Access Control for Databases: Concepts and Systems

By Elisa Bertino, Gabriel Ghinita
and Ashish Kamra

Contents

1 Introduction 3

1.1 An Historical Perspective 6
1.2 Recent Research Directions 9
1.3 Organization of the Monograph 11

2 Background 13

2.1 Access Control Models 13
2.2 Cryptographic Preliminaries 34
2.3 Summary 37

3 Foundations of Access Control for Relational Database Systems 38

3.1 The System R Access Control Model 38
3.2 Content-based Access Control 43
3.3 Mandatory Access Control Models 46
3.4 Summary 50

4 Case Studies 51

4.1 SQL Server 2008 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Oracle Virtual Private Database</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Labeled Oracle</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>Fine-Grained Access Control Models and Mechanisms</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Fine Grained Access Control through Query Rewriting</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>SQL Language Extensions for Fine Grained Access Control</td>
<td>77</td>
</tr>
<tr>
<td>5.3</td>
<td>Fine Grained Access Control with Authorization Views</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>PSAC: A Privilege State Based Access Control System</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Motivation</td>
<td>85</td>
</tr>
<tr>
<td>6.2</td>
<td>Design and Implementation</td>
<td>87</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>Protection from Insider Threats and Separation of Duties</td>
<td>100</td>
</tr>
<tr>
<td>7.1</td>
<td>Oracle Database Vault</td>
<td>101</td>
</tr>
<tr>
<td>7.2</td>
<td>Joint-Threshold Administration</td>
<td>103</td>
</tr>
<tr>
<td>7.3</td>
<td>Summary</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>Access Control for Object Databases, XML Data and Novel Applications</td>
<td>111</td>
</tr>
<tr>
<td>8.1</td>
<td>Requirements</td>
<td>112</td>
</tr>
<tr>
<td>8.2</td>
<td>The Orion Authorization Model</td>
<td>113</td>
</tr>
<tr>
<td>8.3</td>
<td>MAC Models for Object Databases</td>
<td>120</td>
</tr>
<tr>
<td>8.4</td>
<td>Access Control Models for XML Data</td>
<td>120</td>
</tr>
<tr>
<td>8.5</td>
<td>Access Control Models for Geographical Data</td>
<td>122</td>
</tr>
<tr>
<td>8.6</td>
<td>Access Control Models for Digital Libraries</td>
<td>124</td>
</tr>
<tr>
<td>8.7</td>
<td>Summary</td>
<td>125</td>
</tr>
</tbody>
</table>
9 Encryption-based Access Control

9.1 Encryption-based Access Control for XML Documents

9.2 Privacy-preserving Access Control Mechanisms

9.3 Summary

10 Concluding Remarks and Research Directions

References
Access Control for Databases: Concepts and Systems

Elisa Bertino¹, Gabriel Ghinita² and Ashish Kamra³

¹ CS Department, Purdue University, West Lafayette, IN, 47907, USA, bertino@cs.purdue.edu
² CS Department, Purdue University, West Lafayette, IN, 47907, USA, gghinita@cs.purdue.edu
³ ECE Department, Purdue University, West Lafayette, IN, 47907, USA, akamra@purdue.edu

Abstract

As organizations depend on, possibly distributed, information systems for operational, decisional and strategic activities, they are vulnerable to security breaches leading to data theft and unauthorized disclosures even as they gain productivity and efficiency advantages. Though several techniques, such as encryption and digital signatures, are available to protect data when transmitted across sites, a truly comprehensive approach for data protection must include mechanisms for enforcing access control policies based on data contents, subject qualifications and characteristics, and other relevant contextual information, such as time. It is well understood today that the semantics of data must be taken into account in order to specify effective access control policies. To address such requirements, over the years the database security research community has developed a number of access control techniques and
mechanisms that are specific to database systems. In this monograph, we present a comprehensive state of the art about models, systems and approaches proposed for specifying and enforcing access control policies in database management systems. In addition to surveying the foundational work in the area of access control for database systems, we present extensive case studies covering advanced features of current database management systems, such as the support for fine-grained and context-based access control, the support for mandatory access control, and approaches for protecting the data from insider threats. The monograph also covers novel approaches, based on cryptographic techniques, to enforce access control and surveys access control models for object-databases and XML data. For the reader not familiar with basic notions concerning access control and cryptography, we include a tutorial presentation on these notions. Finally, the monograph concludes with a discussion on current challenges for database access control and security, and preliminary approaches addressing some of these challenges.
Today all organizations rely on database systems as the key data management technology for a large variety of tasks, ranging from day-to-day operations to critical decision making. Such widespread use of database systems implies that security breaches to these systems affect not only a single user or application, but also may have disastrous consequences on the entire organization. The recent rapid proliferation of Web-based applications and information systems, and recent trends such as cloud computing and outsourced data management, has further increased the exposure of database systems and, thus, data protection is more crucial than ever. Conventional perimeter-oriented defenses, like firewalls, are inadequate in today’s interconnected world and are unable to offer the fine-grained protection required for selective and secure data sharing among multiple users and applications. Security techniques offered by operating systems may offer some protection at the file system level; however the protected objects are typically files and directories and these protection units are too coarse with respect to the logical protection units, such as records, that are required in database systems. It is also important to appreciate that data need to be protected not only from external threats, but also from insider threats [19].
As discussed by Bertino and Sandhu [19], data security breaches are typically classified as unauthorized data observation, improper data modification, and data unavailability. Unauthorized data observation results in the disclosure of information to subjects not entitled to gain access to the information. All organizations, ranging from governmental and military organizations to social and commercial organizations, may suffer losses from both financial and human points of view as a consequence of unauthorized data observation. The unauthorized disclosure of personally identifiable data may result in privacy breaches, that may lead to identity theft and other serious consequences for the individuals. Improper data modifications, either intentional or unintentional, result in incorrect data. Any use of incorrect data may also result in heavy losses for organizations. When data are unavailable, information crucial for the proper functioning of an organization is not readily available when needed. Thus, a complete solution to data protection must meet three key requirements: (1) secrecy or confidentiality — it refers to the protection of data against unauthorized disclosures; (2) integrity — it refers to the prevention of improper data modifications; and (3) availability — it refers to the prevention and recovery from hardware and software errors and from malicious data access denials making the database system unavailable. These three requirements arise practically in all applications. Consider a database storing medical information about patients of a hospital. It is important that patient records not be released to unauthorized subjects, that records be modified only by the subjects who are properly authorized and their accuracy be assured, and that patient records be readily available to doctors in charge especially in emergency situations.

Securing data is a challenging task. It is ensured collectively by various components of a database management system (DBMS) and may also require components external to the DBMS, such as secure co-processors [1].

A key component for assuring data protection is represented by the access control mechanism. When a subject attempts to access some

---

1 The term ‘subject’ refers to any active entity which tries to access the protected resources in a system. A subject can be an end-user, a process, or an application program, or an organizational role.
data, the access control mechanism checks whether or not the subject has the authorization to perform the action on the data. Authorizations are granted to subjects according to the access control policies of the organization. Confidentiality can be further enhanced by the use of encryption techniques, applied to data when being stored on secondary storage or transmitted on a network or managed by third parties, as in the case of outsourced database management [2].

Integrity is jointly ensured by the access control mechanism and by semantic integrity constraints. Whenever a subject tries to modify some data, the access control mechanism verifies that the subject is authorized to modify the data, and the semantic integrity subsystem verifies that the updated data are correct with respect to a set of semantic conditions, referred to as integrity constraints. To protect data from being tampered with while in transit on a network, data can be digitally signed. Finally, the recovery subsystem and the concurrency control mechanism ensure that data are available and correct despite hardware and software failures and accesses from concurrent application programs. Data availability, especially for databases that are available on the Web, can be further strengthened by the use of techniques protecting against denial-of-service attacks.

As the focus of this monograph is on access control models and mechanisms, we do not cover transaction management or semantic integrity. We refer the reader to [40] for an extensive discussion on transaction models, recovery and concurrency control, and to any database textbook for details on semantic integrity. It is important to notice that because the access control mechanism intercepts every access to protected resources, it can also be used to create profiles of accesses by subjects and thus be used in the context of anomaly detection [49] and insider threat protection. Also as current access control systems, like the ones based on XACML [67], are able to take into account a large variety of information including meta-data associated with the data and context information, they can be used for a variety of goals. An example is to grant access to data based on the confidence level of data [30]; in such case, policies specify which is the minimum level of confidence that certain data must have for a given user to access these data for certain tasks. Such policies thus prevent
the use of incorrect or invalid data for critical tasks. In this example, the metadata used for access control decisions are the confidence levels associated with the data and the goal of the access control policies is not to protect the confidentiality or integrity of the data, but it is to control that users use data that are “good enough” for the tasks they have to perform.

It is also important to note that an access control mechanism must rely for its proper functioning on some authentication mechanism. Such a mechanism identifies users and confirms their identities. Moreover, data may be encrypted when transmitted over networks and when stored on secondary storage. Authentication and encryption techniques are extensively discussed in the current literature on computer network security and we refer the reader to [50] for details on such topics. We will, however, discuss the use of encryption techniques as an approach to implementing access control. We do not attempt to be exhaustive, but try to articulate the rationale for the approaches we believe to be promising.

In the rest of the section, we first present a short historical overview of access control in database systems based on the overview by Bertino and Sandhu [19] (Section 1.1), and then present a road map for the rest of the monograph (Section 1.2).

1.1 An Historical Perspective

Early research proposals in the area of access control systems for DBMSs focused on the development of two different classes of models, based on the discretionary access control (DAC) policy and on the mandatory access control (MAC) policy, respectively. The discretionary access control policy allows subjects to grant authorizations on the data for which they have administration authorization to other subjects. By contrast, the mandatory access control policy regulates accesses to data by subjects on the basis of predefined classifications of subjects and data. Under such a policy even the creator of a data object, like a relation, is not able to grant at its own discretion access authorizations to other subjects. These early access control systems were developed in the framework of relational database systems. The
relational data model, being a declarative high-level model, made it possible to develop declarative languages for the specification of access control policies. The earlier access control models, and the discretionary models in particular, introduced some important principles [36] that set apart access control models for database systems from access control models adopted by operating systems and file systems. The first principle is that access control models for databases should be expressed in terms of the logical data model; thus authorizations for a relational database should be expressed in terms of the logical constructs of the relational data model, that is, relations, relation attributes, and tuples. The second principle is that for databases, in addition to name-based access control, whereby the protected objects are denoted in authorizations by their names, content-based access control has to be supported. Content-based access control allows the system to determine whether to give or deny access to a data item based on the contents of the data item. The development of content-based access control models, which are, in general, based on the specification of conditions against data contents, was made easy in relational databases by the availability of declarative query languages, such as SQL.

