users* from *subjects*. Users are passive entities for whom authorizations can be specified and who can connect to the system. Once connected to the system, users originate processes (subjects) that execute on their behalf and, accordingly, submit requests to the system. Discretionary policies ignore this distinction and evaluate all requests submitted by a process running on behalf of some user against the authorizations of the user. This aspect makes discretionary policies vulnerable from processes executing malicious programs exploiting the authorizations of the user on behalf of whom they are executing. In particular, the access control system can be bypassed by Trojan Horses embedded in programs. A *Trojan Horse* is a computer program with an apparently or actually useful function, which contains additional *hidden* functions that surreptitiously exploit the legitimate authorizations of the invoking process. (Viruses and logic bombs are usually transmitted as Trojan Horses.) A Trojan Horse can improperly use any authorizations of the invoking user, for example, it could even delete all files of the user (this destructive behavior is not uncommon in the case of viruses). This vulnerability to Trojan Horses, together with the fact that *discretionary policies do not enforce any control on the flow of information once this information is acquired by a process,* makes it possible for processes to leak information to users not allowed to read it. All this can happen without the cognizance of the data administrator/owner, and despite the fact that each single access request is controlled against the authorizations. To understand how a Trojan Horse can leak information to unauthorized users despite the discretionary access control, consider the following example. Assume that within an organization, Vicky, a top-level manager, creates a file Market containing important information about releases of new products. This information is very sensitive for the organization and, according to the organization's policy, should not be disclosed to anybody besides Vicky. Consider now John, one of Vicky's subordinates, who wants to acquire this sensitive information to sell it to a competitor organization. To achieve this, John creates a file, let's call it Stolen, and gives Vicky the authorization to write the file. Note that Vicky may not even know about the existence of Stolen, or about the fact that she has the write authorization on it. Moreover, John modifies an application generally used by Vicky, to include two hidden operations, a read operation on file Market and a write operation on file Stolen (Figure 4(a)). Then, he gives the new application to his manager. Suppose now that Vicky executes the application. Since the application executes on behalf of Vicky, every access is checked against Vicky's authorizations, and the read and write operations above are allowed. As a result, during execution, sensitive information in Market is transferred to Stolen and thus made readable to the dishonest employee John, who can then sell it to the competitor (Figure 4(b)).

The reader may object that there is little point in defending against Trojan Horses leaking information flow: such an information flow could have happened anyway, by having Vicky explicitly tell this information to John, possibly even

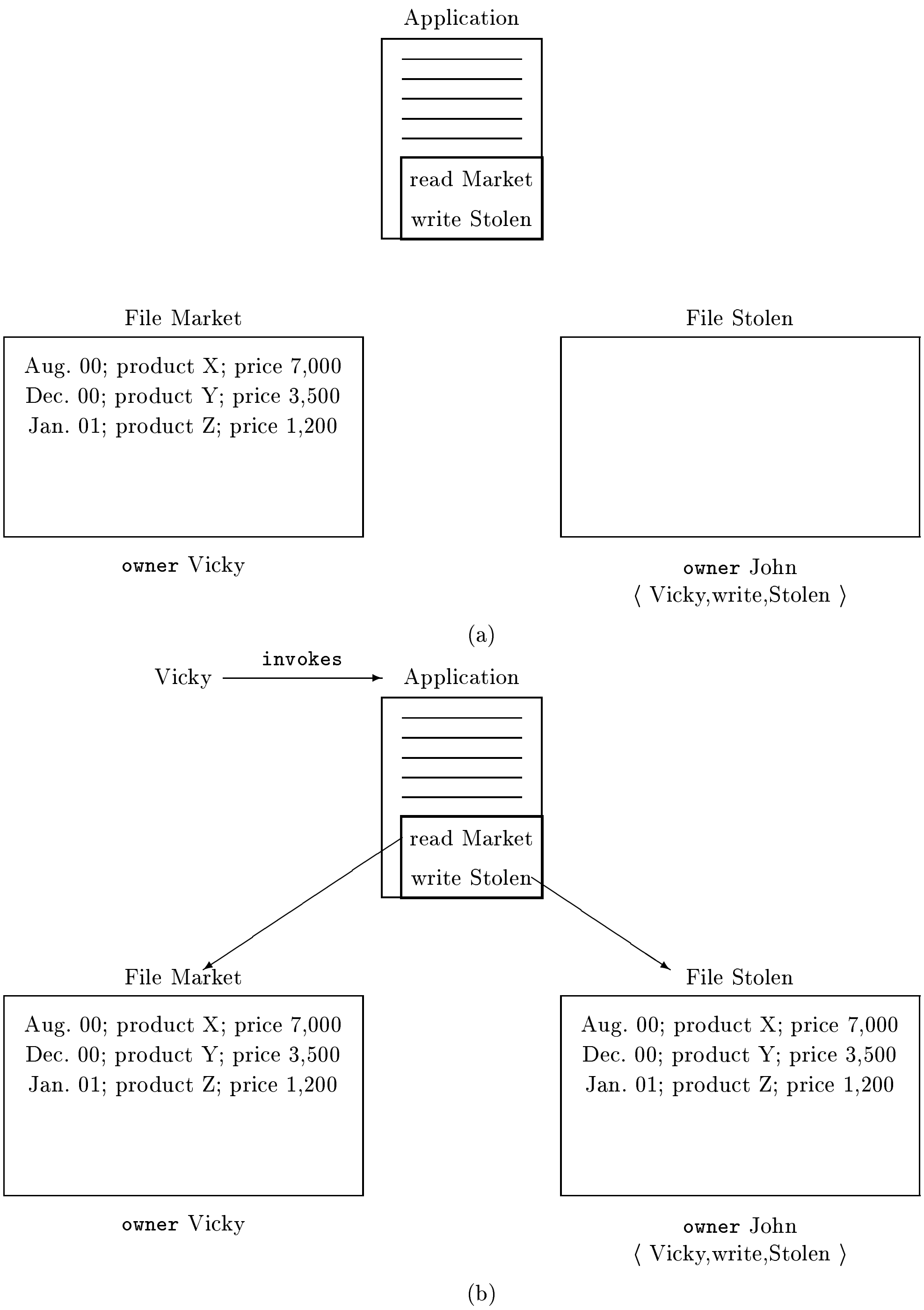

Fig. 4. An example of Trojan Horse improperly leaking information

off-line, without the use of the computer system. Here is where the distinction between users and subjects operating on their behalf comes in. *While users are trusted to obey the access restrictions, subjects operating on their behalf are not.* With reference to our example, Vicky is trusted not to release the sensitive information she knows to John, since, according to the authorizations, John cannot read it. However, the processes operating on behalf of Vicky cannot be given the same trust. Processes run programs which, unless properly certified, cannot be trusted for the operations they execute. For this reason, restrictions should be enforced on the operations that processes themselves can execute. In particular, protection against Trojan Horses leaking information to unauthorized users requires controlling the flows of information within processes execution and possibly restricting them. Mandatory policies provide a way to enforce information flow control through the use of labels.

4 Mandatory policies

Mandatory security policies enforce access control on the basis of regulations mandated by a central authority. The most common form of mandatory policy is the *multilevel security policy*, based on the classifications of *subjects* and *objects* in the system. Objects are passive entities storing information. Subjects are active entities that request access to the objects. Note that there is a distinction between *subjects* of the mandatory policy and the *authorization subjects* considered in the discretionary policies. While authorization subjects typically correspond to users (or groups thereof), mandatory policies make a distinction between *users* and *subjects*. Users are human beings who can access the system, while subjects are processes (i.e., programs in execution) operating on behalf of users. This distinction allows the policy to control the indirect accesses (leakages or modifications) caused by the execution of processes.

4.1 Security classifications

In multilevel mandatory policies, an access class is assigned to each object and subject. The access class is one element of a partially ordered set of classes. The partial order is defined by a *dominance* relationship, which we denote with $\geq$. While in the most general case, the set of access classes can simply be any set of labels that together with the dominance relationship defined on them form a POSET (partially ordered set), most commonly an access class is defined as consisting of two components: a *security level* and a *set of categories*. The security level is an element of a hierarchically ordered set, such as Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U), where $\mathtt{TS} > \mathtt{S} > \mathtt{C} > \mathtt{U}$. The set of categories is a subset of an unordered set, whose elements reflect functional, or competence, areas (e.g., NATO, Nuclear, and Army, for military systems; Financial, Administration, and Research, for commercial systems). The dominance relationship $\geq$ is then defined as follows: an access class c_1 *dominates* ($\geq$) an access class c_2 iff the security level of c_1 is greater than or equal to that of

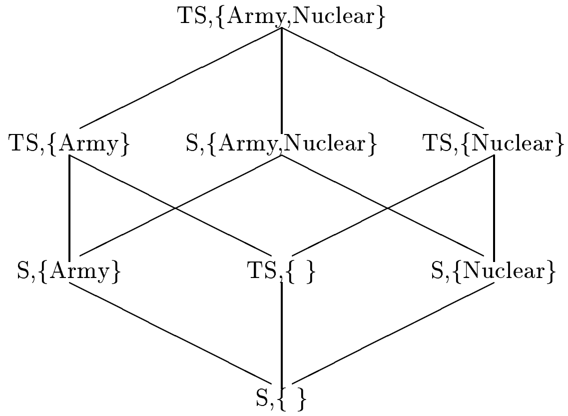

Fig. 5. An example of security lattice

c_2 and the categories of c_1 include those of c_2. Formally, given a totally ordered set of security levels $\mathcal{L}$, and a set of categories $\mathcal{C}$, the set of access classes is $\mathcal{AC} = \mathcal{L} \times \wp(\mathcal{C})^2$, and $\forall c_1 = (L_1, C_1), c_2 = (L_2, C_2) : c_1 \geq c_2 \Longleftrightarrow L_1 \geq L_2 \wedge C_1 \supseteq C_2$. Two classes c_1 and c_2 such that neither $c_1 \geq c_2$ nor $c_2 \geq c_1$ holds are said to be *incomparable*.

It is easy to see that the dominance relationship so defined on a set of access classes $\mathcal{AC}$ satisfies the following properties.

- *Reflexivity:* $\forall x \in \mathcal{AC} : x \geq x$
- *Transitivity:* $\forall x, y, z \in \mathcal{AC} : x \geq y, y \geq z \Longrightarrow x \geq z$
- *Antisymmetry:* $\forall x, y \in \mathcal{AC} : x \geq y, y \geq x \Longrightarrow x = y$
- *Existence of a least upper bound:* $\forall x, y \in \mathcal{AC} : \exists\, !z \in \mathcal{AC}$
  - $z \geq x$ and $z \geq y$
  - $\forall t \in \mathcal{AC} : t \geq x$ and $t \geq y \Longrightarrow t \geq z$.
- *Existence of a greatest lower bound:* $\forall x, y \in \mathcal{AC} : \exists\, !z \in \mathcal{AC}$
  - $x \geq z$ and $y \geq z$
  - $\forall t \in \mathcal{AC} : x \geq t$ and $y \geq t \Longrightarrow z \geq t$.

Access classes defined as above together with the dominance relationship between them therefore form a lattice [31]. Figure 5 illustrates the security lattice obtained considering security levels `TS` and `S`, with `TS>S` and the set of categories {`Nuclear`,`Army`}.

The semantics and use of the classifications assigned to objects and subjects within the application of a multilevel mandatory policy is different depending on whether the classification is intended for a *secrecy* or an *integrity* policy. We next examine secrecy-based and integrity-based mandatory policies.

[2] $\wp(\mathcal{C})$ denotes the powerset of $\mathcal{C}$.

4.2 Secrecy-based mandatory policies

A secrecy mandatory policy controls the direct and *indirect* flows of information to the purpose of preventing leakages to unauthorized subjects. Here, the semantics of the classification is as follows. The security level of the access class associated with an object reflects the sensitivity of the information contained in the object, that is, the potential damage that could result from the unauthorized disclosure of the information. The security level of the access class associated with a user, also called *clearance*, reflects the user's trustworthiness not to disclose sensitive information to users not cleared to see it. Categories define the area of competence of users and data and are used to provide finer grained security classifications of subjects and objects than classifications provided by security levels alone. They are the basis for enforcing *need-to-know* restrictions (i.e., confining subjects to access information they actually need to know to perform their job).

Users can connect to the system at any access class dominated by their clearance. A user connecting to the system at a given access class originates a subject at that access class. For instance, with reference to the lattice in Figure 5, a user cleared ($\texttt{TS},\{\texttt{Nuclear}\}$) can connect to the system as a ($\texttt{S},\{\texttt{Nuclear}\}$), ($\texttt{TS},\emptyset$), or ($\texttt{TS},\emptyset$) subject. Requests by a subject to access an object are controlled with respect to the access class of the subject and the object and granted only if some relationship, depending on the requested access, is satisfied. In particular, two principles, first formulated by Bell and LaPadula [12], must be satisfied to protect information confidentiality:

No-read-up A subject is allowed a read access to an object only if the access class of the subject dominates the access class of the object.

No-write-down A subject is allowed a write access to an object only if the access class of the subject is dominated by the access class of the object.

Satisfaction of these two principles prevents information to flow from high level subjects/objects to subjects/objects at lower (or incomparable) levels, thereby ensuring the satisfaction of the protection requirements (i.e., no process will be able to make sensitive information available to users not cleared for it). This is illustrated in Figure 6, where four accesses classes composed only of a security level (`TS`, `S`, `C`, and `U`) are taken as example. Note the importance of controlling both read and write operations, since both can be improperly used to leak information. Consider the example on the Trojan Horse illustrated in Section 3. Possible classifications reflecting the access restrictions to be enforced could be: Secret for Vicky and Market, and Unclassified for John and Stolen. In the respect of the no-read-up and no-write-down principles, the Trojan Horse will never be able to complete successfully. If Vicky connects to the system as a Secret (or Confidential) subject, and thus the application runs with a Secret (or Confidential) access class, the write operation will be blocked. If Vicky invokes the application as an Unclassified subject, the read operation will be blocked instead.

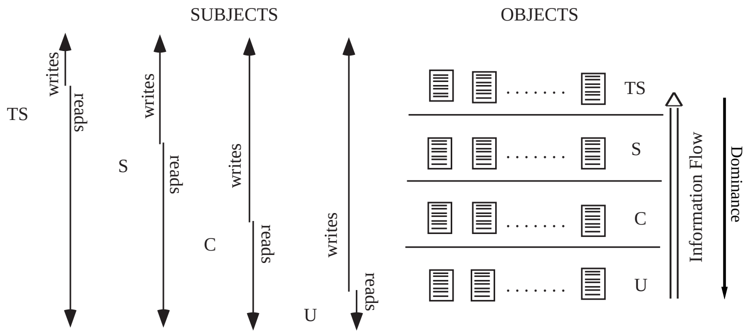

Fig. 6. Information flow for secrecy

Given the no-write-down principle, it is clear now why users are allowed to connect to the system at different access classes, so that they are able to access information at different levels (provided that they are cleared for it). For instance, Vicky has to connect to the system at a level below her clearance if she wants to write some Unclassified information, such as working instructions for John. Note that a lower class does not mean "less" privileges in absolute terms, but only less reading privileges (see Figure 6).