In the area of discretionary access control models for relational database systems, the most important early contribution was the development of the System R access control model by Griffith and Wade [35, 41], from which the access control models of current commercial relational DBMSs have been derived. Key features of this model include the concept of decentralized authorization administration, dynamic granting and revocation of authorizations, and the use of views for content-based access control. Also, the initial format of the authorization grant and revoke commands, that are today part of the SQL standard, was developed as part of this model. Subsequent access control models have extended the System R model with a variety of features, such as negative authorization [18], role-based authorization [77], temporal authorization [6], and context-aware authorization [70].

Discretionary access control mechanisms have, however, a major drawback in that they are not able to control how information is propagated and used once it has been accessed by subjects authorized to do so. This weakness makes discretionary access controls vulnerable to
malicious attacks, such as Trojan Horses. A Trojan Horse is a program with an apparent or actually useful function, which contains some hidden functions exploiting the legitimate authorizations of the invoking process. Sophisticated Trojan Horses may leak information by means of covert channels, enabling illegal access to data. A covert channel is any component or feature of a system that is misused to encode or represent information for unauthorized transmission, without violating the stated access control policy. A large variety of components or features can be exploited to establish covert channels, including the system clock, operating system interprocess communication primitives, error messages, the existence of particular file names, the concurrency control mechanism, and so forth. The goal of mandatory access control and multilevel database systems was to address such problems through the development of access control models based on data and subject classification, some of which were also incorporated in commercial products. Early mandatory access control models were mainly developed for military applications and were very rigid and suited, at best, for closed and controlled environments. There was considerable discussion in the security community concerning how to eliminate covert channels while maintaining the essential properties of the relational model. The concept of polyinstantiation, that is, the presence of multiple copies with different security levels of a same tuple in a relation, was developed and investigated in this period [79]. Because of the lack of applications and commercial success, companies developing multilevel DBMSs discontinued their production in the early nineties. Covert channels were also widely investigated with considerable focus on the concurrency control mechanisms that, by synchronizing transactions running at different security levels, would introduce an obvious covert channel. However, solutions developed in the research arena to the covert channel problem were not incorporated into commercial products. Interestingly, however, at the beginning of the 2000s, strong security requirements arising in a number of civilian applications have driven a “multilevel security reprise” [80]. Companies have thus reintroduced such systems. The most notable of such systems is Labeled Oracle, a multilevel relational DBMS by Oracle, which has much more flexibility in comparison to earlier multilevel secure DBMSs.
These early approaches to access control have then been extended in the context of advanced DBMSs, such as object-oriented DBMSs and object-relational DBMSs, and other advanced data management systems and applications, such as XML repositories, digital libraries and multimedia data, data warehousing systems, and workflow systems. Most of these systems are characterized by data models that are more expressive than the relational model; typically, these extended models include modeling notions such as inheritance hierarchies, aggregation, and methods. An important requirement for those applications concerns the fact that not only the data need to be protected, but also the database schema may contain sensitive information and, thus, accesses to the schema need to be filtered according to the access control policies. Even though early relational DBMSs did not support access control to the schema information, today several products support such feature. In this respect, access control policies may also need to be protected because they may reveal sensitive information. As such, one may need to define access control policies for objects which are not user data, rather they are other access control policies. Another relevant characteristic of advanced applications is that they often deal with multimedia data, for which the automatic interpretation of contents is much more difficult, and they are, in most cases, accessed by a variety of users external to the system boundaries, such as through Web interfaces. As a consequence both discretionary and mandatory access control models developed for relational DBMSs had to be properly extended to deal with additional modeling concepts. Also, these models often need to rely on metadata information in order to support content-based access control for multimedia data and to support credential-based access control policies to deal with external users. Efforts in this direction include the development of comprehensive access control models for XML [9, 67].

1.2 Recent Research Directions

More recent research directions in the area of access control for database systems have been driven by legal requirements as well as by technology developments. A first research direction is related to privacy-preserving
Introduction

techniques for databases, an area recently investigated to a considerable extent. Privacy legislation, such as the early Federal Act [26] of 1974, and the more recent Health Insurance Portability and Accountability Act of 1996 (HIPAA) [43] and the Children’s Online Privacy Protection Act (COPPA) [25], require organizations to deploy adequate fine-grained access control mechanisms able to control access at the finest granularity possible, that is, at the cell level, and also to take into account additional information, such as the data usage purpose and the data retention period [21]. Privacy is also motivating the development of oblivious access control, which is crucial when access control decisions are based by also taking into account (possibly sensitive) information about the subjects seeking accesses to the data. A requirement is thus to be able to enforce access control without disclosing such subject information to the party owning the protected data [22, 81]. A second relevant recent research direction is motivated by the trend of considering databases as a service that can be outsourced to external companies [46]. As outsourced data are encrypted when stored at the service provider, subjects authorized to access the data need to receive the proper keys for decrypting the data. Approaches are thus needed in this context for fine-grained encryption, by which different portions of the data are encrypted with different encryption keys and subjects receive only the keys corresponding to the portions they are entitled to access. A possible approach has been defined in the context of third-party publishing systems for XML data [23]. A third relevant direction is driven by the problem of insider threats, that is, individuals who misuse the data to which they have access to. Protecting from such threats require sophisticated techniques, such as anomaly detection tools able to build profiles of normal data accesses and detect accesses that are anomalous with respect to these profiles. A particular crucial problem in this context is represented by malicious database administrators (DBAs), as a DBA has typically access to the entire database he/she administers. To address this problem solutions have been proposed including the segregation of DBAs from user data, as in the case of the Oracle Database Vault product, and techniques for joint administration of critical database objects.
1.3 Organization of the Monograph

We begin with a brief introduction to relevant background notions concerning access control models and mechanisms, and cryptography (Section 2). We then summarize the foundations of access control systems for relational database systems, including the access system developed as part of System R [41] and its extensions (Section 3). As these foundations have been covered in a previous survey by Bertino and Sandhu [19], we keep the presentation very short here and refer the reader to such survey for details. The presentation on the foundations is complemented by some case studies covering access control models and mechanisms supported by current DBMSs (Section 4). In particular, we discuss the Oracle Virtual Private Database mechanism which is an interesting approach to context-based access control and the access control mechanism of SQL Server which has many interesting capabilities, such as the support for roles and negative authorizations. We then cover approaches to fine-grained access control. These approaches allow one to associate access permissions with fine-grained elements within a relation, such as a single tuple or even a single cell.

Fig. 1.1 Topics covered in the area of access control for the relational data model.
Fine-grained access control is today a key requirement for information privacy. We then cover more innovative approaches focusing on state-based access control (Section 6), the use of access control mechanisms for protection from insider threats (Section 7), and access control systems for object databases and XML data (Section 8). It is important to remark that approaches and notions developed in the context of object databases, such as those developed for the Orion object-oriented DBMS [74], have been applied to relational DBMSs and also to operating systems. Examples of those approaches and notions include hierarchical authorizations, positive and negative authorizations, and schema protections. We then conclude the paper by discussing the use of cryptography to enforce access control (Section 9), and recent research trends (Section 10). Figures 1.1 and 1.2 provide a high-level description of the relationships among the topics covered in the paper for the relational data model and for more complex data models and selected novel applications, respectively.
In this section, we provide background information needed for the discussion in the rest of the monograph. We begin by first covering the basic notions concerning access control and survey the most important access control models, based on the survey by Bertino and Crampton [12]. We then describe the cryptographic notions that are needed for the discussion of cryptography-based approaches to access control.

2.1 Access Control Models

In a computer system, users typically wish to read and write data objects, such as files and relations, browse directories and data repositories, and execute programs. In multi-user systems, different users will be authorized to access different resources. In other words, there will generally be a security policy that determines the resources to which each user has access. An access control mechanism (also known as a reference monitor or authorization service) is a computer system that enforces an access control policy. Any computer system that offers any level of security typically ensures that the access control mechanism intercepts all user requests to access resources in order to ensure
that these user requests are properly authorized before the user gains access to the requested resource. Most access control policies directly or indirectly specify the set of authorized requests. The default approach adopted by most access control policies is that any request which is not authorized by the access control policy is denied; such default approach is referred to as closed policy. Other approaches are however possible. In the open policy, for example, the access control policies specify directly or indirectly all requests that are to be denied; under such a policy any request that is not denied by the access control policy is allowed by default.

Figure 2.1 illustrates a generic architecture for an access control mechanism. The very nature of access control suggests that there is an active subject requiring access to a passive object to perform some specific access operation [53]. A reference monitor permits or denies access. The reference monitor consults the set of access control policies, and information about the subject and the objects in order to make the access control decision. Subjects not only include users, but also application programs and processes running on behalf of some users. Typical information used about subjects and objects in access control is their system identifiers. Advanced access control models however extend this basic information with a large variety of information about the subjects and objects, thus resulting in the so-called attribute-based access control models, and with contextual information, such as time and location. We refer to an access control system as a system comprising of a reference monitor, and all information required to take
access control decisions, such as the access control policies. As typically
the access control system is part of some other system, as it is the case
of access control in DBMSs and operating systems, the access control
decision is typically returned to the latter system, which can then for-
ward the decision to the user or application. An access control model
provides a method for encoding an access control policy and states the
conditions that must be satisfied for an access request to be granted.
In other words, it provides the conceptual model to be implemented by
the access control mechanism. The conditions that determine whether
a request is authorized may be expressed in many different ways. One
approach is to simply require the presence of an authorization that
permits access. Another approach is to require that the subject and
the object have certain relationships, as in the case of the mandatory
access control policies.

In what follows we survey some of the important theoretical con-
cepts and models for access control. These concepts and models are
used as the framework for the development of access control systems
in DBMS and also in operating systems.

2.1.1 The Protection Matrix

The protection matrix has been the first theoretical access control
model [42]. It is an abstract representation of a set of access control
policies as it specifies the access requests that are authorized. A pro-
tection matrix is arranged as a two-dimensional array, with each row
labeled by a subject and each column labeled by an object. A matrix
entry in the row labeled \( s \) and column labeled \( o \) specifies the authorized
actions (also called access rights) for \( s \) with respect to \( o \). For exam-
ple, a matrix entry for \( s \) and \( o \) that contains read specifies that \( s \) is
allowed to read object \( o \). In other words, the protection matrix encodes
triples of the form subject–object–action (referred to as authorizations);
object–action pairs are often referred to as permissions or capabilities.
The protection matrix also includes operations for its manipulations,
namely for entering and deleting actions from the matrix entries, and
for adding and deleting subjects and objects. These operations thus
represent a core set of access control administration operations. The
Background

The protection matrix has proved to be a powerful abstraction for the design of access control mechanisms. It also has been the basis of some important theoretical questions, such as whether the execution of a sequence of manipulation operations may result in giving some permissions that do not comply with the intended access control policies. Such question, known as safety property, is in general undecidable; however it is decidable for restricted cases.

We may use a protection matrix in order to implement an access control system. Such a system would receive access requests and make a decision as to whether the requests should be granted. A request would only be granted if it is authorized by the protection matrix. In order to use the protection matrix, it would be necessary to know the following: the identifier of the subject making the request, the identifier of the object that the subject wishes to access, and the type of action that the subject wishes to execute on the object. The identifier of the subject is typically provided as part of the request or is directly determined by using control information associated with the transaction or process running on behalf of the subject. In most current systems, such identifier is a user account associated with the user that caused the transaction or process to be created. The identifier of the object is specified by the subject as part of the request. For example in a relational database, where relations are protected objects, the identifier of a relation is the relation name which is extracted from the SQL commands issued by application programs. Finally, the type of requested actions is usually represented as an access mask, which is a binary string of \( n \) bits, where \( n \) is the number of action types supported by the system. If a particular bit is set in the access mask, then the corresponding action is being requested. Given a user account identifier \( s \), an object identifier \( o \) and an access mask, the access control system would simply need to find the matrix entry corresponding to \( s \) and \( o \) and compare the access mask in the request with the one in the matrix entry. If every bit that is set in the requested mask is also set in the matrix entry, then every access that has been requested is authorized by the matrix and the access may be granted.

However, an access control system rarely adopts the protection matrix for storing authorization information. This is because in a large
computer system with many subjects and objects, the memory requirements for such a data structure would be prohibitively large. Moreover, many entries in the matrix may be empty, meaning that large amounts of memory allocated for the storage of the protection matrix would remain unused. Clearly such an approach is not viable. To address the problem of efficient implementation of the protection matrix, an alternative approach is to use a set of *access control lists* or *capability lists*. These structures have the feature that only relevant matrix entries are stored, with empty matrix entries being ignored.

An access control list is associated with an object and consists of a number of entries defining the rights assigned to each subject for that object. Conceptually, an access control list is a list of access control entries. Each entry in the list identifies a subject and a set of access rights. In other words, each access control entry in an access control list for object $o$ specifies the actions that the subject identified in the entry can execute on $o$. In order to implement an access control system based on access control lists, the same information is required as for a protection matrix implementation. In order to check whether a request should be granted, the access control system first finds the access control list for the requested object. In relational databases, such lists are stored in the system catalogs and are thus retrieved from these catalogs using the same access mechanisms used for retrieving data from user relations. It then checks each access control entry to see if it refers to the requesting subject. If it does, then the requested rights are compared with the rights in the access control entry. Access is granted if every requested right is in the access control entry and denied otherwise. In contrast, a capability list is associated with a subject. Conceptually, a capability list is a list of permissions, each permission identifying an object and the rights that have been assigned to the subject for that object. In other words, each permission in a capability list for a subject specifies how that subject may interact with the object specified in the permission.

### 2.1.2 Groups

In a large user population, many users typically share certain characteristics. In a commercial setting, perhaps the most obvious characteristic
would be job description or function. All users with the same job description are likely to have similar access to certain objects. Clearly, it would be laborious to create the same matrix entries (or access control entries) to these objects for each of these users. Hence, many access control systems include the ability to specify groups. The idea is that certain permissions can be assigned with a single command to a group. Each user who is a member of the group implicitly receives the permissions assigned to the group. A user may be associated with a number of different groups, and in more advanced group models, a group can be a member of another group. Typically, when a user logs on to a computer system, he is associated with identifiers both for his own user account as well as identifiers for any group to which he belongs. Groups thus represent a first important mechanism to reduce the overhead of the access control administration, that is, the execution of actions to grant and revoke permissions from users. It is important to notice that the notion of group can also be used for objects. Object groups are particularly convenient when based on the structural organization of objects in the system. An example of such structural organization is a file directory; a directory defines a set grouping all files in the directory. If a user has the same permissions on all files in a directory, an approach to reducing the administration costs would be to assign the permission to the directory and automatically propagate the permission to all files in the directory. Access control models supporting the notion of groups not only for users but also for objects have been defined for object-oriented database systems [74] that we discuss in Section 8.

2.1.3 Negative Authorizations

In recent years, several access control mechanisms have been proposed supporting negative authorizations, which explicitly prohibit certain actions. Such authorizations are particularly useful for enforcing exceptions to a more general policy. For example, suppose that we wish to grant access to a set of objects to many different subjects. Clearly, the most convenient way of doing this is to create a group for those subjects and then authorize the group to access those objects. However, if there is one member $s$ of the group $g$ who should not be allowed
access to one particular object \( o \), it is rather cumbersome to enforce this requirement without negative authorizations. The only option in this case is to remove \( s \) from \( g \) and then grant \( s \) access to all the objects to which the members of \( g \) have access except \( o \). A less burdensome way of implementing this requirement is to keep \( s \) in the group and simply prohibit access by \( s \) to \( o \). In Windows 2000, for example, it is possible to include negative access control entries in access control lists. XACML, the recent standard for XML-based authorization policy specification and enforcement [67], also supports negative authorizations.

The use of negative authorizations, however, is not without any problem. In particular, negative authorizations give rise to conflicts with conventional “positive” authorizations. Conflicts arise when a positive and negative authorization may exist for the same request. In the example discussed in the previous paragraph, \( s \) would not only have a positive authorization to access \( o \) from his membership of \( g \), but also a negative authorization explicitly denying him access to \( o \). The intention in this case is that access should be denied, but how can the access control mechanism determine that this is the intention? In order to address conflicts, policy conflict resolution mechanisms are required. The most obvious and widely used mechanism is to insist that a negative authorization always takes precedence over a positive one. This is known as the “deny-overrides” algorithm in XACML. There are other possibilities, such as “permit-overrides” and “first-applicable”, the latter assumes that authorization rules are processed in a particular order and that the first relevant authorization is to be used. Windows 2000 has a hybrid approach, which groups access control entries in a particular order and implements what might be called a “deny-overrides-if-first-applicable” algorithm. This ordering of access control entries is first determined by the creator of the access control entry, with entries created by the creator of the object (rather than ones inherited from the object’s container) taking precedence. Within each such group of entries, negative entries precede positive entries. Other possible approaches to solving conflicts are based on partial ordering defined on the set of objects and the set of subjects in the system [17].
2.1.4 Mandatory Access Control

We now discuss a very different approach to the use of authorizations, in which access to resources is controlled based on the respective attributes of the subject and object. Such approach, known as mandatory access control, was motivated by the need of providing stronger security and in particular to prevent unauthorized accesses to protected resources by the use of Trojan Horses. A Trojan Horse is a piece of code embedded into an application program. When the program is running on behalf of a user, the Trojan Horse exploits the authorizations of this user in order to gain access to protected data and transfer the data into some other objects accessible to users not authorized to access the protected data. Discretionary access control models, like those based on the access control lists, are unable to protect against Trojan Horses in that these models only control whether each single access is authorized; however, they do not control where the data flow once they have been accessed.

Research concerning mandatory access control was carried out for military applications in the late 1960s. The main concern of these research efforts was to ensure the confidentiality of sensitive electronic data. The initial approaches were defined so as to mimic paper-based systems in which documents are stamped with labels such as “Confidential” and “Top Secret”, and are filed securely according to their classification. A user is only allowed access to a document if his/her security clearance is as high as that of the document. Two important models from this period include the lattice-based model for information flow [33] and the Bell–LaPadula model [4]. The action of accessing an object can be considered as starting an information flow. In particular, reading an object causes information to flow from the object to the subject, while the flow is in the opposite direction if the subject writes to the object. An information flow policy thus specifies which information flows are authorized. As one might expect, an information flow policy for confidentiality requires that information from highly sensitive data objects cannot flow into data objects that are not sensitive and thus are less protected. As an example, an unclassified user cannot read top secret data. Such example shows that the allowed accesses are based
on comparing a certain specific property of subjects and data objects. This property, referred to as *access class*, is associated with each subject and object. The value of the access class of a data object typically indicates the sensitivity of the data objects, whereas the value of the access class of a subject indicates how much the subject can be trusted not to release sensitive data. In what follows, we describe in detail the Bell–LaPadula (BLP, for short) model which is perhaps the most well known of all access control models and has had a significant influence on the development of research into access control.

The elements of the BLP model are as follows:

- **Objects**: the passive entities containing information to be protected. Objects are assigned *sensitivity* levels.
- **Subjects**: the active entities requiring accesses to objects (users, processes). Subjects are assigned *clearance* levels and they can operate at a level up to and including their clearance levels.
- **Access modes**: the types of operations performed by subjects on objects, namely: (i) read: reading operation; (ii) append: modification operation; (iii) write: both reading and modification.

The clearance levels and the sensitivity levels take their values from the set of *access classes*. Each access class consists of two components:

- **Security level**: the security level is an element from a totally ordered set. An example of such a set is \{Top Secret (TS), Secret (S), Confidential (C), Unclassified (U)\} where $TS > S > C > U$.
- **Category set**: the category set is a set of elements specific to the application areas to which the data refer to. An example of such set, in a military application domain, is \{Army, Navy, AirForce, Nuclear\}.

The set of the access classes is partially ordered according to a relation called *dominance relation*. Let $\mathcal{L}$ and $\mathcal{C}$ be the set of security levels and categories, respectively. Let $c_i = (L_i, SC_i)$ and $c_k = (L_k, SC_k)$, with $L_i, L_k \in \mathcal{L}$ and $SC_i, SC_k \subseteq \mathcal{C}$, be two access classes. We say that $c_i$
Background
dominates $c_k$, denoted as $c_i \geq c_k$, if the following conditions hold:

- $L_i \geq L_k$, that is, the security level of $c_i$ is greater than or equal to the security level of $c_k$;
- $SC_k \subseteq SC_i$, that is, the category set of $c_i$ includes the category set of $c_k$.

$c_i$ and $c_k$ are said incomparable, denoted as $c_i <> c_k$ if neither $c_i \geq c_k$ nor $c_k \geq c_i$ holds.

As an example consider the following access classes:

- $c_1 = (TS, \{\text{Nuclear, Army}\})$
- $c_2 = (TS, \{\text{Nuclear}\})$
- $c_3 = (C, \{\text{Army}\})$

We have that $c_1 \geq c_2$ and $c_2 <> c_3$.

A second important element of the BLP model, and a key contribution of this model, is the notion of protection state (state, for short), that is, a snapshot of all security relevant information that is subject to change. In the BLP model the state is defined and is described by the pair $(A, lf)$ where:

- $A$ is the set of current accesses, that is, triples of the form $(s, o, m)$ denoting that subject $s$ is exercising access $m$ on object $o$. For example the triple $(\text{Bob}, o_1, \text{read})$ denotes that subject Bob is reading object $o_1$.
- $lf$ is the level function, that is, a function associating with each element in the system its access class. Let $\mathcal{O}$ be the set of objects, $\mathcal{S}$ the set of subjects, and $\mathcal{C}$ the set of access classes, the function is defined as $lf: \mathcal{O} \cup \mathcal{S} \rightarrow \mathcal{C}$.