Although users can connect to the system at any level below their clearance, the strict application of the no-read-up and the no-write-down principles may result too rigid. Real world situations often require exceptions to the mandatory restrictions. For instance, data may need to be downgraded (e.g., data subject to embargoes that can be released after some time). Also, information released by a process may be less sensitive than the information the process has read. For instance, a procedure may access personal information regarding the employees of an organization and return the benefits to be granted to each employee. While the personal information can be considered Secret, the benefits can be considered Confidential. To respond to situations like these, multilevel systems should then allow for exceptions, loosening or waiving restrictions, in a controlled way, to processes that are *trusted* and ensure that information is *sanitized* (meaning the sensitivity of the original information is lost).

Note also that DAC and MAC policies are not mutually exclusive, but can be applied jointly. In this case, an access to be granted needs both *i)* the existence of the necessary authorization for it, and *ii)* to satisfy the mandatory policy. Intuitively, the discretionary policy operates *within the boundaries* of the mandatory policy: it can only restrict the set of accesses that would be allowed by MAC alone.

4.3 The Bell-LaPadula model (some history)

The secrecy based control principles just illustrated summarize the basic axioms of the security model proposed by David Bell and Leonard LaPadula [12]. Here, we illustrate some concepts of the model formalization to give an idea of the different aspects to be taken into account in the definition of a security model. This little bit of history is useful to understand the complications of formalizing a policy and making sure that the policy' axioms actually ensure protection as intended. We note first that different versions of the model have been proposed (due to the formalization of new properties [10, 12, 55], or related to specific application environments [11]), however the basic principles remain the same (and are those illustrated in the previous section). Also, here we will be looking only at the aspects of the formalization needed to illustrate the concepts we want to convey: for the sake of simplicity, the formulation of the model is simplified and some aspects are omitted.

In the Bell and LaPadula model a system is composed of a set of subjects S, objects O, and actions A, which includes `read` and `write`[3]. The model also assumes a lattice L of access classes and a function $\lambda : S \cup O \rightarrow L$ that, when applied to a subject (object, resp.) in a given state, returns the classification of the subject (object, resp.) in that state. A state $v \in V$ is defined as a triple (b, M, λ), where $b \in \wp(S \times O \times A)$ is the set of current accesses (s, o, a), M is the access matrix expressing discretionary permissions (as in the HRU model), and λ is the association of access classes with subjects and objects. A system consists of an initial state v_0, a set of requests R, and a state transition function $T : V \times R \rightarrow V$ that transforms a system state into another state resulting from the execution of a request. Intuitively, requests capture acquisition and release of accesses, granting and revocation of authorizations, as well as changes of levels. The model then defines a set of axioms stating properties that the system must satisfy and that express the constraints imposed by the mandatory policy. The first version of the Bell and LaPadula model stated the following criteria.

simple property A state v satisfies the simple security property iff for every $s \in S$, $o \in O$: $(s, o, \texttt{read}) \in b \Longrightarrow \lambda(s) \geq \lambda(o)$.

***-property** A state v satisfies the *-security property iff for every $s \in S$, $o \in O$: $(s, o, \texttt{write}) \in b \Longrightarrow \lambda(o) \geq \lambda(s)$.

The two axioms above correspond to the no-read-up and no-write-down principles we have illustrated in Section 4.2. A state is then defined to be secure if it satisfies both the simple security property and the *-property. A system (v_0, R, T) is secure if and only if every state reachable from v_0 by executing one or more finite sequences of requests from R is *state secure*.

In the first formulation of their model, Bell and LaPadula provide a *Basic Security Theorem (BST)*, which states that a system is secure if *i)* its initial

[3] For uniformity of the discussion, we use the term "write" here to denote the "write-only" (or "append") action.

state v_0 is secure, and *ii)* the state transition T is security preserving, that is, it transforms a secure state into another secure state.

As noticed by McLean in his example called "System Z" [63], the BST theorem does not actually guarantee security. The problem lies in the fact that no restriction, but to be preserving of state security, is put on transitions. In his System Z example, McLean shows how failing to control transitions can compromise security. Consider a system Z whose initial state is secure and that has only one type of transition: when a subject requests any type of access to an object o, every subject and object in the system are downgraded to the lowest possible access class and the access is granted. System Z satisfies the Bell and LaPadula notion of security, but it is obviously not secure in any meaningful sense. The problem pointed out by System Z is that transitions need to be controlled. Accordingly, McLean proposes extending the model with a new function $C : S \cup O \rightarrow \wp(S)$, which returns the set of subjects allowed to change the level of its argument. A transition is secure if it allows changes to the level of a subject/object x only by subjects in $C(x)$; intuitively, these are subjects trusted for downgrading. A system (v_0, R, T) is *secure* if and only if *i)* v_0 is secure, *ii)* every state reachable from v_0 by executing a finite sequence of one or more requests from R is (BLP) secure, and *iii)* T *is transition secure*.

The problem with changing the security level of subjects and objects was not captured formally as an axiom or property in the Bell and LaPadula, but as an informal design guidance called *tranquility* principle. The tranquility principle states that the classification of active objects should not be changed during normal operation [55]. A subsequent revision of the model [10] introduced a distinction between the level assigned to a subject (*clearance*) and its current level (which could be any level dominated by the clearance), which also implied changing the formulation of the axioms, introducing more flexibility in the control.

Another property included in the Bell and LaPadula model is the *discretionary property* which constraints the set of current accesses b to be a subset of the access matrix M. Intuitively, it enforces discretionary controls.

4.4 Integrity-based mandatory policies: The Biba model

The mandatory policy that we have discussed above protects only the confidentiality of the information; no control is enforced on its integrity. Low classified subjects could still be able to enforce improper indirect modifications to objects they cannot write. With reference to our organization example, for instance, integrity could be compromised if the Trojan Horse implanted by John in the application would write data in file Market (this operation would not be blocked by the secrecy policy). Starting from the principles of the Bell and LaPadula model, Biba [16] proposed a dual policy for safeguarding integrity, which controls the flow of information and prevents subjects to *indirectly* modify information they cannot write. Like for secrecy, each subject and object in the system is assigned an integrity classification. The classifications and the dominance relationship between them are defined as before. Example of integrity levels can be: Crucial

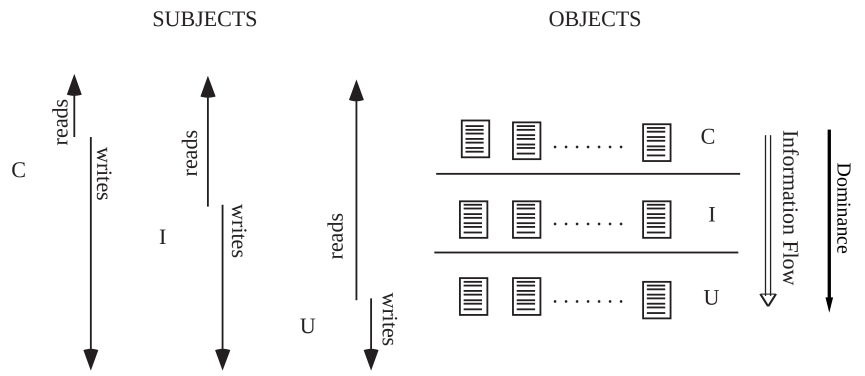

Fig. 7. Information flow for integrity

(C), Important (I), and Unknown (U). The semantics of integrity classifications is as follows. The integrity level associated with a user reflects the user's trustworthiness for inserting, modifying, or deleting information. The integrity level associated with an object reflects both the degree of trust that can be placed on the information stored in the object and the potential damage that could result from unauthorized modifications of the information. Again, categories define the area of competence of users and data. Access control is enforced according to the following two principles:

No-read-down A subject is allowed a read access to an object only if the access class of the object dominates the access class of the subject.

No-write-up A subject is allowed a write access to an object only if the access class of the subject is dominated by the access class of the object.

Satisfaction of these principles safeguard integrity by preventing information stored in low objects (and therefore less reliable) to flow to higher, or incomparable, objects. This is illustrated in Figure 7, where classes composed only of integrity levels (C,I, and U) are taken as example.

The two principles above are the dual of the two principles formulated by Bell and LaPadula. Biba's proposal also investigated alternative criteria for safeguarding integrity, allowing for more dynamic controls. These included the following two policies.

Low-water mark for subjects It constraints write operations according to the no-write-up principle. No restriction is imposed on read operations. However, a subject s that reads an object o has its classification downgraded to the greatest lower bound of the classification of the two, that is, $\lambda'(s) = \mathsf{glb}(\lambda(s), \lambda(o))$.

Low-water mark for objects It constraints read operations according to the no-read-down principle. No restriction is imposed on write operations. However, if a subject s writes an object o, the object has its classification downgraded to the greatest lower bound of the classification of the two, that is, $\lambda'(o) = \mathsf{glb}(\lambda(s), \lambda(o))$.

Intuitively, the two policies attempt to apply a more dynamic behavior in the enforcement of the constraints. The two approaches suffer however of drawbacks. In the low-water mark for subjects approach, the ability of a subject to execute a procedure may depend on the order with which operations are requested: a subject may be denied the execution of a procedure because of read operations executed before. The latter policy cannot actually be considered as safeguarding integrity: given that subjects are allowed to write above their level, integrity compromises can certainly occur; by downgrading the level of the object the policy simply signals this fact.

As it is visible from Figures 6 and 7, secrecy policies allow the flow of information only from lower to higher (secrecy) classes while integrity policies allow the flow of information only from higher to lower (integrity) classes. If both secrecy and integrity have to be controlled, objects and subjects have to be assigned two access classes, one for secrecy control and one for integrity control.

A major limitation of the policies proposed by Biba is that they only capture integrity compromises due to impoproper information flows. However, integrity is a much broader concept and additional aspects should be taken into account (see Section 6.5).

4.5 Applying mandatory policies to databases

The first formulation of the multilevel mandatory policies, and the Bell LaPadula model, simply assumed the existence of objects (information container) to which a classification is assigned. This assumption works well in the operating system context, where objects to be protected are essentially files containing the data. Later studies investigated the extension of mandatory policies to database systems. While in operating systems access classes are assigned to files, database systems can afford a finer-grained classification. Classification can in fact be considered at the level of relations (equivalent to file-level classification in OS), at the level of columns (different properties can have a different classification), at the level of rows (properties referred to a given real world entity or association have the same classification), or at the level of single cells (each data element, meaning the value assigned to a property for a given entity or association, can have a different classification), this latter being the finest possible classification. Early efforts to classifying information in database systems, considered classification at the level of each single element [50, 61]. Element-level classification is clearly appealing since it allows the assignment of a security class to each single real world fact that needs to be represented. For instance, an employee's name can be labeled Unclassified, while his salary can be labeled Secret; also the salary of different employees can take on different classifications. However, the support of fine-grained classifications together with the obvious constraint of

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S

(a)

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Sam	U	Dept1	U	–	U

(b)

Fig. 8. An example of multilevel relation (a) and the Unclassified view on it (b)

maintaining secrecy in the system operation introduces complications. The major complication is represented by the so called *polyinstantiation* problem [49, 60], which is probably one of the main reasons why multilevel databases did not have much success. Generally speaking, polyinstantiation is the presence in the system of multiple instances of the same real world fact or entity, where the instances differ for the access class associated with them.

To illustrate the problem, let us start giving the definition of multilevel relational database. A relational database is composed of a finite set of relations, each defined over a set of attributes $A_1, \ldots, A_n$ (columns of the relation). Each relation is composed of a set of tuples $t_1, \ldots, t_k$ (rows of the relation) mapping attributes to values over their domain. A subset of the attributes, called key attributes, are used to uniquely identify each tuple in the relation, and the following *key constraints* are imposed: *i)* no two tuples can have the same values for the key attributes, and *ii)* key attributes cannot be null. In a multilevel relational database supporting element-level labeling, an access class $\lambda(t[A])$ is associated with each element $t[A]$ in a relation. An example of multilevel relation is illustrated in Figure 8(a). Note that the classification associated with a value does not represent the absolute sensitivity of the value as such, but rather the sensitivity of the fact that the attribute takes on that value for a specific entity in the real world. For instance, classification Secret associated with value `150K` of the last tuple is not the classification of value `150K` by itself, but of the fact that it is the salary of `Sam`.[4]

Access control in multilevel DBMSs applies the two basic principles discussed in Section 4.2, although the no-write-up restriction is usually reduced to the principle of "write at their own level". In fact, while write-up operations can make sense in operating systems, where a file is seen as an information container and subjects may need to append low-level data in a high-level container, element-level classification nullifies this reasoning.

Subjects at different levels have different views on a relation, which is the view composed only of elements they are cleared to see (i.e., whose classification they dominate). For instance, the view of an Unclassified subject on the multilevel relation in Figure 8(a) is the table in Figure 8(b). Note that, in principle, to not

[4] Note that this is not meant to say that the classification of an element is independent of its value. As a matter of fact it can depend on the value; for instance a classification rule may state that all salaries above 100K must be classified as Secret [30].

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S
Ann	U	Dept1	U	100K	U
Sam	U	Dept1	U	100K	U

(a)

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Ann	U	Dept1	U	100K	U
Sam	U	Dept1	U	100K	U

(b)

Fig. 9. An example of a relation with polyinstantiation (a) and the Unclassified view on it (b)

convey information, the Unclassified subject should see no difference between values that are actually `null` in the database and those that are `null` since they have a higher classification.[5] To produce a view consistent with the relational database constraints the classification needs to satisfy at least the following two basic constraints: *i)* the key attributes must be uniformly classified, and *ii)* the classifications of nonkey attributes must dominate that of key attributes. If it were not so, the view at some levels would contain a null value for some or all key attributes (and therefore would not satisfy the key constraints).

To see how polyinstantiation can arise, suppose that an Unclassified subject, whose view on the table in Figure 8(a) is as illustrated in Figure 8(b), requests insertion of tuple `(Ann, Dept1, 100K)`. According to the key constraints imposed by the relational model, no two tuples can have the same value for the key attributes. Therefore if classifications were not taken into account, the insertion could have not been accepted. The database could have two alternative choices: *i)* tell the subject that a tuple with the same key already exists, or *ii)* replace the old tuple with the new one. The first solution introduces a *covert channel*[6], since by rejecting the request the system would be revealing protected information (meaning the existence of a Secret entity named `Ann`), and clearly compromises secrecy. On the other hand, the second solution compromises integrity, since high classified data would be lost, being overridden by the newly inserted tuple. Both solutions are therefore inapplicable. The only remaining solution would then be to accept the insertion and manage the presence of both tuples (see Figure 9(a)). Two tuples would then exist with the same value, but different classification, for their key (*polyinstantiated tuples*). A similar situation happens if the unclassified subject requests to update the salary of `Sam` to value `100K`. Again, telling the subject that a value already exists would compromise secrecy (if the subject is not suppose to distinguish between real nulls and values

[5] Some proposals do not adopt this assumption. For instance, in LDV [43], a special value "restricted" appears in a subject's view to denote the existence of values not visible to the subject.

[6] We will talk more about covert channels in Section 4.6.

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S
Bob	S	Dept2	S	200K	S
Jim	U	Dept1	U	150K	S

(a)

Name	λ_N	**Dept**	λ_D	**Salary**	λ_S
Bob	U	Dept1	U	100K	U
Jim	U	Dept1	U	100K	U
Sam	U	Dept1	U	–	U

(b)

Fig. 10. An example of a relation with polyinstantiation (a) and the Unclassified view on it (b)

for which it does not have sufficient clearance), while overwriting the existing Secret value would compromise integrity (as the Secret salary would be lost). The only remaining solution would therefore seem to be to accept the insertion (Figure 9(a)), implying then the existence of two tuples with the same value and classification for their key, but with different value and classification for one of their attributes (*polyinstantiated elements*). Note that, when producing the view visible to a subject in the presence of polyinstantiation, the DBMS must completely hide those tuples with high polyinstiated values that the subject cannot see. For instance, an unclassified subject querying the relation in Figure 9(a) will see only one tuple for Ann and Sam (see Figure 9(b)).

Polyinstantiation can also occur because of requests by high level subjects. For instance, consider again the relation in Figure 8(a) and assume a Secret subject requests to insert tuple `(Bob, Dept2, 200K)`. A tuple with key `Bob` already exists at level Unclassified. If key uniqueness is to be preserved, the system can either *i)* inform the subject of the conflict and refuse the insertion, or *ii)* overwrite the existing tuple. Again, the solution of refusing insertion is not advisable: although it would not leak protected information, it introduces *denials-of-service*, since high level subjects would not be allowed to insert data. The second solution also is not viable since it would introduce a covert channel due to the effect that the overwriting would have on the view of lower level subjects (which would see the Unclassified tuple disappear). Again, the only possible solution seems to be to accept the insertion and have the two (polyinstantiated) tuples coexist (see Figure 10(a)). A similar problem would arise at the attribute level, for update operations. For instance, if a secret subject requires updating Jim's salary to `150K`, polyinstantiated elements would be introduced (see Figure 10(a)).

Earlier work in multilevel database systems accepted polyinstantiation as an inevitable consequence of fine-grained classification and attempted to clarify the semantics of the database states in the presence of polyinstantiation [50, 61]. For instance, the presence of two tuples with the same value, but different classification, for the primary key (tuple polyinstantiation) can be interpreted as the existence of *two different entities* of the real world (one of which is known only at a higher level). The presence of two tuples with the same key and same key classification but that differ for the value and classification of some of its attributes can be interpreted as a *single* real world entity for which different values are recorded (corresponding to the different beliefs at different levels). However, unfortunately, polyinstantiation quickly goes out of hand, and the execution of few operations could result in a database whose semantics does not appear clear anymore. Subsequent work tried to establish constraints to maintain semantic integrity of the database status [69, 75, 90]. However, probably because of all the complications and semantics confusion that polyinstantiation bears, fine-grained multilevel databases did not have much success, and current DBMSs do not support element-level classification. Commercial systems (e.g., Trusted Oracle [66] and SYBASE Secure SQL Server) support tuple level classification.

It is worth noticing that, although polyinstantiation is often blamed to be the reason why multilevel relational databases did not have success, polyinstantiation is not necessarily always bad. Controlled polyinstantiation may, for example, be useful to support *cover stories* [38, 49], meaning non-true data whose presence in the database is meant to hide the existence of the actual value. Cover stories are useful when the fact that a given data is not released is by itself a cause of information leakage. For instance, suppose that a subject requires access to a hospital's data and the hospital returns, for all its patients, but for few of them, the illness for which they are being cured. Suppose also that HIV never appears as an illness value. Observing this, the recipient may infer that it is probably the case that the patients for which illness is not disclosed suffer from HIV. The hospital could have avoided exposure to such an inference by simply releasing a non-true alternative value (*cover story*) for these patients. Intuitively, cover stories are "lies" that the DBMS says to uncleared subjects not to disclose (directly or indirectly) the actual values to be protected. We do note that, while cover stories are useful for protection, they have raise objections for the possible integrity compromises which they may indirectly cause, as low level subjects can base their actions on cover stories they believe true.

A complicating aspects in the support of a mandatory policy at a fine-grained level is that the definition of the access class to be associated with each piece of data is not always easy [30]. This is the case, for example, of *association* and *aggregation* requirements, where the classification of a set of values (properties, resp.) is higher than the classification of each of the values singularly taken. As an example, while names and salaries in an organization may be considered Unclassified, the association of a specific salary with an employee's name can be considered Secret (association constraint). Similarly, while the location of a single military ship can be Unclassified, the location of all the ships of a fleet can be Secret (aggregation constraint), as by knowing it one could infer that some operations are being planned. Proper data classification assignment is also complicated by the need to take into account possible inference channels [30, 47, 59]. There is an inference channel between a set of data x and a set of data y if, by knowing x a user can infer some information on y (e.g., an inference channel can exist between an employee's taxes and her salary). Inference-aware

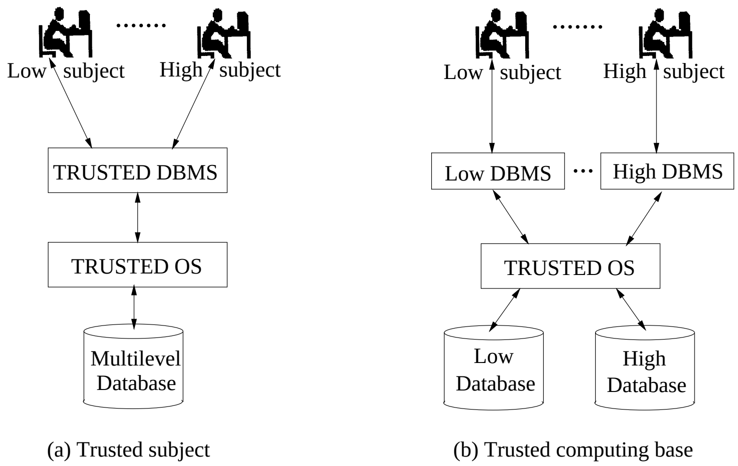

Fig. 11. Multilevel DBMSs architectures

classification requires that no information x be classified at a level lower (or incomparable) than the level of the information y that can be inferred from it. Capturing and blocking all inference channels is a complex process, also because of the intrinsic difficulty of detecting all the semantics relationships between the data that can cause inference channels.

An interesting point that must be taken into account in multilevel database systems is the system architecture, which is concerned with the need of confining subjects accessing a multilevel database to the data that can be made visible to them. This problem comes out in any data system where classification has a finer granularity than the stored objects (e.g., multilevel object-oriented systems). Two possible approaches are [68]:

- *Trusted subject:* data at different levels are stored in a single database (Figure 11(a)). The DBMS itself must be *trusted* to ensure obedience of the mandatory policy (i.e., subjects will not gain access to data whose classification they do not dominate).
- *Trusted computing base:* data are partitioned in different databases, one for each level (Figure 11(b)). In this case only the operating system needs to be trusted since every DBMS will be confined to data which subjects using that DBMS can access. Decomposition and recovery algorithms must be carefully constructed to be correct and efficient [33].

4.6 Limitations of mandatory policies

Although mandatory policies, unlike discretionary ones, provide protection against indirect information leakages they do not guarantee complete secrecy of the information. In fact, secrecy mandatory policies (even with tranquility) control only *overt* channels of information (i.e., flow through *legitimate* channels); they still remain vulnerable to *covert channels*. Covert channels are channels that are not intended for normal communication, but still can be exploited to infer information. For instance, consider the request of a low level subject to write a non-existent high level file (the operation is legitimate since write-up operations are allowed). Now, if the system returns the error, it exposes itself to improper leakages due to malicious high level processes creating and destroying the high level file to signal information to low processes. However, if the low process is not informed of the error, or the system automatically creates the file, subjects may not be signalled possible errors made in legitimate attempts to write. As another example, consider a low level subject that requires a resource (e.g., CPU or lock) that is busy by a high level subject. The system, by not allocating the resource because it is busy, can again be exploited to signal information at lower levels (high level processes can module the signal by requiring or releasing resources). If a low process can see any different result due to a high process operation, there is a channel between them. Channels may also be enacted without modifying the system's response to processes. This is, for example, the case of *timing channels*, that can be enacted when it is possible for a high process to affect the system's response time to a low process. With timing channels the response that the low process receives is always the same, it is the time at which the low process receives the response that communicates information. Therefore, in principle, any *common resource or observable property* of the system state can be used to leak information. Consideration of covert channels requires particular care in the design of the enforcement mechanism. For instance, locking and concurrency mechanisms must be revised and be properly designed [7]. A complication in their design is that care must be taken to avoid the policy for blocking covert channels to introduce denials-of-service. For instance, a trivial solution to avoid covert channels between high and low level processes competing over common resources could be to always give priority to low level processes (possibly terminating high level processes). This approach, however, exposes the systems to denials-of-service attacks whereby low level processes can impede high level (and therefore, presumably, more important) processes to complete their activity.

Covert channels are difficult to control also because of the difficulty of mapping an access control model's primitive to a computer system [64]. For this reason, covert channels analysis is usually carried out in the implementation phase, to make sure that the implementation of the model's primitive is not too weak. Covert channel analysis can be based on tracing the information flows in programs [31], checking programs for shared resources that can be used to transfer information [52], or checking the system clock for timing channels [92]. Beside the complexity, the limitation of such solutions is that covert channels are found out at the end of the development process, where system changes are much more expensive to correct. Interface models have been proposed which attempt to rule out covert channels analysis in the modeling phase [64, 37]. Rather than specifying a particular method to enforce security, interface models specify restrictions on a system's input/output that must be obeyed to avoid covert channels. It is then task of the implementor to determine a method for satisfying the specifications. A well known principle which formed the basis of interface models is the *non-interference* principle proposed by Goguen and Meseguer [40]. Intuitively, non-interference requires that high-level input cannot interfere with low-level output. Non-interference constraints enhance the security properties that can be formalized and proved in the model; it is however important to note that security models do not establish complete security of the system, they merely establish security with respect to a model, they can prove only properties that have been captured into the model.

5 Enriching DAC with mandatory restrictions

As we have discussed in the previous section, mandatory policies guarantee better security than discretionary policies, since they can also control indirect information flows. However, their application may result too rigid. Several proposals have attempted a combination of mandatory flow control and discretionary authorizations. We illustrate some of them in this section.

5.1 The Chinese Wall policy

The Chinese Wall [22] policy was introduced as an attempt to balance commercial discretion with mandatory controls. The goal is to prevent information flows which cause conflict of interest for individual consultants (e.g., an individual consultant should not have information about two banks or two oil companies). However, unlike in the Bell and LaPadula model, access to data is not constrained by the data classifications but by what data the subjects have already accessed. The model is based on a hierarchical organization of data objects as follows:

- *basic objects* are individual items of information (e.g., files), each concerning a single corporation;
- *company datasets* define groups of objects that refer to a same corporation;
- *conflict of interest classes* define company datasets that refer to competing corporations.

Figure 12 illustrates an example of data organization where nine objects of four different corporations, namely `A`,`B`,`C`, and `D`, are maintained. Correspondingly four company datasets are defined. The two conflict of interest classes depicted define the conflicts between `A` and `B`, and between `C` and `D`.

Given the object organization as above, the Chinese Wall policy restricts access according to the following two properties [22]:

Simple security rule A subject s can be granted access to an object o only if the object o:
  - is in the same company datasets as the objects already accessed by s, that is, “within the Wall”, or

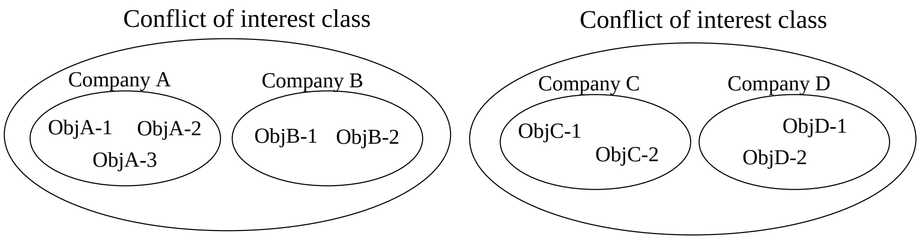

Fig. 12. An example of object organization

- belongs to an entirely different conflict of interest class.

***-property** Write access is only permitted if

- access is permitted by the simple security rule, and
- no object can be read which *i)* is in a different company dataset than the one for which write access is requested, and *ii)* contains unsanitized information.

The term subject used in the properties is to be interpreted as user (meaning access restrictions are referred to users). The reason for this is that, unlike mandatory policies that control processes, the Chinese Wall policy controls users. It would therefore not make sense to enforce restrictions on processes as a user could be able to acquire information about organizations that are in conflict of interest simply running two different processes.

Intuitively, the simple security rule blocks direct information leakages that can be attempted by a single user, while the *-property blocks indirect information leakages that can occur with the collusion of two or more users. For instance, with reference to Figure 12, an indirect improper flow could happen if, *i)* a user reads information from object ObjA-1 and writes it into ObjC-1, and subsequently *ii)* a different user reads information from ObjC-1 and writes it into ObjB-1.

Clearly, the application of the Chinese Wall policy still has some limitations. In particular, strict enforcement of the properties may result too rigid and, like for the mandatory policy, there will be the need for exceptions and support of sanitization (which is mentioned, but not investigated, in [22]). Also, the enforcement of the policies requires keeping and querying the history of the accesses. A further point to take into consideration is to ensure that the enforcement of the properties will not block the system working. For instance, if in a system composed of ten users there are eleven company datasets in a conflict of interest class, then one dataset will remain inaccessible. This aspect was noticed in [22], where the authors point out that there must be at least as many users as the maximum number of datasets which appear together in a conflict of interest class. However, while this condition makes the system operation possible, it cannot ensure it when users are left completely free choice on the datasets they access. For instance, in a system with ten users and ten datasets, again one dataset may remain inaccessible if two users access the same dataset.