In other words, the state is the set of those requests that have been granted and represents the set of objects currently in use by subjects. Based on the notion of state, the BLP model includes two access control rules assuring that all possible states into which the system may transition are secure. These access control rules are as follows:

- Simple security property (no-read-up): a given state $(A, lf)$ satisfies the simple security property if for each element
2.1 Access Control Models

\( a = (s, o, m) \in A \) one of the following conditions holds:

1. \( m = \text{append} \)
2. \( m = \text{read} \) or \( m = \text{write} \), and \( lf(s) \geq lf(o) \)

For example, a subject with access class \((C, \{\text{Army}\})\) is not allowed to read data objects with access classes \((C, \{\text{Navy}, \text{AirForce}\})\) or \((U, \{\text{AirForce}\})\).

The simple security thus prevents subjects from reading data with access classes dominating or incomparable with respect with the subject access class. It ensures that subjects have access only to information for which they have the necessary access class.

- Star (\(*\)−) property (no-write-down): a given state \((A, lf)\) satisfies the \(*\)−property if for each element \(a = (s, o, m) \in A\) one of the following conditions holds:
  1. \( m = \text{read} \)
  2. \( m = \text{append} \) and \( lf(o) \geq lf(s) \)
  3. \( m = \text{write} \) and \( lf(o) = lf(s) \)

For example a subject with access class \((C, \{\text{Army, Nuclear}\})\) is not allowed to append data into data objects with access class \((U, \{\text{Army, Nuclear}\})\).

The \(*\)−property has been defined to prevent information flow into objects with lower-level access classes or incomparable classes by mean of append operations.

The key points of the above properties can be summarized as follows. A subject has read access to an object if its access class dominates the access class of the object. A subject has append access to an object if its access class is dominated by that of the object. For a system to be secure both properties must be verified by any system state.

In many practical situations, however, the two properties, especially the \(*\)−property, are too restrictive. For example a trusted user may be allowed to access some sensitive data and, perhaps after sanitizing it, to transfer it into some unclassified data object. To address this problem, the model provides a mechanism by which each subject has a \textit{maximum}
access class and a current access class. A subject may change its access class; however its current access class must at any time be dominated by the maximum access class. We will discuss in Section 3 the implications of applying this model to relational databases and in Section 4 we will survey some elements of the mandatory access control model supported by the Oracle product.

2.1.5 Role-based Access Control

A long-standing practical problem in access control is the administrative costs of maintaining access control lists or other similar access control data structures. In a system with 1000 users, 100,000 objects and 10 access rights (a relatively small system by today’s standards), there are $10^9$ possible authorizations. Moreover, if the user population is highly dynamic, the number of grant and revoke operations to be performed can become very difficult to manage. In addition, end-users do not own the information to which they are allowed access. The corporation or agency is the actual owner of data objects and control is often based on employee functions rather than data ownership. Role-based access control (RBAC) is an attempt to reduce administration costs and support access control policies directly able to represent the organizational structure of the corporation or agency. RBAC achieves its goals by introducing the concept of a role, which acts as an intermediary between users and permissions (see Figure 2.2). The idea is that there will be far fewer roles than either users or permissions. Typically

Fig. 2.2 Role-based access control.
2.1 Access Control Models

The basic concepts of RBAC are illustrated in Figure 2.3. The ANSI RBAC standard was released in 2004 [3]. It is based on earlier work by Sandhu et al. [77]. The two main components are the core component (see Figure 2.3), which does not include role hierarchies, and the hierarchical component, which does.

As shown by Figure 2.3, RBAC is defined in terms of several sets: a set of users $U$, a set of permissions $P$, and a set of roles $R$. A permission is usually assumed to be an object–action pair; the specific types and formats of permissions depend on the system in which RBAC is deployed. When used to control accesses to a relational SQL database, the objects are tables and other database objects, whereas the actions correspond to SQL commands, such as `SELECT` to read from a table, `INSERT` to add tuples to a table, and so forth. Users are associated with roles using a user-role assignment relation $UA$. This relation is a set of pairs of the form $(u, r)$, meaning that user $u$ is assigned to role $r$. Permissions are similarly associated with roles using a permission-role assignment relation $PA$.

Users interact with an RBAC system by activating a session. The notion of session is quite abstract as it is defined as “a mapping between a user and an activated subset of roles that are assigned to the user” [3]. We can consider the notion of database transaction as an instantiation of the notion of RBAC session. Typically, the user authenticates to the system and chooses to act in one or more of the roles to which he is assigned. If the user makes an access request for a permission during the session, the permissions of the session roles are considered. If
the requested permission is among them, the access request is granted. In general not many actual implementations of RBAC support sessions [56], mainly because it is a notion whose implementation highly depends on the actual system adopting RBAC for access control.

RBAC further reduces the administrative burden by introducing the idea of a role hierarchy, which is modeled as a directed acyclic graph in which the roles are the nodes. In other words, the role hierarchy is represented as a binary relation \( RH \) on \( R \). The transitive closure of this relation defines a partial ordering on the set of roles. The basic idea is that a role high up in the hierarchy will inherit the permissions of lower roles, without having to be explicitly assigned to those permissions. Clearly, this significantly reduces the number of permissions that need to be assigned to more senior roles, thereby reducing the administrative overheads in an RBAC system. Such an approach, however, introduces an overhead into the access control enforcement algorithm, because it is necessary to consider the permissions of all junior roles when making a decision.

RBAC is now widely supported in commercial systems, such as Oracle, Sybase, Windows 2003 and Solaris. A RBAC profile exists for XACML and it is widely used in workflow management systems. The interested reader is directed to the recent book by Ferraiolo et al., which provides an excellent overview of RBAC and its applications [37].

2.1.6 XACML

The widespread deployment of XML-based Web applications and Web services and the need for collaborating and sharing data across different organizational domains have pushed the development of a rich XML-based language to express access control policies. This has resulted in the definition of XACML, which is an extensible, XML-encoded language, for specifying access control policies, access requests, and access control decisions. XACML was conceived as one component of a distributed and inter-operable access control framework and is characterized by the following key aspects:

- It is based on the triple-based authorization: \( \langle \text{Resource}, \text{Subject}, \text{Action} \rangle \). (The term ‘Resource’ is the one adopted by
XACML to mean ‘protected object’.) Each entity in such a triple can be specified by using identifiers for the entity or by specifying a set of conditions against some attributes associated with the entity. As such, XACML can be considered as an attribute-based access control model.

- It supports a structured organization of access control policies. Basically the top element of an XACML policy is a set of policies, each of which aggregates other policy sets or policy elements. The main component of a policy element is the rule set, consisting of multiple rules, that is, of multiple triple-based authorizations.
- It supports negative authorizations and provides different algorithms for solving conflicting access control decisions resulting from different rules (e.g., rule-combining algorithms) and from different policies (e.g., policy-combining algorithms).
- The input/output to the XACML policy processor is clearly defined as XACML context data structure. Input data is referred by XACML-specific attribute designator as well as XPath expression.
- It is extensible. Extension points include: functions, identifiers, data types, rule-combining algorithms, and policy-combining algorithms.

It is important to note that XACML does not introduce a new policy model, with respect to the discretionary, mandatory and RBAC models, and also it does not include an administration model. However XACML introduces a generalized approach for denoting subjects, objects, and actions in access control policies. Such generalized approach, referred to as attribute-based access control, is based on the principle that all subjects, objects, and actions are denoted in access control policies by means of conditions against some of their attributes. By properly defining such attributes, one can use XACML to encode all the other models.

The XACML standard also includes a non-normative data flow model, shown in Figure 2.4, that identifies the major components
Fig. 2.4 The flow model of XACML.

involved in processing access requests and their interactions. This model, which can be considered to be an evolution of the ISO 10181-3 model [45], can be used as a reference model for the implementation of an XACML engine. Relevant components of this flow model include:

- The Policy Administration Point (PAP), which supports authorization policy authoring and stores the authored policies in the appropriate repository.
2.1 Access Control Models

- The Policy Enforcement Point (PEP), which performs access control by making decision requests and enforcing authorization decisions.
- The Policy Information Point (PIP), which serves as the source of attribute values, or the data required for policy evaluation.
- The Policy Decision Point (PDP), which evaluates the applicable policy and renders a response to the PEP containing the authorization decision. The possible response values are: Permit, Deny, Indeterminate (in case an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request cannot be answered by this service).

The PEP and PDP might both be co-located within the same application, or might be distributed across different servers. In XACML, the access request is represented by the Request schema that specifies the requesting Subject, the requested Object and the specific Action requested on the Object. The XACML policy language was designed to be general enough so as to describe general access control requirements.

In XACML, a policy is the smallest element that the PDP can evaluate. A policy represents a single access control policy, expressed through a set of rules. A rule specifies the target to which it applies and the effect of the rule; that is, Permit or Deny. The target basically models the access request, by means of a set of simplified conditions for the subject, resource and action that must be met, i.e., evaluate to the Boolean value true, for the rule to apply to a given request. In other words, the target of a rule specifies the set of requests to which the rule is applicable; such specification is intensionally expressed by a set of conditions. Any number of rule elements may be used, each of which generates a true or false outcome. Combining these outcomes yields a single decision for the policy, which may be a Permit, a Deny, an Indeterminate, or a NotApplicable decision. A decision may also contain some obligations, that is, actions, that must be executed once the subject has obtained access to the data. Examples of obligations include
logging information about the accessed data or notifying an individual that his/her personal data have been accessed. As such, obligations are important for applications for which data privacy is an important requirement [66]. Figure 2.5 provides a graphical overview of the policy element, which is the main XACML element for the definition of policies.

As a policy example, consider the following one: “MPEG movie for adults cannot be downloaded by users with age less than 18 years”. The movie is the resource to which access must be controlled; it will be modeled by an element having an attribute “category”. Similarly, the subject will have an attribute “age”. In this case, the policy comprises a single rule, that specifies:

- the condition “age less than 18” for the subject;
- the condition “category = adult only” for the resource;
- the condition: “download” for the action;
- and the effect Deny.

Note that several rules, or even several policies, may be applicable to a given access request. The combining algorithms are then used to reconcile multiple outcomes into a single decision, namely, deny-overrides, permit-overrides, first-applicable, only-one-applicable.
Even though XACML has contributed to widely known important research notions developed by academia, such as attribute-based and credential-based access control, it has also some drawbacks. It has been designed to be used in applications in which there is a centralized administration; it is also very complex and thus the analysis of XACML policies is quite difficult. Some preliminary approaches have been proposed by Rao et al. [75, 58]; however much more work is needed with respect to tools for managing, composing, refining, and analyzing XACML policies.