Although the model does have some limitations and drawbacks, the Chinese Wall policy represents a good example of *dynamic separation of duty* constraints present in the real world, and has been taken as a reference in the development of several subsequent policies and models (see Section 7).

5.2 Authorization-based information flow policies

Other proposals that tried to overcome the vulnerability of discretionary policies have worked on complementing authorization control with information flow restrictions, interpreting the mandatory and information flow policies [31, 55] in a discretionary context.

The work in [19, 51] proposes interposing, between programs and the actual file system, a protected system imposing further restrictions. In particular, Boebert and Ferguson [19] forces all files to go through a dynamic linker that compares the name of the user who invoked the program, the name of the originator of the program, and the name of the owner of any data files. If a user invokes a program owned by someone else and the program attempts to write the user's files, the dynamic linker will recognize the name mismatch and raise an alarm. Karger [51] proposes instead the specification of name restrictions on the files that programs can access, and the refusal by the system of all access requests not satisfying the given patterns (e.g., a FORTRAN compiler may be restricted to read only files with suffix ".for" and to create only files with suffix ".obj" and ".lis").

McCollum et al. [62] point out data protection requirements that neither the discretionary nor the mandatory policies can effectively handle. They propose a dissemination control system that maintains access control over one's data by attaching to the data object an access control list (imposing access restrictions) that propagates, through subject and object labels, to all objects into which its content may flow. Examples of restrictions can be: NOCONTRACT (meaning no access to contractors) or NOFORN (no releasable to foreign nationals). By propagating restrictions and enforcing the control, intuitively, the approach behaves like a dynamic mandatory policy; however, explicit restrictions in the access list give more flexibility than mandatory security labels. The model also provides support for exceptions (the originator of an ACL can allow restrictions to be waived) and downgrading (trusted subjects can remove restrictions imposed on objects).

A similar approach appears in [85], which, intuitively, interprets the information flow model of Denning [31] in the discretionary context. In [85] each object has two protection attributes: the *current access* and the *potential access*. The current access attribute describes what operations each user can apply on the object (like traditional ACLs). It is a subset of the potential access attribute. The potential access attribute describes what operations which users can potentially apply to the information contained in that object, information that, in the future, may be contained in any object and may be of any type. The potential access attributes therefore control information flow. When a new value of some object y is produced as a function of objects in $x_1, \ldots, x_n$, then the potential access attribute of y is set to be the intersection of the potential access attributes of $x_1, \ldots, x_n$.

Walter et al. [87] propose an interpretation of the mandatory controls within the discretionary context. Intuitively, the policy behind this approach, which we call *strict* policy, is based on the same principles as the mandatory policy. Access control lists are used in place of labels, and the inclusion relationship between sets is used in place of the dominance relationship between labels. Information flow restrictions impose that a process can write an object o only if o is protected in reading at least as all the objects read by the process up to that point. (An object o is at least as protected in reading as another object o' if the set of subjects allowed to read o is contained in the set of subjects allowed to read o'.) Although the discretionary flexibility of specifying accesses is not lost, the overall flexibility is definitely reduced by the application of the strict policy. After having read an object o, a process is completely unable to write any object less protected in reading than o, even if the write operation would not result in any improper information leakage.

Bertino et al. [14] present an enhancement of the strict policy to introduce more flexibility in the policy enforcement. The proposal bases on the observation that whether or not some information can be released also depends on the procedure enacting the release. A process may access sensitive data and yet not release any sensitive information. Such a process should be allowed to bypass the restrictions of the strict policy, thus representing an *exception*. On the other side, the information produced by a process may be more sensitive than the information the process has read. An exception should in this case restrict the write actions otherwise allowed by the strict policy. Starting from these observations, Bertino et al. [14] allow procedures to be granted exceptions to the strict policy. The proposal is developed in the context of object-oriented systems, where the modularity provided by methods associated with objects allows users to identify specific pieces of *trusted* code for which exceptions can be allowed, and therefore provide flexibility in the application of the control. Exceptions can be positive or negative. A positive exception overrides a restriction imposed by the strict policy, permitting an information flow which would otherwise be blocked. A negative exception overrides a permission stated by the strict policy forbidding an information flow which would otherwise be allowed. Two kinds of exceptions are supported by the model: *reply-exceptions* and *invoke-exceptions*. Reply exceptions apply to the information returned by a method. Intuitively, positive reply exceptions apply when the information returned by a method is less sensitive than the information the method has read. Reply exceptions can waive the strict policy restrictions and allow information returned by a method to be disclosed to users not authorized to read the objects that the method has read. Invoke exceptions apply during a method's execution, for write operations that the method requests. Intuitively, positive invoke exceptions apply to methods that are trusted not to leak (through write operations or method invocations) the information they have acquired. The mechanism enforcing the control is based on the notion of *message filter* first introduced by Jajodia and Kogan [46] for the enforcement of mandatory policies in object-oriented systems. The *message filter* is a trusted system component that acts as a reference monitor, intercepting every message exchanged among the objects in a transaction execution to guarantee that no unsafe flow takes place. To check whether a write or create operation should be blocked, the message filter in [14] keeps track of the information transmitted between executions and of the users who are allowed to know (read) it. A write operation on object o is allowed if, based on the ACLs of the objects read and on the exceptions encountered, the information can be released to all users who have read privileges on o.

6 Discretionary access control policies

In Section 2 we introduced the basic concepts of the discretionary policy by illustrating the access matrix (or HRU) model. Although the access matrix still remains a framework for reasoning about accesses permitted by a discretionary policy, discretionary policies have developed considerably since the access matrix was proposed.

6.1 Expanding authorizations

Even early approaches to authorization specifications allowed *conditions* to be associated with authorizations to restrict their validity. Conditions can make the authorization validity dependent on the satisfaction of some system predicates (*system-dependent* conditions) like the time or location of access. For instance, a condition can be associated with the bank-clerks' authorization to access accounts, restricting its application only from machines within the bank building and in working hours. Conditions can also constraint access depending on the content of objects on which the authorization is defined (*content-dependent* conditions). Content-dependent conditions can be used simply as way to determine whether or not an access to the object should be granted or as way to restrict the portion of the object that can be accessed (e.g., a subset of the tuples in a relation). This latter option is useful when the authorization object has a coarser granularity than the one supported by the data model [29]. Other possible conditions that can be enforced can make an access decision depend on accesses previously executed (*history dependent* conditions).

Another feature usually supported even by early approaches is the concept of *user groups* (e.g., Employees, Programmers, Consultants). Groups can be nested and need not be disjoint. Figure 13 illustrates an example of user-group hierarchy. Support of groups greatly simplifies management of authorizations, since a single authorization granted to a group can be enjoyed by all its members. Later efforts moved to the support of groups on all the elements of the authorization triple (i.e., subject, object, and action), where, typically, groups are abstractions hierarchically organized. For instance, in an operating system the hierarchy can

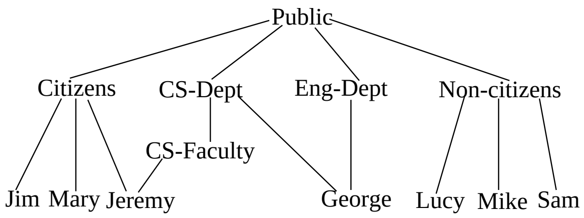

Fig. 13. An example of user-group hierarchy

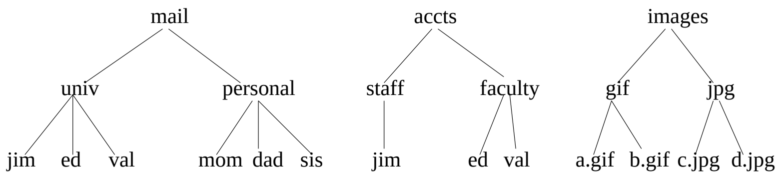

Fig. 14. An example of object hierarchy

reflect the logical file system tree structure, while in object-oriented system it can reflect the class (is-a) hierarchy. Figure 14 illustrates an example of object hierarchy. Even actions can be organized hierarchically, where the hierarchy may reflect an implication of privileges (e.g., write is more powerful than read [70]) or a grouping of sets of privileges (e.g., a "writing privileges" group can be defined containing write, append, and undo [84]). These hierarchical relationships can be exploited *i)* to support preconditions on accesses (e.g., in Unix a subject needs the execute, `x`, privilege on a directory in order to access the files within it), or *ii)* to support authorization implication, that is, authorizations specified on an abstraction apply to all its members. Support of abstractions with implications provides a short hand way to specify authorizations, clearly simplifying authorization management. As a matter of fact, in most situations the ability to execute privileges depends on the membership of users into groups or objects into collections: translating these requirements into basic triples of the form (user,object,action) that then have to be singularly managed is a considerable administrative burden, and makes it difficult to maintain both satisfactory security and administrative efficiency. However, although there are cases where abstractions can work just fine, many will be the cases where exceptions (i.e., authorizations applicable to all members of a group but few) will need to be supported. This observation has brought to the combined support of both *positive* and *negative* authorizations. Traditionally, positive and negative authorizations have been used in mutual exclusion corresponding to two classical approaches to access control, namely:

Closed policy: authorizations specify permissions for an access (like in the HRU model). The closed policy allows an access if there exists a positive authorization for it, and denies it otherwise.

Open policy: (negative) authorizations specify denials for an access. The open policy denies an access if there exists a negative authorization for it, and allows it otherwise.

The open policy has usually found application only in those scenarios where the need for protection is not strong and by default access is to be granted. Most systems adopt the closed policy, which, denying access by default, ensures better protection; cases where information is public by default are enforced with a positive authorization on the root of the subject hierarchy (e.g., Public).

The combined use of positive and negative authorizations was therefore considered as a way to conveniently support exceptions. To illustrate, suppose we wish to grant an authorization to all members of a group composed of one thousand users, except to one specific member Bob. In a closed policy approach, we would have to express the above requirement by specifying a positive authorization for each member of the group except Bob.[7] However, if we combine positive and negative authorizations we can specify the same requirement by granting a positive authorization to the group and a negative authorization to Bob.

The combined use of positive and negative authorizations brings now to the problem of how the two specifications should be treated:

- what if for an access no authorization is specified? (*incompleteness*)
- what if for an access there are both a negative and a positive authorization? (*inconsistency*)

Completeness can be easily achieved by assuming that one of either the open or closed policy operates as a *default*, and accordingly access is granted or denied if no authorization is found for it. Note that the alternative of explicitly requiring completeness of the authorizations is too heavy and complicates administration.

Conflict resolution is a more complex matter and does not usually have a unique answer [48, 58]. Rather, different decision criteria could be adopted, each applicable in specific situations, corresponding to different policies that can be implemented. A natural and straightforward policy is the one stating that "the most specific authorization should be the one that prevails"; after all this is what we had in mind when we introduced negative authorizations in the first place (our example about Bob). Although the most-specific-takes-precedence principle is intuitive and natural and likely to fit in many situations, it is not enough. As a matter of fact, even if we adopt the argument that the most specific authorization always wins (and this may not always be the case) it is not always clear what more specific is:

- what if two authorizations are specified on non-disjoint, but non-hierarchically related groups (e.g., Citizens and CS-Dept in Figure 13)?

[7] In an open policy scenario, the dual example of all users, but a few, who have to be denied an access can be considered.

– what if for two authorizations the most specific relationship appear reversed over different domains? For instance, consider authorizations (CS-Faculty, read+, mail) and (CS-Dept, read−, personal); the first has a more specific subject, while the second has a more specific object (see Figures 13 and 14).

A slightly alternative policy on the same line as the most specific policy is what in [48] is called *most-specific-along-a-path-takes-precedence*. This policy considers an authorization specified on an element x as overriding an authorization specified on a more general element y only for those elements that are members of y because of x. Intuitively, this policy takes into account the fact that, even in the presence of a more specific authorization, the more general authorization can still be applicable because of other paths in the hierarchy. For instance, consider the group hierarchy in Figure 13 and suppose that for an access a positive authorization is granted to Public while a negative authorization is granted to CS-Dept. What should we decide for George? On the one side, it is true that CS-Dept is more specific than Public; on the other side, however, George belongs to Eng-Dept, and for Eng-Dept members the positive authorization is not overridden. While the most-specific-takes-precedence policy would consider the authorization granted to Public as being overridden for George, the most-specific-along-a-path considers both authorizations as applicable to George. Intuitively, in the most-specific-along-a-path policy, an authorization propagates down the hierarchy until overridden by a more specific authorization [35].

The most specific argument does not always apply. For instance, an organization may want to be able to state that consultants should not be given access to private projects, *no exceptions allowed*. However, if the most specific policy is applied, any authorization explicitly granted to a single consultant will override the denial specified by the organization. To address situations like this, some approaches proposed adopting *explicit priorities*. In ORION [70], authorizations are classified as *strong* or *weak*: weak authorizations override each other based on the most-specific policy, and strong authorizations override weak authorizations (no matter their specificity) and *cannot be overridden*. Given that strong authorizations must be certainly obeyed, they are required to be consistent. However, this requirement may be not always be enforceable. This is, for example, the case where groupings are not explicitly defined but depend on the evaluation of some conditions (e.g., "all objects owned by Tom", "all objects created before 1/1/01"). Also, while the distinction between strong and weak authorizations is convenient in many situations and, for example, allows us to express the organizational requirement just mentioned, it is limited to two levels of priority, which may not be enough. Many other conflict resolution policies can be applied. Some approaches, extending the strong and weak paradigm, proposed adopting *explicit priorities*; however, these solutions do not appear viable as the authorization specifications may result not always clear. Other approaches (e.g., [84]) proposed making authorization priority dependent on the *order in which authorizations are listed* (i.e., the authorizations that is encountered first applies). This approach, however, has the drawback that granting or removing an au-

- *Denials-take-precedence:* negative authorizations take precedence (satisfies the "fail safe principle")
- *Most-specific-takes-precedence* the authorization that is "more specific" w.r.t. a partial order (i.e., hierarchy) wins
- *Most-specific-along-a-path-takes-precedence:* the authorization that is "more specific" wins only on the paths passing through it. Intuitively, an authorization propagates down a hierarchy until overridden by a more specific authorization.
- *Strong/weak:* authorizations are classified as strong or weak: weak authorizations override each other based on the most-specific policy, and strong authorizations override weak authorizations (no matter their specificity). Strong authorizations are therefore required to be consistent.
- *Priority level:* each authorization is associated with a priority level, the authorization with the highest priority wins.
- *Positional:* the priority of the authorizations depends on the order in which they appear in the authorization list.
- *Grantor-dependent:* the priority of the authorizations depends on who granted them.
- *Time-dependent* the priority of the authorizations depends on the time at they have been granted (e.g., more recent wins)

Fig. 15. Examples of conflict resolution policies

thorization requires inserting the authorization in the proper place in the list. Beside the administrative burden put on the administrator (who, essentially, has to explicitly solve the conflicts when deciding the order), specifying authorizations implies explicitly writing the ACL associated with the object, and may impede delegation of administrative privileges. Other possible ways of defining priorities, and therefore solving conflicts, can make the authorization's priority dependent on the *time* at which the authorizations was granted (e.g., more recent authorizations prevails) or on priorities between the *grantors*. For instance, authorizations specified by an employee may be overridden by those specified by his supervisor; the authorizations specified by an object's owner may override those specified by other users to whom the owner has delegated administrative authority.

As it is clear from this discussion, different approaches can be taken to deal with positive and negative authorizations. Also, if it is true that some solutions may appear more natural than others, none of them represents "the perfect solution". Whichever approach we take, we will always find one situation for which it does not fit. Also, note that different conflict resolution policies are not mutually exclusive. For instance, one can decide to try solving conflicts with the most-specific-takes-precedence policy first, and apply the denials-take-precedence principle on the remaining conflicts (i.e., conflicting authorizations that are not hierarchically related).

The support of negative authorizations does not come for free, and there is a price to pay in terms of authorization management and less clarity of the specifications. However, the complications brought by negative authorizations are not due to negative authorizations themselves, but to the different semantics that the presence of permissions and denials can have, that is, to the complexity of the different real world scenarios and requirements that may need to be captured. There is therefore a trade-off between expressiveness and simplicity. For this reason, most current systems adopting negative authorizations for exception support impose specific conflict resolution policies, or support a limited form of conflict resolution. For instance, in the Apache server [6], authorizations can be positive and negative and an ordering ("deny,allow" or "allow,deny") can be specified dictating how negative and positive authorizations are to be interpreted. In the "deny,allow" order, negative authorizations are evaluated first and access is allowed by default (open policy). Any client that does not match a negative authorization or matches a positive authorization is allowed access. In the "allow,deny" order, the positive authorizations are evaluated first and access is denied by default (closed policy). Any client that does not match a positive authorization *or* does match a negative authorization will be denied access.

More recent approaches are moving towards the development of flexible frameworks with the support of multiple conflict resolution and decision policies. We will examine them in Section 8.

Other advancements in authorization specification and enforcement have been carried out with reference to specific applications and data models. For instance, authorization models proposed for object-oriented systems (e.g., [2, 35, 71]) exploit the *encapsulation* concept, meaning the fact that access to objects is always carried out through methods (read and write operations being primitive methods). In particular, users granted authorizations to invoke methods can be given the ability to successfully complete them, without need to have the authorizations for all the accesses that the method execution entails. For instance, in OSQL, each derived function (i.e., method) can be specified as supporting *static* or *dynamic* authorizations [2]. A dynamic authorization allows the user to invoke the function, but its successful completion requires the user to have the authorization for all the calls the function makes during its execution. With a *static* authorization, calls made by the function are checked against the creator of the function, instead of those of the calling user. Intuitively, static authorizations behave like the setuid (set user id) option, provided by the Unix operating system that, attached to a program (e.g., lpr) implies that all access control checks are to be performed against the authorizations of the program's owner (instead of those of the caller as it would otherwise be). A similar feature is also proposed in [71], where each method is associated with a principal, and accesses requested during a method execution are checked against the authorization of the method's principal. Encapsulation is also exploited by the Java 2 security model [83] where authorizations can be granted to code, and requests to access resources are checked against the authorizations of the code directly attempting the access.

6.2 Temporal authorizations

Bertino et al. [13] propose extending authorizations with temporal constraints and extending authorization implication with time-based reasoning. Authorizations have associated a validity specified by a temporal expression identifying the instants in which the authorization applies. The temporal expression is formed by a *periodic expression* (e.g., `9 a.m. to 1 p.m. on Working-days`, identifying the periods from 9a.m. to 1p.m. in all days excluding weekends and vacations), and a *temporal interval* bounding the scope of the periodic expression (e.g., `[2/1997,8/1997]`, restricting the specified periods to those between February and August 1997). The model allows also the specification of derivation rules, expressing temporal dependencies among authorizations, that allow the derivation of new authorizations based on the presence or absence of other authorizations in specific periods of time. For instance, it is possible to specify that two users, working on the same project, must receive the same authorizations on given objects, or that a user should receive the authorization to access an object in certain periods, only if nobody else was ever authorized to access the same object in any instant within those periods. Like authorizations, derivation rules are associated with a temporal expression identifying the instants in which the rule applies. A derivation rule is a triple $([t_b,t_e],\ \mathtt{P},\ \mathtt{A}\ \langle\textsc{op}\rangle\ \mathcal{A})$, where interval $[t_b,t_e]$ and period `P` represent the temporal expression, `A` is the authorization to be derived, $\mathcal{A}$ a is boolean formula of authorizations on which derivation is based, and OP is one of the following operators: WHENEVER, ASLONGAS, UPON. The three operators correspond to different temporal relationships between authorizations on which derivation can work, and have the following semantics:

- WHENEVER derives `A` for each instant in $([t_b,t_e],\mathtt{P})$ for which $\mathcal{A}$ is valid.
- ASLONGAS derives `A` for each instant in $([t_b,t_e],\mathtt{P})$ such that $\mathcal{A}$ has been "continuously" valid in $([t_b,t_e],\mathtt{P})$.
- UPON derives `A` from the first instant in $([t_b,t_e],\mathtt{P})$ for which $\mathcal{A}$ is valid up to t_e.

A graphical representation of the semantics of the different temporal operators is given in Figure 16. Intuitively, WHENEVER captures the usual implication of authorizations. For instance, a rule can state that summer-staff can read a document for every instance (i.e., WHENEVER) in the summer of year 2000 in which regular-staff can read it. ASLONGAS works in a similar way but stops the derivation at the first instant in which the boolean formula on which derivation works is not satisfied. For instance, a rule can state that regular-staff can read a document every working day in year 2000 until the first working day in which (i.e., ASLONGAS) summer-staff is allowed for that. Finally, UPON works like a trigger. For instance, a rule can state that Ann can read pay-checks each working day starting from the first working day in year 2000 in which (i.e., UPON) Tom can write pay-checks.

The enforcement mechanism is based on a translation of temporal authorizations and derivation rules into logic programs (Datalog programs with negation and periodicity and order constraints). The materialization of the logic program

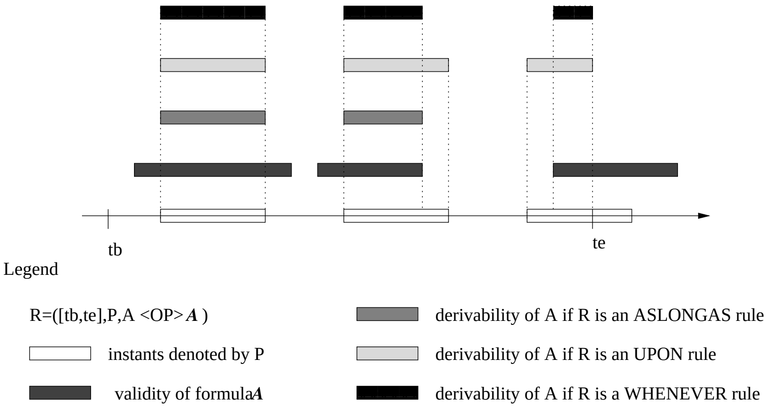

Fig. 16. Semantics of the different temporal operators [13]

guarantees efficient access. The model is focussed on time-based constraints and reasoning and allows expressing authorization relationships and derivation not covered in other models. However, it does not address the enforcement of different implication and conflict resolution policies (conflicts between permissions and denials are solved according to the denials-take-precedence policy).

6.3 A calculus for access control

Abadi et al. [1] present a calculus for access control that combines authentication (i.e., identity check) and authorization control, taking also into account possible delegation of privileges among parties. The calculus is based on the notion of *principals*. Principals are sources of requests and make statements (e.g., "read file tmp"). Principals can be either simple (e.g., users, machines, and communication channels) or composite. Composite principals are obtained combining principals by means of constructors that allow to capture groups and delegations.a Principals can be as follows [1]:

- *Users* and *machines*.
- *Channels*, such as input devices and cryptographic channels.
- *Conjunction of principals*, of the form $A \wedge B$. A request from $A \wedge B$ is a request that both A and B make (it is not necessary that the request be made in concert).
- *Groups*, define groups of principals, membership of principal P_i in group G_i is written $P_i \Longrightarrow G_i$. Disjunction $A \vee B$ denotes a group composed only of A and B.

- Principals in *roles*, of the form A *as* R. The principal A may adopt the role R and act under the name "A *as* R" when she wants to diminish her powers, in particular as protection against blunders.[8]
- Principals *on behalf* of principals, of the form A *for* B. The principal A may delegate authority to B, and B can then act on her behalf, using the identity B *for* A. In most cases, A is a user delegating to a machine B; delegation can also occur between machines.
- Principals *speaking for* other principals, of the form $A \circ B$, denoting that B speaks on behalf of A, but not necessarily with a proof that A has delegated authority to B.

The process of determining whether a request from a principal should be granted or denied is based on a modal logic that extends the algebra of principals and serves as a basis for different algorithms and protocols. Intuitively, a request on an object will be granted if it is authorized according to the authorizations stated in the ACL of the object and the implication relationships and delegations holding among principals.

6.4 Administrative policies

Administrative policies determine who is authorized to modify the allowed accesses. This is one of the most important, and probably least understood, aspect of access controls. In multilevel mandatory access control the allowed accesses are determined entirely on basis of the security classification of subjects and objects. Security levels are assigned to users by the security administrator. Security levels of objects are determined by the system on the basis of the levels of the users creating them. The security administrator is typically the only one who can change security levels of subjects and objects. The administrative policy is therefore very simple. Discretionary access control permits a wide range of administrative policies. Some of these are described below.

- *Centralized:* A single authorizer (or group) is allowed to grant and revoke authorizations to the users.
- *Hierarchical:* A central authorizer is responsible for assigning administrative responsibilities to other administrators. The administrators can then grant and revoke access authorizations to the users of the system. Hierarchical administration can be applied, for example, according to the organization chart.
- *Cooperative:* Special authorizations on given resources cannot be granted by a single authorizer but need cooperation of several authorizers.

[8] Note that there is a difference in the semantics assigned to roles in [1] and in role-based access control model (see Section 7). In [1] a principal's privileges always diminish when the principal takes on some role; also an implication relationship is enforced allowing a principal P to use authorizations granted to any principal of the form P *as* R.

- *Ownership:* Each object is associated with an owner, who generally coincides with the user who created the object. Users can grant and revoke authorizations on the objects they own.
- *Decentralized:* Extending the previous approaches, the owner of an object (or its administrators) can delegate other users the privilege of specifying authorizations, possibly with the ability of further delegating it.

Decentralized administration is convenient since it allows users to delegate administrative privileges to others. Delegation, however, complicates the authorization management. In particular, it becomes more difficult for users to keep track of who can access their objects. Furthermore, revocation of authorizations becomes more complex. There are many possible variations on the way decentralized administration works, which may differ in the way the following questions are answered.

- what is the granularity of administrative authorizations?
- can delegation be restricted, that is, can the grantor of an administrative authorization impose restrictions on the subjects to which the recipient can further grant the authorization?
- who can revoke authorizations?
- what about authorizations granted by the revokee?

In general, existing decentralized policies allow users to grant administration for a specific privilege (meaning a given access on a given object). They do not allow, however, to put constraints on the subjects to which the recipient receiving administrative authority can grant the access. This feature could, however, result useful. For instance, an organization could delegate one of its employees to granting access to some resources constraining the authorizations she can grant to employees working within her laboratory. Usually, authorizations can be revoked only by the user who granted them (or, possibly, by the object's owner). When an administrative authorization is revoked, the problem arises of dealing with the authorizations specified by the users from whom the administrative privilege is being revoked. For instance, suppose that Ann gives Bob the authorization to read File1 and gives him the privilege of granting this authorization to others (in some systems, such capability of delegation is called *grant option* [42]). Suppose then that Bob grants the authorization to Chris, and susequently Ann revokes the authorization from Bob. The question now is: what should happen to the authorization that Chris has received? To illustrate how revocation can work it is useful to look at the history of System R [42]. In the System R authorization model, users creating a table can grant other users access privileges on it. Authorizations can be granted with the *grant-option*. If a user receives the authorization for an access with the grant-option she can grant the access (and the grant option on it) to others. Intuitively, this introduces a chain of authorizations. The original System R policy, which we call *(time-based) cascade* revocation, adopted the following semantics for revocation: when a user is revoked the grant option on an access, all authorizations that

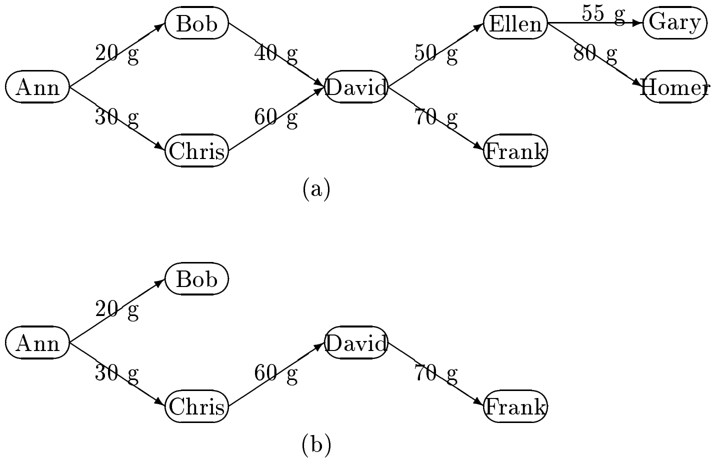

Fig. 17. Example of the original System-R, time-based cascade revocation

she granted and could not have been granted had the revoked authorization not been present, should also be (recursively) deleted. The revocation is recursive since it may, in turn, cause other authorizations to be deleted. More precisely, let $AUTH$ be the initial authorization state and $G_1, \ldots, G_n$ be a sequence of grant requests (history) that produced authorization state $AUTH'$. The revocation of a grant G_k should result in authorization state $AUTH''$ as if G_k had never been granted, that is, resulting from history $G_1, \ldots, G_{k-1}, G_{k+1}, \ldots, G_n$. Enforcement of this revocation semantics requires to keep track of *i)* who granted which authorization, and *ii)* the time at which the authorization was granted. To illustrate, consider the sequence of grant operations pictured in Figure 17(a), referred to the delegation of a specific privilege. Here, nodes represent users, and arcs represent the granting of a specific access from one user to another. The label associated with the arc states the time at which the authorization was granted and whether the grant option was granted as well. For instance, Ann granted the authorization, with the grant option, to Bob at time 20, and to Chris at time 30. Suppose now that Bob revokes the authorization he granted to David. According to the revocation semantics to be enforced, the authorization that David granted to Ellen must be deleted as well, since it was granted at time 50 when, had David not hold the authorizations being revoked, the grant request would have been denied. Consequently, and for the same reasoning, the two authorizations granted by Ellen also need to be deleted, resulting in the authorization state of Figure 17(b).