2.1.7 Advanced Access Control Models

The above discussed models have been widely extended. An important class of extensions is represented by access control models able to take into account context information when making access control decisions. Two notable such models, GEO-RBAC [31] and T-RBAC [7], have been defined as extensions of RBAC to take into account spatial contextual information and temporal contextual information, respectively.

GEO-RBAC is motivated by the security requirements of location-based services and mobile applications as well as the increased concern for the management and sharing of geographical information in strategic applications like environmental protection and homeland security. These applications pose interesting requirements for access control systems. In particular, the permissions assigned to users depend on their position in a reference space; users often belong to well defined categories; objects to which permissions must be granted are located in that space; access control policies must grant permissions based on object and user locations. As an example, consider a mobile application for the personnel and patients of a health care organization. Individuals are given a location-aware terminal with which they can request information services provided by an application server. The organization consists of individuals who have different functional roles, e.g., Nurse, Doctor and Patient. We note that, depending on the organizational context, the services available to users may differ based on the functional roles of users. For example, the services available to nurses may be different from those available to doctors, not simply because
of the individual preferences but mainly because of organizational and functional reasons. Further, the availability of services may depend on the position of the requester. For example, a nurse may be allowed to request the record of a patient only when located in the department to which she has been assigned.

To deal with the requirements listed above, an access control model with spatial capabilities is needed. However, conventional RBAC does not suffice to support such applications and needs to be extended with suitable constraints specifying location constraints: that is, constraints concerning the locations in which a given role can be accessed by a user. It is important to notice that locations can be physical, that is, expressed as coordinates in the reference space, or logical, that is, expressed in terms of spatial objects (such as: the city of Rome, the West Valley Hospital) that have a semantics which is relevant to the specific application domains. GEO-RBAC directly supports such type of location constraints. It is based on the notion of a spatial role that is a geographically bounded organizational function. The boundary of a role is defined as a geographical feature, such as a road, a city or a hospital, and specifies the spatial extent in which the user has to be located in order to use the role. Besides a physical position obtained from a given mobile terminal such as a GPS-based vehicle tracking device or a cellular phone, users are also assigned a logical and device independent position, representing the feature in which the user is located. Logical positions can be computed from real positions by using specific mapping functions and can be represented at different granularities, depending on the spatial role played by the user. If the user is located inside the spatial boundary of the role which has been selected (activated) during the session, the role is said to be enabled. Only when a role is enabled, the user is actually allowed to use the permissions granted to the role.

The T-RBAC model constrains the use of permissions assigned to roles to specific temporal periods. Therefore, even though a user has the permission to use a role, and thus to use all the permissions assigned to this role, the user may only use the role in specified temporal intervals, which can also be periodic (for example, “every Monday from 9 am to 5 pm”). T-RBAC extends RBAC with the association of temporal (possibly periodic) intervals with roles. In T-RBAC roles can be in two
mutually exclusive states: active state, which is the state in which the role can be used; non-active state, which is the state in which the role cannot be used. In addition, T-RBAC introduces the notion of dependencies among such role activation/de-activation, expressed by means of role triggers, whose actions may be either executed immediately, or be deferred by an explicitly specified amount of time. Both triggers and periodic activations/deactivations may have a priority associated with them, in order to resolve conflicting actions. A formal semantics for the specification language is provided, and a polynomial safeness check is introduced to reject ambiguous or inconsistent specifications. We refer the reader to [7] for additional details.

2.1.8 Authorization Administration

The last, but an important component of an access control model based on the use of authorizations is the authorization administration model. The administration model specifies which subjects can enter authorizations into the access control system. This is obviously a crucial element to be defined for any comprehensive access control solution. Several approaches are possible. The most straightforward approach is a centralized approach; some special users, like the system administrators or the database administrators (DBA), are in charge of granting and revoking authorizations. A more effective approach is based on administration decentralization. Such an approach gives different users the authority to administer different objects. The most common decentralized approach is based on the ownership principle, that is, the owner of an object is the user in charge of administering the authorizations on the object. The owner is usually the user who has created the object; however exceptions are possible in some systems whereby a user can create an object indicating another user as the owner. Decentralization is often coupled with delegation, by which a user who has the authority to administer authorizations on an object can give other users the authority to administer authorizations on the object. When delegation is used, an object may end up with several users who are authorized to administer authorizations on it and users may thus receive multiple authorizations for the same object. Delegation is typically included in access control models defined for relational databases. It has also
been widely investigated for RBAC, where authorization administration also includes managing the set of roles and the role hierarchies, if defined. We refer the reader to the paper by Crampton and Loizu [28] for foundational work on RBAC administration.

2.2 Cryptographic Preliminaries

Cryptography is concerned with principles and protocols of data transformation with the purpose of ensuring confidentiality, integrity and authenticity. In this section, we provide a brief overview of symmetric and asymmetric encryption functions, hashing and digital signatures. We also touch upon basic concepts of elliptic curve cryptography.

An encryption function takes as input a block of data called plaintext and transforms it into a ciphertext with the help of an encryption key. The contents of the plaintext can be recovered from the ciphertext with the help of a decryption key. Without the decryption key, it is hard for an adversary to recover the original data block from its encrypted version.

In symmetric cryptographic methods, the same key is used for both encryption and decryption. Advanced Encryption Standard (AES) is a widely-used encryption technique that uses fixed data blocks of 128 bits and key sizes of 128, 192 or 256 bits. The encryption algorithm comprises of a sequence of transformation rounds consisting of shift and addition operations in a finite field. Symmetric encryption has the advantage of being fast, as the required block operations can be efficiently implemented. However, if two (or more) parties exchange data encrypted with a symmetric method, then the key must be transmitted over a secure communication channel.

Asymmetric, or public-key encryption uses different keys for encryption and decryption. Specifically, given encryption and decryption functions $E$ and $D$, public key $K_{pub}$ and private key $K_{priv}$, the transformations between a plaintext message $M$ and a ciphertext message $C$ are performed as follows:

$$C = E_{K_{pub}} (M)$$

$$M = D_{K_{priv}} (C)$$
Furthermore,

\[ D_{K_{\text{priv}}} (E_{K_{\text{pub}}} (M)) = M \]

and

\[ E_{K_{\text{pub}}} (D_{K_{\text{priv}}} (M)) = M \]

Asymmetric encryption has the advantage that parties exchanging data do not need a secure communication channel to exchange keys. Each party keeps its own private key secret, whereas the public key can be released on an unsecured channel. However, asymmetric encryption schemes are more expensive in terms of computational time. RSA\(^1\) is one of the most widely used public-private key schemes, and relies on the computational intractability of factoring large integers.

Hash functions are one-way transformations that map variable-length messages into a fixed-length code, or hash (typically 128, 160 or 256 bits). Furthermore, the transformation is hard to reverse, i.e., given the hash code, it is computationally intractable to recover the initial message. In addition, it is computationally intractable to find another message that results in the same hash code. A hash code that satisfies the latter property is called collision resistant. The most widely used collision resistant hash functions are MD5 and SHA-1.

Public key cryptography in conjunction with one-way hashes provide the support to implement digital signatures. Digital signatures allow message authentication and verification of integrity, i.e., a receiving party can check that a message was indeed sent by its claimed source, and that the message has not been tampered with during network transfer. Specifically, the source computes the hash value \( H(M) \) of a message, and encrypts the result with its private key \( D \), which also acts as a signing key. The signature is denoted by

\[ \sigma = D_{K_{\text{priv}}} (H(M)) \]

The recipient retrieves the message, re-computes the hash, decrypts the received hash code and checks whether the two hash values match. If they do, then the integrity and authenticity of the message are verified.

\(^1\)http://www.rsa.com.
A particular family of public-key encryption schemes that has received increasing attention in recent years is elliptic curve cryptography (ECC) over finite fields. Specifically, bilinear mappings over elliptic curve groups can allow for advanced encryption and signature functionality.

Let $G$ and $G_T$ be two groups of order $p$, where $p$ is a large prime, and denote by $g$ a generator of group $G$. A bilinear mapping $e$ is an application

$$e: G \times G \rightarrow G_T$$

such that

- $\forall a, b \in \mathbb{Z}_p$, it holds that $e(g^a, g^b) = e(g, g)^{ab}$ and
- $e(g, g) \neq 1$

A bilinear map is called admissible if it can be computed efficiently.

Later in Section 9.3 we will use two intractability assumption, namely the strong Diffie–Hellman assumption and the Diffie–Hellman exponent assumption [22].

Strong Diffie–Hellman Assumption. The $\ell$ strong Diffie–Hellman assumption holds for a group $G$ of prime order $p > 2^k$, where $k$ is a security parameter, if for any polynomial-time algorithm $A$, the probability

$$\Pr[A(g, g^x, \ldots, g^{x^\ell}) = (c, g^{1/(x+c)})]$$

is negligible in $k$, where $x$ and $c$ are randomly chosen from $\mathbb{Z}_p$.

Diffie–Hellman Exponent Assumption. The decision $\ell$-bilinear Diffie–Hellman exponent assumption holds for two groups $G, G_T$ of prime order $p > 2^k$, where $k$ is a security parameter, if for any polynomial-time algorithm $A$, the probability

$$\Pr[A(g, h, g^\alpha, \ldots, g^{\alpha^\ell+1}, g^{\alpha^{2^\ell}}, e(g, h)^{\alpha^\ell}) = 1] - \Pr[A(g, h, g^\alpha, \ldots, g^{\alpha^\ell+1}, g^{\alpha^{2^\ell}}, S) = 1]$$

is negligible in $k$, where $g$ and $h$ are randomly chosen from $G$, $S$ is chosen at random from $G_T$ and $\alpha \in \mathbb{Z}_p$.

Finally, we introduce the concept of Zero-Knowledge Proof of Knowledge (ZKPK), which is used in the privacy-preserving access
control mechanism presented in Section 9 [22]. ZKPK is a two-party protocol executed between a prover and a verifier. The objective of the protocol is to allow the prover to convince the verifier that s/he knows a secret value that satisfies some secret condition (the “proof-of-knowledge” property) without the verifier learning anything about the secret value or the condition (the “zero-knowledge” property).

2.3 Summary

In this section, we have surveyed the main notions and models for access control and the most relevant access control systems and standards. It is important to emphasize that the security research is very active in the area of access control and many relevant research directions are being investigated. A relevant direction is represented by access control for grid computing systems and virtualized environments. Those systems and environments are quite challenging because of the very large number of users and of the distributed administration of resources. They are characterized by the fact that there is no single authority controlling all resources that may be required by a user to perform certain tasks. In such cases, the user must be able to obtain multiple authorizations from independent administrative authorities; this approach however entails the issue of conflicting authorizations. Another important area is access control for web service security and workflow systems. Despite the several initiatives ongoing in the industry community, and in particular the definition of an XACML profile for web services, the problem of access control is still largely unexplored. Research is needed to address the problem of conversational web services and the development of access control systems suitable for business processes expressed according to workflow models. Finally as access control systems evolve to take into account context information in access control decisions, the problem of dealing with missing and uncertain information arises. A preliminary approach, based on fuzzy inferences, has been proposed by Ni et al. [65]. However, research and case studies are also needed from the perspective of risk assessment in order to devise satisfactory approaches.
In this section, we discuss how access control models are applied and extended for use in relational databases. We start by introducing the access control model of System R which is the first example of a comprehensive discretionary model for relational databases. We then focus on the notion of content-based access control which is a type of access control specific to database systems. The main idea of content-based access control is that the decision on whether to permit or deny an access request depends on the contents of the data to be accessed. We finally discuss the implication of the use of mandatory access control in relational databases. The presentation is partially based on the papers by Bertino and Sandhu [19] and Bertino et al. [16].

3.1 The System R Access Control Model

Under the System R access control model, the main protection objects are tables and views, also referred to as virtual tables. The possible access modes that subjects can exercise on tables correspond to SQL operations that can be executed on tables, that is: select (to retrieve tuples from a table), insert (to add tuples to a table), delete (to remove
3.1 The System R Access Control Model

39

...tuples from a table), and update (to modify tuples in a table). The update access mode can be further refined by specifying the table columns to which it applies. Therefore, a user may receive the privilege to modify only a subset of the columns of a table. The same access modes are defined for views with the difference that some access modes may not be applicable to a view depending on the view definition. For example, delete, insert, and update operations may not be allowed on views defined as joins or containing aggregate functions. Therefore, for such views those operations are not allowed by the access control mechanism. In the remainder, we use the term table to refer to both base tables and views.

Current DBMSs have extended the basic model by introducing new types of objects to be protected as a consequence of extensions to the relational data model. Therefore the set of protection modes that one finds in these DBMSs is much larger than the set defined as part of the basic model. For example, the introduction of trigger mechanisms [87] has required the introduction of a specific access mode allowing a subject to create a trigger on a table. Similarly, the introduction of mechanisms for referential integrity, through the use of foreign keys, has required the introduction of a related access mode allowing a subject to reference a table from another table.

Authorization administration in the System R model is based on the ownership approach coupled with administration delegation. Any database user authorized to do so can create a new table. When a user creates a table, he/she becomes the owner of the table and is solely and fully authorized to exercise all access modes on the table. The owner can further delegate privileges on the table to other subjects by granting these subjects authorizations with the so-called grant option. The command for granting authorizations, introduced by System R, is the GRANT command; its basic format is as follows:

\[
\text{GRANT \{ALL PRIVILEGES|<privileges>\} ON <table> TO \{<user-list>|PUBLIC\} [WITH GRANT OPTION]}\]

1In the command syntax the curly brackets denote a set of alternative elements of the command, one of which must always be included in the command. The square brackets denote optional components of the command.
The **ALL PRIVILEGES** and **PUBLIC** are keywords that denote the set of all access rights and all users, respectively. The **WITH GRANT OPTION** specifies that the user receiving the privileges specified by the command, that is, the *grantee*, receives also the administration authorization on these privileges and can thus grant them to other users. The user issuing a **GRANT** command is referred to as the *grantor* of the authorizations. The following are examples of the use of the grant command.

In the example commands the name before the character ‘:’ denotes the grantor.

- Bob: `GRANT select, insert ON Employee TO Ann WITH GRANT OPTION;`
- Bob: `GRANT select ON Employee TO Jim WITH GRANT OPTION;`
- Ann: `GRANT select, insert ON Employee TO Jim;`  

As a result of the execution of the above **GRANT** commands, user Jim has the **select** privilege (received from both Bob and Ann) and the **insert** privilege (received from Ann). Jim in turn can grant to other users the **select privilege** because he has received it with the grant option; however, he cannot grant the **insert** privilege. A special set of catalogs, referred to as the authorization catalogs, keep track for each user of the privileges the user possesses and of the ones that the user can delegate. Whenever a user *u* executes a **Grant** command, the system intersects the delegatable privileges of *u* with the set of privileges specified in the command. If the intersection is empty, the command is not executed.

Authorizations can be dynamically revoked at any time through the use of the **REVOKE** command. An important principle stated by the System R model is that a user can revoke an authorization only if the user is the grantee of this authorization. The format of the **REVOKE** command is as follows:

```plaintext
REVOKE {ALL PRIVILEGES| <privileges>} ON <table>
FROM {<user-list> | PUBLIC}
```

Note that it is not possible to revoke only the grant option. If the grant option must be revoked, one must revoke the entire authorization, and then re-issue it without the grant option. Also as a user may receive
3.1 The System R Access Control Model

the same privileges from multiple grantors, because of decentralized administration, the execution of a revoke operation does not necessarily result in the user loosing the privilege. As an example consider the following sequence of GRANT and REVOKE commands that revoke some of the privileges granted commands:

- Bob: GRANT select ON Employee TO Jim WITH GRANT OPTION;
- Bob: GRANT select ON Employee TO Ann WITH GRANT OPTION;
- Jim: GRANT select ON Bob.Employee TO Tim;
- Ann: GRANT select ON Bob.Employee TO Tim;
- Jim: REVOKE select ON Bob.Employee FROM Tim;

After the execution of the revoke operation, Tim continues to hold the select access right privilege on table Employee since he has independently obtained such privilege from Ann.

The possibility of delegating authorization administration introduces however some interesting issues concerning the semantics of the revoke operations. A user, to whom the administration right on a given table has been granted and then revoked, may have granted to another user a right to access this table. The question is what happens to this authorization when the revocation takes place. The semantics of the revocation of a privilege from a subject (revokee) by another subject (revoker) is to consider as valid only the privileges that would have been present had the revoker never granted the revokee the privilege. This semantics is formally defined as follows [41].

Let $G_1, \ldots, G_n$ be a sequence of grant operations with a single privilege on the same relation, such that for $i, k = 1, \ldots, n$, if $i < k$, then $G_i$ is executed before $G_k$. Let $R_i$ be the revoke operation for the privilege granted with operation $G_i$. The semantics of the recursive revoke requires that the state of the authorization system after the execution of the sequence $G_1, \ldots, G_n, R_i$ be identical to the state that one would have after the execution of the sequence $G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n$.

As a consequence, every time a privilege is revoked from a subject, a recursive revocation takes place to remove all authorizations for this table from the revokee. The revocation is iteratively applied to all the
subjects that received access privileges from the revokee. The revoke operation thus takes into account the temporal sequence according to which the grant operations were made. The temporal sequence is determined according to the timestamps that are associated with the granted authorizations.

To understand this concept, consider the sequence of grant operations for privilege \( p \) on table \( t \) represented as a graph in Figure 3.1(a), where every node represents a user and an arc between the node denoting user \( u_i \) and the node denoting user \( u_j \) indicates that \( u_i \) granted \( p \) on \( t \) to \( u_j \). The label of the arc indicates the time at which the privilege was granted. For sake of simplicity, we assume that all authorizations are granted with the grant option. Suppose now that Ann revokes the privilege on the table from Jim. According to the semantics of recursive revocation, the resulting authorization state has to be as if Jim had never received the authorization from Ann. If Jim had never received the authorization from Ann, he could not have granted the privilege to Sue (his request would have been rejected by the system). Analogously, Sue could not have granted the authorization to Dave. Therefore, the authorization granted by Jim to Sue and by Sue to Dave must also be deleted. Note that the authorization granted by Jim to Pat does not have to be deleted since Jim could have granted it even if he had not received the authorization from Ann.

![Fig. 3.1](image-url)

Fig. 3.1 Ann revokes the privilege from Jim. (a) The authorization state before the revoke. (b) The authorization state after the revoke.
never received the authorization from Ann (because of the authorization from Chris). The set of authorizations holding in the system after the revocation is shown in Figure 3.1(b).

Several extensions to the basic model have been proposed with the goal of enriching the expressive power of the authorization language, including negative authorizations [18] and temporal authorizations [6]. Negative authorizations, also referred to as denials, allow one to specify that a subject cannot receive an authorization. Negative authorizations are crucial for supporting exceptions to a given set of authorizations. Temporal authorizations have an associated temporal interval, which can also be periodic; such an interval specifies that the authorization is valid, that is, can be used only during the interval. We do not elaborate on these extensions and we refer the reader to the survey by Bertino and Sandhu [19] for more details and a discussion of additional extensions to the System R authorization model.

3.2 Content-based Access Control

Content-based access control is an important requirement that any access control mechanism for use in a DBMS should satisfy. Essentially, content-based access control requires that access control decisions be based on data contents. Consider an example of a table recording information about employees of a company; a content-based access control policy would be that a manager can only access the employees that work in his/her division. Whenever a manager issues a query, the system has to filter the query result by returning only the tuples related to the employees that verify the condition of working in the division managed by this manager. Support for this type of access control has been made possible by the fact that SQL is a language for which most data operations, such as queries, are content-based, that is, they identify the tuples to which the operations apply by specifying conditions on the tuples’ content.

The most common mechanism, adopted by relational DBMSs to support content-based access control, is based on the use of views [41]. A view is defined by a query that can filter out tuples and columns, and also return aggregation of values computed through statistical
operators, like \texttt{AVG}, and \texttt{COUNT}. As such, views provide a mechanism that, even though very simplistic, allows one to release only statistical summaries of data. A view can be considered as a dynamic window able to select subsets of columns and rows; these subsets are specified by defining a query, referred to as a \textit{view definition query}, which is associated with the name of the view. Whenever a query is issued against a view, the query is modified through an operation called \textit{view composition} that replaces the view referenced in the query with its definition. An effect of this operation is that the “where clause” in the original query is combined, through the AND Boolean connective, with the “where clause” of the view definition query. Thus, the query which is executed against the base table, that is, the table on which the view is defined, filters out the tuples that do not satisfy the predicates in the view. Also when a view contains a subset of the columns of the underlying table, any query specified against the view can only refer the columns available in the view and can only return values from these columns. When access by a user has to be restricted through a view, one has to grant this user access privileges on the view and not on the underlying table. The following example illustrates the discussion.

Assume that we have a table, called \texttt{Employees} storing information, such as salary, bonus, empno, and job, about employees of a company. Assume that we would like to authorize user Ann to access only the employees whose salary is lower than 20,000. The SQL commands that would have to be executed are:

- \texttt{CREATE VIEW Vemp AS SELECT * FROM Employees WHERE Salary < 20000;}
- \texttt{GRANT Select ON Vemp TO Ann;}

Suppose now that Ann would like to retrieve all employees who are programmers. Thus, as she can only access the view, she will issue the following query:

\texttt{SELECT * FROM Vemp WHERE Job = Programmer;}

The view composition operation, executed as part of query processing, will generate the following query which is then executed on
3.2 Content-based Access Control

the table on which the view is defined:

```sql
SELECT * FROM Employee WHERE Salary < 20000 AND Job = Programmer;
```

Notice that the query processor, when processing a query on a view, first performs access control to verify that the query issuer is authorized for access to the view, and only then performs view composition. The correct execution order for these two operations — access control and view composition — is thus crucial as access control has to be executed against the view and not on the underlying table.