Although the time-based cascade revocation has a clean semantics, it is not always accepted. Deleting all authorizations granted in virtue of an authorization that is being revoked is not always wanted. In many organizations, the authorizations that users possess are related to their particular tasks or functions within the organization. Suppose there is a change in the task or function of a user (say, because of a job promotion). This change may imply a change in the responsibilities of the user and therefore in her privileges. New authorizations will be granted to the user and some of her previous authorizations will be revoked. Applying a recursive revocation will result in the undesirable effect of deleting all authorizations the revokee granted and, recursively, all the authorizations granted through them, which then will need to be re-issued. Moreover, all application programs depending on the revoked authorizations will be invalidated. An alternative form of revocation was proposed in [15], where *non-cascade* revocation is introduced. Instead of deleting all the authorizations granted by the revokee in virtue of the authorizations being revoked, non-recursive revocation re-specifies them to be under the authority of the revoker, which can then retain or selectively delete them. The original time-based revocation policy of System R, was changed to not consider time anymore. In SQL:1999 [28] revocation can be requested *with* or *without cascade*. Cascade revocation recursively deletes authorizations if the revokee does not hold anymore the grant option for the access. However, if the revokee still holds the grant option for the access, the authorizations she granted are not deleted (regardless of time they were granted). For instance, with reference to Figure 17(a), the revocation by Bob of the authorization granted to David, would only delete the authorization granted to David by Bob. Ellen's authorization would still remain valid since David still holds the grant option of the access (because of the authorization from Chris). With the non cascade option the system rejects the revoke operation if its enforcement would entail deletion of other authorizations beside the one for which revocation is requested.

6.5 Integrity policies

In Section 4.4 we illustrated a mandatory policy (namely Biba's model) for protecting information integrity. Biba's approach, however, suffers of two major drawbacks: *i)* the constraints imposed on the information flow may result too restrictive, and *ii)* it only controls integrity intended as the prevention of a flow of information from low integrity objects to high integrity objects. However, this notion of one-directional information flow in a lattice captures only a small part of the data integrity problem [74].

Integrity is concerned with ensuring that no resource (including data and programs[9]) has been modified in an *unauthorized* or *improper* way and that the data stored in the system correctly reflect the real world they are intended to represent (i.e., that users expect). Integrity preservation requires prevention of frauds and errors, as the term "improper" used above suggests: violations to data integrity are often enacted by legitimate users executing authorized actions but misusing their privileges.

Any data management system today has functionalities for ensuring integrity [8]. Basic integrity services are, for example, *concurrency control* (to

[9] Programs improperly modified can fool the access control and bypass the system restrictions, thus violating the secrecy and/or integrity of the data (see Section 3).

ensure correctness in case of multiple processes concurrently accessing data) and *recovery* techniques (to reconstruct the state of the system in the case of violations or errors occur). Database systems also support the definition and enforcement of integrity constraints, that define the valid states of the database constraining the values that it can contain. Also, database systems support the notion of *transaction*, which is a sequence of actions for which the *ACID* properties must be ensured, where the acronym stands for: *Atomicity* (a transaction is either performed in its entirety or not performed at all); *Consistency* (a transaction must preserve the consistency of the database); *Isolation* (a transaction should not make its updates visible to other transactions until it is committed); and *Durability* (changes made by a transaction that has committed must never be lost because of subsequent failures).

Although rich, the integrity features provided by database management systems are not enough: they are only specified with respect to the data and their semantics, and do not take into account the subjects operating on them. Therefore, they can only protect against obvious errors in the data or in the system operation, and not against misuses by subjects [23]. The task of a security policy for integrity is therefore to fill this gap and control data modifications and procedure executions with respect to the subjects performing them. An attempt in this respect is represented by the Clark and Wilson's proposal [25], where the following four basic criteria for achieving data integrity are defined.

1. *Authentication.* The identity of all users accessing the system must be properly authenticated (this is an obvious prerequisite for correctness of the control, as well as for establishing accountability).
2. *Audit.* Modifications should be logged for the purpose of maintaining an audit log that records every program executed and the user who executed it, so that changes could be undone.
3. **Well-formed transactions** Users should not manipulate data arbitrarily but only in constrained ways that ensure data integrity (e.g., double entry bookkeeping in accounting systems). A system in which transactions are well-formed ensures that only legitimate actions can be executed. In addition, well-formed transactions should provide logging and serializability of resulting subtransactions in a way that concurrency and recovery mechanisms can be established.
4. **Separation of duty** The system must associate with each user a valid set of programs to be run. The privileges given to each user must satisfy the separation of duty principle. Separation of duty prevents authorized users from making improper modifications, thus preserving the consistency of data by ensuring that data in the system reflect the real world they represent.

While authentication and audit are two common mechanisms for any access control system, the latter two aspects are peculiar to the Clark and Wilson proposal.

The definition of well-formed transaction and the enforcement of separation of duty constraints is based on the following concepts.

C1: All IVPs must ensure that all CDIs are in a valid state when the IVP is run.
C2: All TPs must be certified to be valid (i.e., preserve validity of CDIs' state)
C3: Assignment of TPs to users must satisfy separation of duty
C4: The operations of TPs must be logged
C5: TPs execute on UDIs must result in valid CDIs
E1: Only certified TPs can manipulate CDIs
E2: Users must only access CDIs by means of TPs for which they are authorized
E3: The identity of each user attempting to execute a TP must be authenticated
E4: Only the agent permitted to certify entities can change the list of such entities associated with other entities

Fig. 18. Clark and Wilson integrity rules

– *Constrained Data Items.* CDIs are the objects whose integrity must be safeguarded.
– *Unconstrained Data Items.* UDIs are objects that are not covered by the integrity policy (e.g., information typed by the user on the keyboard).
– *Integrity Verification Procedures.* IVPs are procedures meant to verify that CDIs are in a valid state, that is, the IVPs confirm that the data conforms to the integrity specifications at the time the verification is performed.
– *Transformation Procedures.* TPs are the only procedures (well-formed procedures) that are allowed to modify CDIs or to take arbitrary user input and create new CDIs. TPs are designed to take the system from one valid state to the next

Intuitively, IVPs and TPs are the means for enforcing the well-formed transaction requirement: all data modifications must be carried out through TPs, and the result must satisfy the conditions imposed by the IVPs.

Separation of duty must be taken care of in the definition of authorized operations. In the context of the Clark and Wilson's model, authorized operations are specified by assigning to each user a set of well-formed transactions that she can execute (which have access to constraint data items). Separation of duty requires the assignment to be defined in a way that makes it impossible for a user to violate the integrity of the system. Intuitively, separation of duty is enforced by splitting operations in subparts, each to be executed by a different person (to make frauds difficult). For instance, any person permitted to create or certify a well-formed transaction should not be able to execute it (against production data).

Figure 18 summarizes the nine rules that Clark and Wilson presented for the enforcement of system integrity. The rules are partitioned into two types: certification (C) and enforcement (E). Certification rules involve the evaluation of transactions by an administrator, whereas enforcement is performed by the system.

The Clark and Wilson's proposal outlines good principles for controlling integrity. However, it has limitations due to the fact that it is far from formal and it is unclear how to formalize it in a general setting.

7 Role-Based Access Control Policies

Role-based access control (RBAC) is an alternative to traditional discretionary (DAC) and mandatory access control (MAC) policies that is attracting increasing attention, particularly for commercial applications. The main motivation behind RBAC is the necessity to specify and enforce enterprise-specific security policies in a way that maps naturally to an organization's structure. In fact, in a large number of business activities a user's identity is relevant only from the point of view of accountability. For access control purposes it is much more important to know what a user's organizational responsibilities are, rather than who the user is. The conventional discretionary access controls, in which individual user ownership of data plays such an important part, are not a good fit. Neither are the full mandatory access controls, in which users have security clearances and objects have security classifications. Role-based access control tries to fill in this gap by merging the flexibility of explicit authorizations with additionally imposed organizational constraints.

7.1 Named protection domain

The idea behind role-based access control is grouping privileges (i.e., authorizations). The first work proposing collecting privileges for authorization assignment is probably the work by Baldwin [9], where the concept of *named protection domain* (NPD) is introduced as a way to simplify security management in an SQL-based framework. Intuitively, a named protection domain identifies a set of privileges (those granted to the NPD) needed to accomplish a well-defined task. For instance, in a bank organization, an NPD `Accounts_Receivable` can be defined to which all the privileges needed to perform the account-receivable task are granted. NPD can be granted to users as well as to other NPDs, thus forming a chain of privileges. The authorization state can be graphically represented as a directed acyclic graph where nodes correspond to privileges, NPDs, and users, while arcs denote authorization assignments. An example of privilege graph is illustrated in Figure 19, where three NPDs (`Accounts_Receivable`, `Accounts_Payable`, and `Accounts_Supervisor`) and the corresponding privileges are depicted. Users can access objects only by activating NPDs holding privileges on them. Users can only activate NPDs that have been directly or indirectly assigned to them. For instance, with reference to Figure 19, `Bob` can activate any of three NPDs, thus acquiring the corresponding privileges. To enforce *least privilege*, users are restricted to activate only one NPD at the time. NPDs can also be used to group users. For instance, a NPD named `Employees` can be defined which corresponds to the set of employees of an organization. NPDs correspond to the current concept of *roles* in SQL:1999 [28].

7.2 Role-based policies

Role-based policies regulate the access of users to the information on the basis of the organizational activities and responsibility that users have in a system.

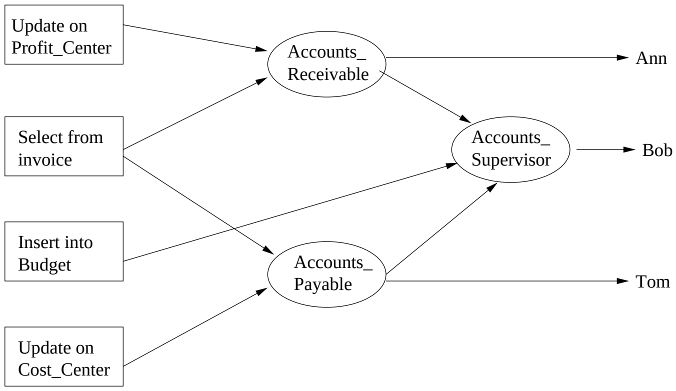

Fig. 19. An example of NPD privilege graph [9]

Although different proposals have been made (e.g., [3, 36, 45, 56, 67, 76, 80]), the basic concepts are common to all approaches. Essentially, role based policies require the identification of *roles* in the system, where a role can be defined as a set of actions and responsibilities associated with a particular working activity. The role can be widely scoped, reflecting a user's job title (e.g., `secretary`), or it can be more specific, reflecting, for example, a task that the user needs to perform (e.g., `order_processing`). Then, instead of specifying all the accesses each users is allowed to execute, access authorizations on objects are specified for roles. Users are then given authorizations to adopt roles (see Figure 20). The user playing a role is allowed to execute all accesses for which the role is authorized. In general, a user can take on different roles on different occasions. Also the same role can be played by several users, perhaps simultaneously. Some proposals for role-based access control (e.g., [76, 80]) allow a user to exercise multiple roles at the same time. Other proposals (e.g., [28, 48]) limit the user to only one role at a time, or recognize that some roles can be jointly exercised while others must be adopted in exclusion to one another. It is important to note the difference between groups (see Section 6) and roles: groups define sets of users while roles define sets of privileges. There is a semantic difference between them: roles can be "activated" and "deactivated" by users at their discretion, while group membership always applies, that is, users cannot enable and disable group memberships (and corresponding authorizations) at their will. However, since roles can be defined which correspond to organizational figures (e.g., `secretary`, `chair`, and `faculty`), a same "concept" can be seen both as a group and as a role.

The role-based approach has several advantages. Some of these are discussed below.

Authorization management Role-based policies benefit from a logical independence in specifying user authorizations by breaking this task into two parts: *i)* assignement of roles to users, and *ii)* assignement of authorizations

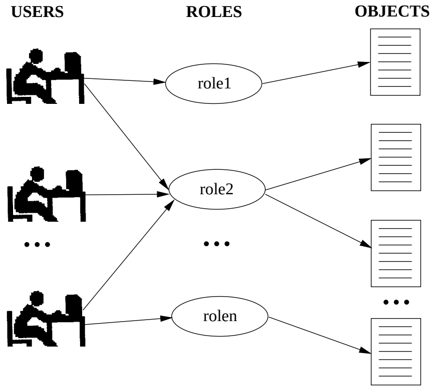

Fig. 20. Role-based access control

to access objects to roles. This greatly simplifies the management of the security policy: when a new user joins the organization, the administrator only needs to grant her the roles corresponding to her job; when a user's job changes, the administrator simply has to change the roles associated with that user; when a new application or task is added to the system, the administrator needs only to decide which roles are permitted to execute it.

Hierarchical roles In many applications there is a natural hierarchy of roles, based on the familiar principles of generalization and specialization. Figure 21 illustrates an example of role hierarchy: each role is represented as a node and there is an arc between a specialized role and its generalization. The role hierarchy can be exploited for authorization implication. For instance, authorizations granted to roles can be propagated to their specializations (e.g., the `secretary` role can be allowed all accesses granted to `adm_staff`). Authorization implication can also be enforced on role assignments, by allowing users to activate all generalizations of the roles assigned to them (e.g., a user allowed to activate `secretary` will also be allowed to activate role `adm_staff`). Authorization implication has the advantage of further simplifying authorization management. Note however that not always implication may be wanted, as propagating all authorizations is contrary to the least privilege principle. The hierarchy has also been exploited in [77] for the definition of administrative privileges: beside the hierarchy of organizational roles, an additional hierarchy of administrative roles is defined; each administrative role can be given authority over a portion of the role hierarchy.

Least privilege Roles allow a user to sign on with the least privilege required for the particular task she needs to perform. Users authorized to powerful roles do not need to exercise them until those privileges are actually needed. This minimizes the danger of damage due to inadvertent errors, Trojan Horses, or intruders masquerading as legitimate users.