An important question to be discussed about views is which are the access rights that the user defining the query, e.g., the view definer, receives on the view. Views are different from conventional stored tables in that they are derived from other objects. Therefore the access rights that a view definer receives on his/her view are constrained by the access rights the view definer has on the underlying table. These access rights received on a view by the view definer can be also further restricted depending on the operations that can be executed on the view. The following example illustrates the discussion.

Suppose that a user Bob has the \texttt{SELECT} access right on the table \texttt{Employee}; and moreover he has the \texttt{UPDATE} access right on all the columns of this table. Suppose that Bob creates a view by issuing the following statement:

```sql
CREATE VIEW V1 (EmpNo, Total_Sal)
AS SELECT EmpNo, Salary + Bonus FROM Employee WHERE Job = Programmer;
```

The access rights that Bob receives are \texttt{SELECT}, and \texttt{UPDATE} only on the column \texttt{EmpNo} as the update operation is not defined on column \texttt{Total_Sal} of the view because this column is defined by an expression. Notice also that, even though the delete operation is possible on this view, Bob does not receive the \texttt{DELETE} access right because he does not have this access right on the underlying table. Basically, to determine the privileges that the view definer has on the view, the system needs to intersect the set of privileges that the view definer has on the base tables with the set of privileges corresponding to the operations that can be performed on the view. A similar approach is used to determine
whether the view definer can grant authorizations on his/her own table to other users.

There are several advantages in the use of the view mechanism for content-based access control. Content-based access control policies are expressed at a high level in a language consistent with the query language. Modifications to the data do not need modifications to the access control policies; if new data are entered that satisfy a given policy, these data will be automatically included as part of the data returned by the corresponding view. However the view mechanism has several disadvantages when fine-grained access control at the tuple level, or even at the cell level, has to be enforced. A naive approach would require the specification of a view for each tuple or part of a tuple that is to be protected. Moreover, because access control policies are often different for different users, the number of views would further increase. Furthermore, application programs would have to code different interfaces for each user, or group of users, as queries and other data management commands would need to use for each user, or group of users, the correct view. Modifications to the access control policies would require the creation of new views with consequent modifications to application programs. Alternative approaches that address some of these issues have been proposed, and these approaches are based on the idea that queries are written against the base tables and, then, automatically rewritten by the system against the view available to the user. The Oracle VPD mechanism is an example of such approaches. The Oracle VPD mechanism does not require one to code different interfaces for different users and, thus, addresses one of the main problems in the use of conventional view mechanisms. We discuss the Oracle VPD mechanism in more detail in Section 4 and more advanced approaches to fine-grained access control in Section 5.

3.3 Mandatory Access Control Models

The application of the mandatory access control model to relational databases has been extensively investigated. The use of such access control model for databases requires addressing several challenging issues.
Solutions to some of these issues have resulted in extensions to the definition of the relational model itself, resulting in the so-called *multilevel relational model*, and to fundamental notions such as the notion of relational key.

A multilevel relation is characterized by the fact that different tuples in a same relation may have different access classes. The relation is thus partitioned into different security partitions, one for each access class. A partition associated with an access class \( c \) contains all tuples whose access class is \( c \). A subject having access class \( c \) can read all tuples in partitions of access classes that are equal to or lower than \( c \); such a set of tuples is referred to as a *view of the multilevel relation* at access class \( c \). By contrast, a subject having access class \( c \) can insert tuples into partitions or update and delete tuples in partitions that have an access class equal to or higher than \( c \). In some implementations of the multilevel relational model, data modification operations at higher access classes are not allowed for integrity reasons. Such a restriction is usually known as a *no write-up restriction*.

The multilevel relational model is further complicated if tuples are allowed to have attributes classified at different access classes. Each attribute of each tuple, that is, a cell, thus has a *label* attribute, denoting the access class of the attribute in the tuple, and a tuple label, which is the lowest element in the set of access classes associated with the attributes of the tuple. Figure 3.2 shows a multilevel relation in which tuples and their attributes may have classes from the access class set \{Low, High\}. The example shows that a possible approach to recording for each tuple and each column within each tuple the access class is to add for each column in a relation a corresponding column recording

<table>
<thead>
<tr>
<th>Name</th>
<th>C(_{\text{Name}})</th>
<th>Dept#</th>
<th>C(_{\text{Dept#}})</th>
<th>Salary</th>
<th>C(_{\text{Dept#}})</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Low</td>
<td>Dept1</td>
<td>Low</td>
<td>100K</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Ann</td>
<td>High</td>
<td>Dept2</td>
<td>High</td>
<td>200K</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Sam</td>
<td>Low</td>
<td>Dept1</td>
<td>Low</td>
<td>150K</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Fig. 3.2 An example of multilevel relation.
the access class. An additional column, which in the example has name TC, records the access class of the entire tuple.

The “view” of a table at an access class c is called multi-level relation instance at class c. Figures 3.3 and 3.4 show the low instance and high instance of the relation in Figure 3.2. In these examples, we use the value “null” to denote null values. These values are used to replace the values of tuple’s attributes that cannot be accessed at a given access class.

Because each instance must be correct with respect to the primary key constraint, two correctness conditions must be satisfied by a multilevel relation:

- for each tuple in a multilevel relation, the attributes of the primary key must have the same access class;
- for each tuple in a multilevel relation, the access class associated with the non-key attributes must dominate the access class of the primary key.

These conditions assure that at each access class, where tuples of the relation are visible, the tuples have a key which has all key attributes visible and different from null.
A consequence of the fact that the same tuple may belong to several partitions of a multilevel relation results in the tuple *polyinstantiation* and, thus, in update anomalies. Handling polyinstantiation requires revisiting several classical notions of the relational model, such as the notion of a key. Because of such problems, commercial implementations of the multilevel relational model only support tuple-based labeling.

The development of multilevel secure DBMSs entails, however, extending not only the data model, but also the system architecture to make sure that covert channels would be closed. A *covert channel* allows one to transfer information that violates the security policy. There are two main types of covert channels: *timing channels*, under which information is conveyed by the timing of events or processes; and *storage channels* that do not require any temporal synchronization in that information is conveyed by accessing system information. A well-known type of covert channel in a DBMS is represented by the 2-phase locking (2PL) protocol used for transaction synchronization. The problem of designing concurrency control mechanisms secure against covert channels has been widely investigated. Most approaches were based on the principle that transactions cannot be delayed or aborted due to a lock conflict with a higher-level transaction. Hence, low-level transactions have higher priority on low-level data than higher-level transactions. The consequence is that even though a transaction may have acquired a read lock on a lower-level data item, it may be forced to release this lock if a lower-level transaction requires a write lock on it. Due to such prioritization, transaction execution histories may not always be serializable. Several approaches have been proposed to address the issue of how to synchronize transactions so that timing channels do not occur and, at the same time, serializability is achieved. However, they suffer from several shortcomings, such as starvation of high-level transactions that can be repeatedly aborted, or require multiple versions of data, or force high-level transactions to read stale data. A different approach [10] was later defined based on the notion of application-level recovery and notification-based locking protocols combined with nested transaction protocols [62].
3.4 Summary

In this section, we have discussed the foundations of discretionary and mandatory access control models for relational databases and techniques for content-based access control, which is an important form of access control specific to databases. Even though current models and mechanisms are quite comprehensive, research is needed to understand how DBMSs can efficiently support richer access control languages, such as the XACML standard language for access control policies.
In this section, we present three case studies that provide an overview of the different flavors of access control mechanisms implemented in popular commercial DBMSs. We first describe the core access control mechanism in Microsoft’s SQL Server 2008 DBMS [83]. We then present the Oracle Virtual Private Database technology that implements a fine-grained access control mechanism in an Oracle database [69]. Finally, we describe the Oracle Label security mechanism that implements the concepts of mandatory access control policies in an Oracle database.

4.1 SQL Server 2008

SQL Server 2008 is Microsoft’s latest offering of their relational database management system (RDBMS) product [83]. In what follows, we provide an overview of SQL Server’s capabilities for authorizing user access to the stored data.

4.1.1 Base Model

The base authorization model of SQL Server 2008 consists of the traditional components of an access control model introduced in Section 2.3,
namely, *principals*, *resources* (*securables* in SQL Server terminology), and *permissions*. SQL Server regulates the actions of principals on securables by verifying that they have been granted appropriate permissions. We describe each of the components in detail below.

**Principals.** A *principal* is an entity that can request access to resources managed by an SQL Server instance. Principals can be arranged in a hierarchy just like many other components of the SQL Server authorization model. SQL Server defines three types of principals: *Windows-level* principals such as the Windows domain and local users; *SQL Server-level* principals such as the SQL Server logins and SQL Server roles; and *Database-level* principals such as the database users, database roles, and application roles. The *scope of influence* of a principal depends upon the scope of its definition, that is, Windows, server, or database. For every user that needs to access SQL Server resources, an SQL Server login must be created. Such login is then mapped to one or more database users for granting access to a database’s resources.

**Securables.** A *securable* is an SQL Server managed resource to which the SQL Server Database Engine regulates access. Securables can be categorized into *scopes* that in turn may be arranged in a hierarchy. The securable scopes are *server*, *database*, and *schema*. Figure 4.1 shows the various SQL Server securables in their respective scopes.

**Permissions.** Every SQL Server securable has associated *permissions* that can be granted to a principal. SQL Server introduces the concept of *generic* permissions that apply to all securables and *specific* permissions such as *select*, *insert* that apply only to specific securables. Some examples of generic permissions are as follows:

- **ALTER.** This permission confers the grantee the ability to change any property of a securable except its ownership.
- **ALTER ANY Server Securable.** This permission allows the grantee to create, alter, or drop individual instances of the Server Securable. For example, ALTER ANY LOGIN confers the ability to create, alter, or drop any server login in the instance.
4.1 SQL Server 2008

Fig. 4.1 Hierarchy of securables in SQL Server 2008.

- **ALTER ANY Database Securable.** This permission confers the grantee the ability to CREATE, ALTER, or DROP individual instances of the Database Securable. For example, ALTER ANY SCHEMA confers the ability to create, alter, or drop any schema in the database.

- **CREATE Server Securable.** This permission confers the grantee the ability to create the Server Securable.

- **CREATE Database Securable.** This permission confers the grantee the ability to create the Database Securable.