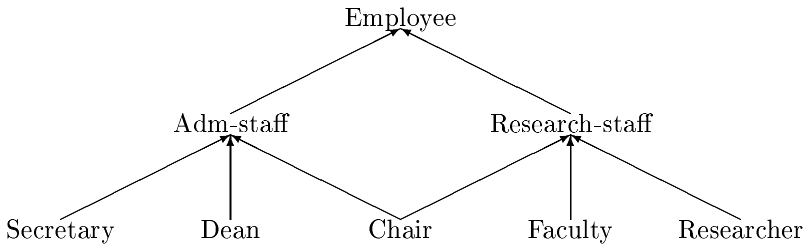

Fig. 21. An example of role hierarchy

Separation of duties Separation of duties refer to the principle that no user should be given enough privileges to misuse the system on their own. For instance, the person authorizing a paycheck should not be the same person who can prepare them. Separation of duties can be enforced either statically (by defining conflicting roles, that is, roles which cannot be executed by the same user) or dynamically (by enforcing the control at access time). An example of dynamic separation of duty restriction is the two-person rule. The first user to execute a two-person operation can be any authorized user, whereas the second user can be any authorized user different from the first [79].

Constraints enforcement Roles provide a basis for the specification and enforcement of further protection requirements that real world policies may need to express. For instance, cardinality constraints can be specified, that restrict the number of users allowed to activate a role or the number of roles allowed to exercise a given privilege. The constraints can also be dynamic, that is, be imposed on roles activation rather than on their assignment. For instance, while several users may be allowed to activate role `chair`, a further constraint can require that at most one user at a time can activate it.

Role-based policies represent a promising direction and a useful paradigm for many commercial and government organizations. However, there is still some work to be done to cover all the different requirements that real world scenarios may present. For instance, the simple hierarchical relationship as intended in current proposals may not be sufficient to model the different kinds of relationships that can occur. For example, a secretary may need to be allowed to write specific documents *on behalf* of her manager, but neither role is a specialization of the other. Different ways of propagating privileges (delegation) should then be supported. Similarly, administrative policies should be enriched. For instance, the traditional concept of ownership may not apply anymore: a user does not necessarily own the objects she created when in a given role. Also, users' identities should not be forgotten. If it true that in most organizations, the role (and not the identity) identifies the privileges that one may execute, it is also true that in some cases the requestor's identity needs to be considered even when a role-based policy is adopted. For instance, a doctor may be allowed to specify treatments and access files but she may be restricted to treatments and files for her own patients, where the doctor-patient relationships is defined based on their identity.

8 Advanced access control models

Throughout the chapter we investigated different issues concerning the development of an access control system, discussing security principles, policies, and models proposed in the literature. In this section we illustrate recent proposals and ongoing work addressing access control in emerging applications and new scenarios.

8.1 Logic-based authorization languages

As discussed in Section 6, access control systems based only on the closed policy clearly have limitations. The support of abstractions and authorization implications along them and the support of positive and negative authorizations provide more flexibility in the authorization specifications. As we have seen, several access control policies can be applied in this context (e.g., denials-take-precedence, most-specific-takes-precedence, strong and weak) and have been proposed in the literature. Correspondingly, several authorization models have been formalized and access control mechanisms enforcing them implemented. However, each model, and its corresponding enforcement mechanism, implements a single specified policy, which is in fact built into the mechanism. As a consequence, although different policy choices are possible in theory, each access control system is in practice bound to a specific policy. The major drawback of this approach is that a single policy simply cannot capture all the protection requirements that may arise over time. As a matter of fact, even within a single system:

- different users may have different protection requirements;
- a single user may have different protection requirements on different objects;
- protection requirements may change over time.

When a system imposes a specific policy on users, they have to work within the confines imposed by the policy. When the protection requirements of an application are different from the policy built into the system, in most cases, the only solution is to implement the policy as part of the application code. This solution, however, is dangerous from a security viewpoint since it makes the tasks of verification, modification, and adequate enforcement of the policy difficult.

Recent proposals have worked towards languages and models able to express, in a single framework, different access control policies, to the goal of providing a single mechanism able to enforce multiple policies. Logic-based languages, for their expressive power and formal foundations, represent a good candidate. The first work investigating logic-languages for the specification of authorizations is the work by Woo and Lam [91]. Their proposal makes the point for the need of flexibility and extensibility in access specifications and illustrates how these advantages can be achieved by abstracting from the low level authorization triples and adopting a high level authorization language. Their language is essentially a many-sorted first-order language with a rule construct, useful to express authorization derivations and therefore model authorization implications and default decisions (e.g., closed or open policy). The use of a very general language, which has almost the same expressive power of first order logic, allows the expression of different kinds of authorization implications, constraints on authorizations, and access control policies. However, as a drawback, authorization specifications may result difficult to understand and manage. Also, the trade-off between expressiveness and efficiency seems to be strongly unbalanced: the lack of restrictions on the language results in the specification of models which may not even be decidable and therefore will not be implementable. As noted in [48], Woo and Lam's approach is based on truth in extensions of arbitrary default theories, which is known, even in the propositional case to be NP-complete, and in the first order case, is worse than undecidable.

Starting from these observations, Jajodia et al. [48] worked on a proposal for a logic-based language that attempted to balance flexibility and expressiveness on the one side, and easy management and performance on the other. The language allows the representation of different policies and protection requirements, while at the same time providing understandable specifications, clear semantics (guaranteeing therefore the behavior of the specifications), and bearable data complexity. Their proposal for a Flexible Authorization Framework (FAF) identifies a polynomial time (in fact quadratic time) data complexity fragment of default logic; thus resulting effectively implementable. The language identifies the following predicates for the specification of authorizations. (Below s, o, and a denote a subject, object, and action term, respectively, where a term is either a constant value in the corresponding domain or a variable ranging over it).

cando(o,s,⟨*sign*⟩a) represents authorizations explicitly inserted by the security administrator. They represent the accesses that the administrator wishes to allow or deny (depending on the sign associated with the action).

dercando(o,s,⟨*sign*⟩a) represents authorizations derived by the system using logical rules of inference (modus ponens plus rules for stratified negation). Logical rules can express hierarchy-based authorization derivation (e.g., propagation of authorizations from groups to their members) as well as different implication relationships that may need to be represented.

do(o,s,⟨*sign*⟩a) definitely represents the accesses that must be granted or denied. Intuitively, `do` enforces the conflict resolution and access decision policies, that is, it decides whether to grant or deny the access possibly solving existing conflicts and enforcing default decisions (in the case where no authorization has been specified for an access).

done(o,s,r,a,t) keeps the history of the accesses executed. A fact of the form `done(o,s,r,a,t)` indicates that `s` operating in role `r` executed action `a` on object `o` at time `t`.

error signals errors in the specification or use of authorizations; it can be used to enforce static and dynamic constraints on the specifications.

In addition, the language considers predicates, called `hie-`predicates, for the evaluation of hierarchical relationships between the elements of the data system (e.g., user's membership in groups, inclusion relationships between objects). The language also allows the inclusion of additional application-specific predicates, called `rel-` predicates. These predicates can capture the possible different relationships, existing between the elements of the data system, that may need to be taken into account by the access control system. Examples of these predicates can be `owner(user,object)`, which models ownership of objects by users, or `supervisor(user1,user2)`, which models responsibilities and controls between users according to the organizational structure.

Authorization specifications are stated as logic rules defined on the predicates of the language. To ensure clean semantics and implementability, the format of the rules is restricted to guarantee (local) stratification of the resulting program (see Figure 22).[10] The stratification also reflects the different semantics given to the predicates: `cando` will be used to specify basic authorizations, `dercando` to enforce implication relationships and produce derived authorizations, and `do` to take the final access decision. Stratification ensures that the logic program corresponding to the rules has a unique stable model, which coincides with the well founded semantics. Also, this model can be effectively computed in polynomial time. The authors also present a materialization technique for producing and storing the model corresponding to a set of logical rules. Materialization has been usually coupled with logic-based authorization languages. Indeed, given a logic program whose rules correspond to an authorization specification in the given language, one can assess a request to execute a particular action on an object by checking if it is true in the unique stable model of the logic program. If so, the request is authorized, otherwise it is denied. However, when implementing an algorithm to support this kind of evaluation, one needs to consider the following facts:

- the request should be either authorized or denied very fast, and
- changes to the specifications are far less frequent than access requests.

Indeed, since access requests happen all the time, the security architecture should optimize the processing of these requests. Therefore, Jajodia et al. [48] propose implementing their FAF with a *materialized view architecture*, which maintains the model corresponding to the authorization specifications. The model is computed on the initial specifications and updated with incremental maintenance strategies.

[10] A program is locally stratified if there is no recursion among predicates going through negation.

Stratum	**Predicate**	**Rules defining predicate**
0	`hie`-predicates `rel`-predicates `done`	base relations. base relations. base relation.
1	`cando`	body may contain `done`, `hie`- and `rel`-literals.
2	`dercando`	body may contain `cando`, `dercando`, `done`, `hie`-, and `rel`- literals. Occurrences of `dercando` literals must be positive.
3	`do`	in the case when head is of the form `do`$(\_, \_, +a)$ body may contain `cando`, `dercando`, `done`, `hie`- and `rel`- literals.
4	`do`	in the case when head is of the form `do`$(o, s, -a)$ body contains just one literal $\neg$`do`$(o, s, +a)$.
5	`error`	body may contain `do`, `cando`, `dercando`, `done`, `hie`-, and `rel`- literals.

Fig. 22. Rule composition and stratification of the proposal in [48]

8.2 Composition of access control policies

In many real world situations, access control needs to combine restrictions independently stated that should be enforced as one, while retaining their independence and administrative autonomy. For instance, the global policy of a large organization can be the combination of the policies of its different departments and divisions as well as of externally imposed constraints (e.g., privacy regulations); each of these policies should be taken into account while remaining independent and autonomously managed. Another example is represented by the emerging dynamic coalition scenarios where different parties, coming together for a common goal for a limited time, need to merge their security requirements in a controlled way while retaining their autonomy. Since existing frameworks assume a single monolithic specification of the entire access control policy, the situations above would require translating and merging the different component policies into a single "program" in the adopted access control language. While existing languages are flexible enough to obtain the desired combined behavior, this method has several drawbacks. First, the translation process is far from being trivial; it must be done very carefully to avoid undesirable side effects due to interference between the component policies. Interference may result in the combined specifications not reflecting correctly the intended restrictions. Second, after translation it is not possible anymore to operate on the individual components and maintain them autonomously. Third, existing approaches cannot take into account incomplete policies, where some components are not (completely) known a priori (e.g., when somebody else is to provide that component). Starting from these observations, Bonatti et al. [20] make the point for the need of a policy composition framework by which different component policies can be integrated while retaining their independence. They propose an algebra for combining security policies. Compound policies are formulated as expressions of the algebra, constructed by using the following operators.

Addition merges two policies by returning their union. For instance, in an organization composed of different divisions, access to the main gate can be authorized by any of the administrator of the divisions (each of them knows which users need access to reach their division). The totality of the accesses through the main gate to be authorized should then be the union of the statements of each division. Intuitively, additions can be applied in any situation where accesses can be authorized if allowed by any of the component policies.

Conjunction merges two policies by returning their intersection. For instance, consider an organization in which divisions share certain documents (e.g., clinical folders of patients). An access to a document may be allowed only if all the authorities that have a say on the document agree on it. That is, if the corresponding authorization triple belongs to the intersection of their policies.

Subtraction restricts a policy by eliminating all the accesses in a second policy. Intuitively, subtraction specifies exceptions to statements made by a policy, and encompasses the functionality of negative authorizations in existing approaches.

Closure closes a policy under a set of derivation (i.e., implication) rules, w.r.t. which the algebra is parametric. Rules can be, for example, expressed with a logic-based language (e.g., [48]).

Scoping restriction restricts the application of a policy to a given set of subjects, objects, and actions that satisfy certain properties (i.e., belong to a given class). It is useful to enforce authority confinement (e.g., authorizations specified in a given component can be referred only to specific subjects and objects).

Overriding replaces portion of a policy with another. For instance, a laboratory policy may impose authorizations granted by the lab tutors to be overridden by the department policy (which can either confirm the authorization or not) for students appearing in a blacklist for infraction to rules.

Template defines a partially specified (i.e., parametric) policy that can be completed by supplying the parameters. Templates are useful for representing policies where some components are to be specified at a later stage. For instance, the components might be the result of further policy refinement, or might be specified by a different authority.

Enforcement of compound policies is based on a translation from policy expressions into logic programs, which provide executable specifications compatible with different evaluation strategies (e.g., run time, materialized view, partial evaluation). The logic program simply provides an enforcement mechanism and is transparent to the users, who can therefore enjoy the simplicity of algebra expressions. The modularity of the algebra, where each policy can be seen as a different component, provides a convenient way for reasoning about policies at different levels of abstractions. Also, it allows for the support of heterogeneous policies and policies that are unknown a priori and can only be queried at access control time.

8.3 Certificate-based access control

Today's Globally Internetworked Infrastructure connects remote parties through the use of large scale networks, such as the World Wide Web. Execution of activities at various levels is based on the use of remote resources and services, and on the interaction between different, remotely located, parties that may know little about each other. In such a scenario, traditional assumptions for establishing and enforcing access control regulations do not hold anymore. For instance, a server may receive requests not just from the local community of users, but also from remote, previously unknown users. The server may not be able to authenticate these users or to specify authorizations for them (with respect to their identity). The traditional separation between *authentication* and *access control* cannot be applied in this context, and alternative access control solutions should be devised. A possible solution to this problem is represented by the use of digital certificates (or credentials), representing statements certified by given entities (e.g., certification authorities), which can be used to establish properties of their holder (such as identity, accreditation, or authorizations) [39]. Trust-management systems (e.g., PolicyMaker [18], Keynote [17], REFEREE [24], and DL [57]) use credentials to describe specific delegation of trusts among keys and to bind public keys to authorizations. They therefore depart from the traditional separation between authentication and authorizations by granting authorizations directly to keys (bypassing identities). Trust management systems provide an interesting framework for reasoning about trust between unknown parties; however, assigning authorizations to keys, may result limiting and make authorization specifications difficult to manage. A promising direction exploiting digital certificates to regulate access control is represented by new authorization models making the access decision of whether or not a party may execute an access dependent on properties that the party may have, and can prove by presenting one or more certificates (authorization certificates in [18] being a specific kind of them). Besides a more complex authorization language and model, there is however a further complication arising in this new scenario, due to the fact that the access control paradigm is changing. On the one side, the server may not have all the information it needs in order to decide whether or not an access should be granted (and exploits certificates to take the decision). On the other side, however, the requestor may not know which certificates she needs to present to a (possibly just encountered) server in order to get access. Therefore, the server itself should, upon reception of the request, return the user with the information of what she should do (if possible) to get access. In other words the system cannot simply return a "yes/no" access decision anymore. Rather, it should return the information of the requisites that it requires be satisfied for the access to be allowed. The certificates mentioned above are

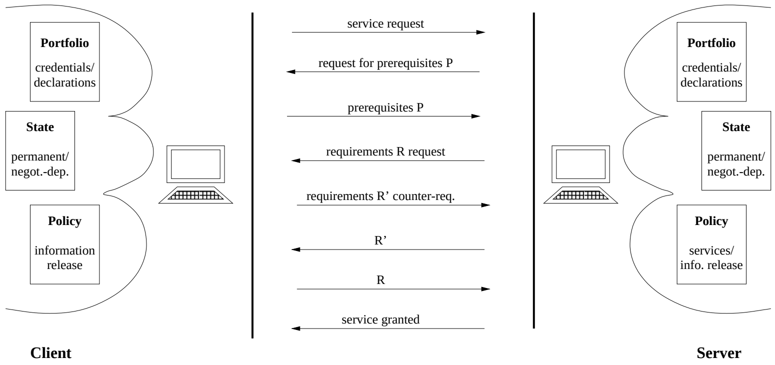

Fig. 23. Client/server interplay in [21]

one type of access requisites. In addition, other uncertified declarations (i.e., not signed by any authority) may be required. For instance, we may be requested our credit card number to perform an electronic purchase; we may be requested to fill in a profile when using public or semipublic services (e.g., browsing for flight schedules). The access control decision is therefore a more complex process and completing a service may require communicating information not related to the access itself, but related to additional restrictions on its execution, possibly introducing a form of negotiation [21, 72, 89]. Such information communication makes the picture even more complex, since it introduces two new protection requirements (in addition to the obvious need of protecting resources managed by the server from unauthorized or improper access):

Client portfolio protection: the client (requestor) may not be always willing to release information and digital certificates to other parties [65], and may therefore impose restrictions on their communication. For this purpose, a client may—like a server—require the counterpart to fulfill some requirements. For instance, a client may be willing to release a AAA membership number only to servers supplying a credential stating that the travel agent is approved by AAA.

Server's state protection: the server, when communicating requisites for access to a client, wants to be sure that possible sensitive information about its access control policy is not disclosed. For instance, a server may require a digitally signed guarantee to specific customers (who appear blacklisted for bad credit in some database it has access to); the server should simply ask this signed document, it should not tell the customer that she appears blacklisted.

The first proposals investigating the application of credential-based access control regulating access to a server, were made by Winslett et al. [82, 89]. Access control rules are expressed in a logic language and rules applicable to an access can be communicated by the server to clients. The work was also extended in [88, 93] investigating trust negotiation issues and strategies that a party can apply to select credentials to submit to the opponent party in a negotiation. In particular, [88] distinguishes between *eager* and *parsimonious* credential release strategies. Parties applying the first strategy turn over all their credentials if the release policy for them is satisfied, without waiting for the credentials to be requested. Parsimonious parties only release credentials upon explicit request by the server (avoiding unnecessary releases). Yu et al. [93] present a prudent negotiation strategy to the goal of establishing trust among parties, while avoiding disclosure of irrelevant credentials.

A credential-based access control is also presented by Bonatti and Samarati in [21]. They propose a uniform framework for regulating service access and information disclosure in an open, distributed network system like the Web. Like in previous proposals, access regulations are specified as logical rules, where some predicates are explicitly identified. Besides credentials, the proposal also allows to reason about declarations (i.e., unsigned statements) and user-profiles that the server can maintain and exploit for taking the access decision. Communication of requisites to be satisfied by the requestor is based on a filtering and renaming process applied on the server's policy, which exploits partial evaluation techniques in logic programs. The filtering process allows the server to communicate to the client the requisites for an access, without disclosing possible sensitive information on which the access decision is taken. The proposal allows also clients to control the release of their credentials, possibly making counter-requests to the server, and releasing certain credentials only if their counter-requests are satisfied (see Figure 23). Client-server interplay is limited to two interactions to allow clients to apply a parsimonious strategy (i.e., minimizing the set of information and credentials released) when deciding which set credentials/declarations release among possible alternative choices they may have.

While all these approaches assume access control rules to be expressed in logic form, often the people specifying the security policies are unfamiliar with logic based languages. An interesting aspect to be investigated concerns the definition of a language for expressing and exchanging policies based on a high level formulation that, while powerful, can be easily interchangeable and both human and machine readable. Insights in this respect can be taken from recent proposals expressing access control policies as XML documents [26, 27].

All the proposals above open new interesting directions in the access control area.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15:706–734, 1993.
2. R. Ahad, J. David, S. Gower, P. Lyngbaek, A. Marynowski, and E. Onuebge. Supporting access control in an object-oriented database language. In *Proc. of*

the Int. Conference on Extending Database Technology (EDBT), Vienna, Austria, 1992.
3. G. Ahn and R. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proc. of the fourth ACM Workshop on Role-based Access Control*, pages 43–54, Fairfax, VA, USA, October 1999.
4. A. Aho, J. Hoperoft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
5. J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic System Division/AFSC, Bedford, MA, October 1972.
6. Apache http server version 2.0. http://www.apache.org/docs-2.0/misc/tutorials.html.
7. V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.
8. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems*. McGraw-Hill, 1999.
9. Robert W. Baldwin. Naming and grouping privileges to simplify security management in large database. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 61–70, Oakland, CA, April 1990.
10. D.E. Bell. Secure computer systems: A refinement of the mathematical model. Technical Report ESD-TR-278, vol. 3, The Mitre Corp., Bedford, MA, 1973.
11. D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-278, vol. 4, The Mitre Corp., Bedford, MA, 1973.
12. D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-278, vol. 1, The Mitre Corp., Bedford, MA, 1973.
13. E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, September 1998.
14. E. Bertino, S. De Capitani di Vimercati, E. Ferrari, and P. Samarati. Exception-based information flow control in object-oriented systems. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):26–65, 1998.
15. E. Bertino, P. Samarati, and S. Jajodia. An extended authorization model for relational databases. *IEEE-TKDE*, 9(1):85–101, January-February 1997.
16. K.J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, The Mitre Corporation, Bedford, MA, April 1977.
17. M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Springer Verlag – LNCS State-of-the-Art series, 1998.
18. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996.
19. W.E. Boebert and C.T. Ferguson. A partial solution to the discretionary Trojan horse problem. In *Proc. of the* 8^{th} *Nat. Computer Security Conf.*, pages 141–144, Gaithersburg, MD, 1985.
20. P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.
21. P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.

22. D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, 1989.
23. S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.
24. Y-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.
25. D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, pages 184–194, Oakland, CA, May 1987.
26. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1–6):59–75, June 2000.
27. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Fine grained access control for SOAP e-services. In *Tenth International World Wide Web Conference*, Hong Kong, China, May 2001.
28. Database language SQL – part 2: Foundation (SQL/foundation). ISO International Standard, ISo/IEC 9075:1999, 1999.
29. C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.
30. S. Dawson, S. De Capitani di Vimercati, P. Lincoln, and P. Samarati. Minimal data upgrading to prevent inference and association attacks. In *Proc. of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, Philadelphia, CA, 1999.
31. D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
32. D.E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982.
33. D.E. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *Proc. of the 1985 IEEE Symposium on Security and Privacy*, pages 134–146, April 1985.
34. S. De Capitani di Vimercati, P. Samarati, and S. Jajodia. Hardware and software data security. In *Encyclopedia of Life Support Systems*. EOLSS publishers, 2001. To appear.
35. E.B. Fernandez, E. Gudes, and H. Song. A model for evaluation and administration of security in object-oriented databases. *IEEE Transaction on Knowledge and Data Engineering*, 6(2):275–292, 1994.
36. D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. of the 15th NIST-NCSC Naional Computer Security Conference*, pages 554–563, Baltimore, MD, October 1992.
37. R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9), September 1997.
38. T.D. Garvey and T.F. Lunt. Cover stories for database security. In C.E. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, North-Holland, 1992. Elsevier Science Publishers.
39. B. Gladman, C. Ellison, and N. Bohm. Digital signatures, certificates and electronic commerce. http://jya.com/bg/digsig.pdf.
40. J.A Goguen and J. Meseguer. Unwinding and inference control. In *Proc. of the 1984 Symposium on Research in Security and Privacy*, pages 75–86, 1984.

41. G.S. Graham and P.J. Denning. Protection – principles and practice. In AFIPS Press, editor, *Proc. Spring Jt. Computer Conference*, volume 40, pages 417–429, Montvale, N.J., 1972.
42. P.P. Griffiths and B.W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.
43. J.T. Haigh, R.C. O'Brien, and D.J. Thomsen. The LDV secure relational DBMS model. In S. Jajodia and C.E. Landwehr, editors, *Database Security, IV: Status and Prospects*, pages 265–279, North-Holland, 1991. Elsevier Science Publishers.
44. M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.
45. T. Jaeger and A. Prakash. Requirements of role-based access control for collaborative systems. In *Proc. of the first ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, USA, November 1995.
46. S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 76–85, Oakland, CA, 1990.
47. S. Jajodia and C. Meadows. Inference problems in multilevel secure database management systems. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security: An Integrated Collection of Essays*, pages 570–584. IEEE Computer Society Press, 1995.
48. S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible supporting for multiple access control policies. *ACM Transactions on Database Systems*, 2000. To appear.
49. S. Jajodia and R. Sandhu. Polyinstantiation for cover stories. In *Proc. of the Second European Symposium on Research in Computer Security*, pages 307–328, Toulouse, France, November 1992.
50. S. Jajodia and Ravi S. Sandhu. Toward a multilevel secure relational data model. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 50–59, Denver, CO, May 1991.
51. P.A. Karger. Limiting the damage potential of discretionary Trojan Horses. In *Proc. IEEE Symposium on Security and Privacy*, pages 32–37, Oakland, CA, 1987.
52. R. Kemmerer. Share resource matrix methodology: an approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, April 1983.
53. B.W. Lampson. Protection. In *5th Princeton Symposium on Information Science and Systems*, pages 437–443, 1971. Reprinted in *ACM Operating Systems Review* 8(1):18–24, 1974.
54. C.E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, 1981.
55. L.J. LaPadula and D.E. Bell. Secure computer systems: A mathematical model. Technical Report ESD-TR-278, vol. 2, The Mitre Corp., Bedford, MA, 1973.
56. G. Lawrence. The role of roles. *Computers and Security*, 12(1), 1993.
57. N. Li, B.N. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 27–42, Oakland, CA, 2000.
58. Teresa Lunt. Access control policies: Some unanswered questions. In *IEEE Computer Security Foundations Workshop II*, pages 227–245, Franconia, NH, June 1988.
59. T.F. Lunt. Aggregation and inference: Facts and fallacies. In *Proc. IEEE Symposium on Security and Privacy*, pages 102–109, Oakland, CA, 1989.

60. T.F. Lunt. Polyinstantiation: an inevitable part of a multilevel world. In *Proc. Of the IEEE Workshop on computer Security Foundations*, pages 236–238, Franconia, New Hampshire, June 1991.
61. T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990.
62. C.J. McCollum, J.R. Messing, and L. Notargiacomo. Beyond the pale of MAC and DAC - Defining new forms of access control. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, CA, 1990.
63. J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.
64. J. McLean. Security models. In *Encyclopedia of Software Engineering*. Wiley Press, 1994.
65. Communication of the ACM. Special issue on internet privacy. *CACM*, February 1999.
66. Oracle Corporation, Redwood City, CA. *Trusted Oracle7 Server Administration Guide, Version 7.0*, January 1993.
67. S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.
68. W.R. Polk and L.E. Bassham. Security issues in the database language SQL. Technical Report NIST special publication 800-8, Institute of Standards and Technology, 1993.
69. X. Qian and T.F. Lunt. A MAC policy framework for multilevel relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):1–14, February 1996.
70. F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM TODS*, 16(1):89–131, March 1991.
71. J. Richardson, P. Schwarz, and L. Cabrera. CACL: Efficient fine-grained protection for objects. In *Proceedings of OOPSLA*, 1992.
72. M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, CA, May 1996.
73. P. Samarati and S. Jajodia. Data security. In J.G. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1999.
74. R. Sandhu. On five definitions of data integrity. In *Proc. of the IFIP WG 11.3 Workshop on Database Security*, Lake Guntersville, Alabama, September 1993.
75. R. Sandhu and F. Chen. The multilevel relational (MLR) data model. *ACM Transactions on Information and System Security (TISSEC)*, 2000.
76. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. of the fifth ACM Workshop on Role-based Access Control*, pages 47–63, Berlin Germany, July 2000.
77. R. Sandhu and Q. Munawer. The ARBAC99 model for administration of roles. In *Proc. of the 15th Annual Computer Security Applications Conference*, Phoenix, Arizona, December 1999.
78. R. Sandhu and P. Samarati. Authentication, access control and intrusion detection. In A. Tucker, editor, *CRC Handbook of Computer Science and Engineering*, pages 1929–1948. CRC Press Inc., 1997.
79. Ravi S. Sandhu. Transaction control expressions for separation of duties. In *Fourth Annual Computer Security Application Conference*, pages 282–286, Orlando, FL, December 1988.

80. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.
81. R.S. Sandhu. The typed access matrix model. In *Proc. of 1992 IEEE Symposium on Security and Privacy*, pages 122–136, Oakland, CA, May 1992.
82. K. E. Seamons, W. Winsborough, and M. Winslett. Internet credential acceptance policies. In *Proceedings of the Workshop on Logic Programming for Internet Applications*, Leuven, Belgium, July 1997.
83. Security. http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html.
84. H. Shen and P. Dewan. Access control for collaborative environments. In *Proc. Int. Conf. on Computer Supported Cooperative Work*, pages 51–58, November 1992.
85. A. Stoughton. Access flow: A protection model which integrates access control and information flow. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 9–18, Oakland, CA, 1981.
86. R.C. Summers. *Secure Computing: Threats and Safeguard*. McGraw-Hill, 1997.
87. K.G. Walter, W.F. Ogden, W.C. Rounds, F.T. Bradshaw, S.R. Ames, and D.G. Sumaway. Primitive models for computer security. Technical Report TR ESD-TR-4-117, Case Western Reserve University, 1974.
88. W. Winsborough, K. E. Seamons, and V. Jones. Automated trust negotiation. In *Proc. of the DARPA Information Survivability Conf. & Exposition*, Hilton Head Island, SC, USA, January 25-27 2000. IEEE-CS.
89. M. Winslett, N. Ching, V. Jones, and I. Slepchin. Assuring security and privacy for digital library transactions on the web: Client and server security policies. In *Proceedings of ADL '97 — Forum on Research and Tech. Advances in Digital Libraries*, Washington, DC, May 1997.
90. M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, December 1994.
91. T.Y.C. Woo and S.S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.
92. J. Wray. An analysis of covert timing channels. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1991.
93. T. Yu, X. Ma, and M. Winslett. An efficient complete strategy for automated trust negotiation over the internet. In *Proceedings of 7th ACM Computer and Communication Security*, Athens, Greece, November 2000.