- **CREATE Schema-contained Securable.** This permission confers the ability to create the schema-contained securable.

- **CONTROL.** A special mention must be made of the generic Control permission that confers ownership like capabilities to the grantee. A grantee that owns the Control permission on a securable can also grant permission to other principals.
Moreover, the Control permission implies all permissions on all securables under the scope of the securable on which it is granted.

SQL Server also follows a hierarchical permission model with the notion of covering and implied permissions. For example, the Alter permission on a schema implies Alter permission on all securables defined under that schema. In such case, the Alter permission on the securables defined under the schema is the implied permission and the Alter permission on the schema itself is the covering permission. Figure 4.2 shows a partial view of the permission hierarchy in SQL Server 2008.

4.1.2 Access Control Administration

In this section, we give a brief overview of some of the interesting access control administration features in SQL Server 2008.
Table 4.1. Fixed server roles in SQL server 2008.

<table>
<thead>
<tr>
<th>Role</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Every SQL Server login belongs to the public server role.</td>
</tr>
<tr>
<td>dbcreator</td>
<td>Can create, alter, drop, and restore any database.</td>
</tr>
<tr>
<td>diskadmin</td>
<td>Can manage disk files.</td>
</tr>
<tr>
<td>bulkadmin</td>
<td>Can run the BULK INSERT statement.</td>
</tr>
<tr>
<td>setupadmin</td>
<td>Can add and remove linked servers.</td>
</tr>
<tr>
<td>processadmin</td>
<td>Can end processes that are running in an instance of SQL Server.</td>
</tr>
<tr>
<td>securityadmin</td>
<td>Manages logins and their properties. Can also GRANT,</td>
</tr>
<tr>
<td></td>
<td>DENY, and REVOKE database-level permissions.</td>
</tr>
<tr>
<td>serveradmin</td>
<td>Can change server-wide configuration options and shut down the server.</td>
</tr>
<tr>
<td>sysadmin</td>
<td>Can perform any activity in the server.</td>
</tr>
</tbody>
</table>

### 4.1.2.1 Role-based Administration

SQL Server adopts the role-based administration model for permission management. Roles are defined at the SQL Server level and at the database level. The permission scope of server level roles is across the entire server instance. The server level roles, however, are predefined, that is, it is neither possible to add user-defined server level roles nor is it possible to add new permissions to the existing roles. Table 4.1 describes the various server level roles that ship with SQL Server 2008 and their purpose.

Database-level roles are database-wide in their permissions scope. SQL Server 2008 ships with a set of fixed database roles described in Table 4.2. Such roles exist in each SQL Server database by default. In addition, it is possible to add user-defined database roles using the `create role` SQL command. The `create role` SQL command is described in Listing 4.1.

```
CREATE ROLE role_name [ AUTHORIZATION owner_name ]
```

Listing 4.1. Role Creation in SQL Server 2008

The meaning of the various arguments is as follows [29]:

- **role_name.** Is the name of the role to be created.
- **AUTHORIZATION owner_name.** Is the database user or role that is to own the new role. If no user is specified, the role will be owned by the user who executes CREATE ROLE.
Case Studies

Table 4.2. Fixed database roles in SQL Server 2008.

<table>
<thead>
<tr>
<th>Role</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_owner</td>
<td>Can perform all configuration and maintenance activities on the database.</td>
</tr>
<tr>
<td>db_securityadmin</td>
<td>Can modify role membership and manage permissions.</td>
</tr>
<tr>
<td>db_accessadmin</td>
<td>Can add or remove access to the database for Windows logins, Windows groups, and SQL Server logins.</td>
</tr>
<tr>
<td>db_backupoperator</td>
<td>Can back up the database.</td>
</tr>
<tr>
<td>db_ddladmin</td>
<td>Can run any DDL command in a database.</td>
</tr>
<tr>
<td>db_datawriter</td>
<td>Can add, delete, or change data in all user tables.</td>
</tr>
<tr>
<td>db_datareader</td>
<td>Can read all data from all user tables.</td>
</tr>
<tr>
<td>db_denydatawriter</td>
<td>Cannot add, modify, or delete any data in the user tables within a database.</td>
</tr>
<tr>
<td>db_denydatareader</td>
<td>Cannot read any data in the user tables within a database.</td>
</tr>
</tbody>
</table>

The following example creates the database role `manager` that is owned by the `db_securityadmin` fixed database role.

CREATE ROLE manager AUTHORIZATION db_securityadmin;

Roles can also be arranged in a hierarchy using the role membership relation. A member (user or role) added to a role inherits the permissions of the role. Note that adding members to user-defined database roles requires one of the following:

- Membership in the `db_owner` fixed database role.
- Membership in the `db_securityadmin` fixed database role.
- Membership in the role that owns the role.
- ALTER permission on the role.

Adding members to the fixed database roles requires membership in the `db_owner` fixed database role.

**Application Roles.** Another interesting feature of SQL Server 2008 is the concept of application roles. An application role is a database level principal that allows an application to run with its own user-like permissions. Application roles can be used to enable access to specific portions of the database to only those users that connect through a particular application. In addition, unlike database roles, application roles do not contain any members and are inactive by default. The following steps make up the process by which an application role
switches security contexts. First, a user executes a client application. Then, the client application connects to an instance of the SQL Server as a database user. The application then sets the application role using the password associated with that role. At this point, the connection loses the permissions of the initial database user (through which the application initiated the connection) and assumes the permissions of the application role. The permissions acquired through the application role remain in effect for the duration of the connection.

4.1.2.2 Permission Management

In SQL Server 2008, there are two ways in which a user or role can obtain permission on a database object. First is the discretionary model where the user or role is granted the permission by another principal that has the capability to grant that permission. Second is through the role–membership relation. The SQL command for granting permissions is the GRANT command. Previously granted permissions may be revoked using the REVOKE command. The revoke command may be CASCADING, that is, it removes the revoked permission from other principals to which it has been granted by the principal whose permission is getting revoked.

An interesting feature of SQL Server 2008 is the DENY command that is used to implement the concept of negative authorizations [18]. The DENY command denies a permission to a principal, preventing that principal from inheriting the permission through its role memberships. The syntax of the DENY command is described in the Listing 4.2.

```
DENY { ALL [ PRIVILEGES ] } | permission [ ( column [ ,...n ] ) ] [ ,...n ] | ON [ class :: ] securable | TO principal [ ,...n ] | CASCADE | AS principal
```

Listing 4.2. DENY command syntax in SQL Server 2008

The meaning of the various arguments is as follows [34]:

- **permission.** The name of a permission.
- **column.** Specifies the name of a column in a table on which permissions are being denied.
• **class.** Specifies the class of the securable on which the permission is being denied.

• **securable.** Specifies the securable on which the permission is being denied.

• **TO principal.** The name of a principal. The principals to which permissions on a securable can be denied vary, depending on the securable.

• **CASCADE.** Indicates that the permission is denied to the specified principal and to all other principals to which the principal granted the permission. It is required when the principal has the permission with GRANT OPTION.

• **AS principal.** Specifies a principal from which the principal executing this statement derives its right to deny the permission.

### 4.2 Oracle Virtual Private Database

Oracle Virtual Private Database (Oracle VPD) is a fine-grained access control mechanism that allows Oracle database administrators to control database access at the row and column level [70]. At the very basic level, Oracle VPD dynamically attaches a WHERE clause to an SQL statement issued against a table or view to which an Oracle VPD policy has been attached. This WHERE clause is returned by an Oracle function implementing the VPD policy. The Oracle server then rewrites the SQL statement with the additional WHERE clause and executes it. Oracle VPD provides the following benefits:

• **Scalability.** Consider a table of employees that contain 1000 employee records. Suppose the requirement is such that employees should be able to access their own records only. Using views, we would need to create 1000 views. Using VPD, the requirement can be achieved with a single policy function.

• **Security.** Since the VPD policies are enforced at the database server level, it ensures that the same security is in force no matter how the data is accessed.
• Simplicity. The security policy needs to be added to a database object only once, rather than repeatedly adding to all applications that use the database objects.

4.2.1 VPD Policies

Creating a VPD policy is a two step process. First, one has to create a function that defines the restrictions to be enforced. The function takes as input arguments a schema name and a table or view name. The values for these input parameters are populated when the policy is created to attach the function to a particular table or view. The function provides a return value for the WHERE clause predicate that is attached to an SQL statement when a policy is activated. In most cases, the WHERE clause is designed to be different for each user, each group of users, or each application that accesses the objects that have VPD policies attached to them. For example, if a clerk logs in, the WHERE clause can be specific to the privileges of that particular clerk. This can be achieved by incorporating an application context, using which the function can access the user session information and use it in the WHERE clause generation code. In fact, depending upon the business requirements, an application can set any arbitrary application specific attributes (e.g., client location, position, etc.) that can be used by a VPD function implementing a VPD policy. Thus, VPD policies can also be used to implement context-aware access control.

The second step in the VPD policy creation process is creating the actual policy that associates the function with a table or a view. The policy may also specify the SQL statement(s) on the table or the view that results in a policy activation. In what follows, we give an example of a VPD policy. Suppose a user Bob has access to the following table:

\[
\text{employee_data(\text{emp\_name varchar, sal integer})}
\]

Suppose the policy requirement is such that a user should be able to access only their own salary data. The function shown in Listing 4.3 fulfills the requirement.
Create function example_func(p_schema char, p_obj char) 
Return char 
As 
  db_user CHAR;
Begin 
  db_user := SYS_CONTEXT(userenv, SESSION_USER);
  return emp_name = || db_user;
End;

Listing 4.3. Example of VPD function.

where userenv is the pre-defined application context attribute that is set to the SESSION_USER by the DBMS.

The policy to attach this function to the employee_data table is shown in Listing 4.4.

DBMS_RLD.add_policy 
  (object_schema => Sch ,
   object_name => employee_data ,
   policy_name => sal_policy ,
   policy_function => example_func ,
   statement_types => select );

Listing 4.4. Example of VPD policy.

Now suppose that Bob accesses the table with the SQL statement ‘Select sal from employee_data’. The query is dynamically re-written by the Oracle DBMS as ‘Select sal from employee_data where user = Bob’ because of the VPD policy attached to the employee_data table. Note that it is also possible to restrict the insert and update commands using a VPD policy. To restrict Bob from updating any other employee’s salary other than its own, simply add insert as the statement_type in the VPD Policy as shown in Listing 4.5.

DBMS_RLD.add_policy 
  (... 
   statement_types => select , insert);

Listing 4.5. Example of VPD policy with insert.

With such policy in force, the DBMS will allow an SQL statement from Bob such as ‘Insert into employee_data values(‘Bob’, 20000)’ but will disallow a statement like ‘Insert into employee_data values (‘Alice’, 2383)’.
4.2 Oracle Virtual Private Database

Note that it is also possible to associate multiple VPD policies with a database object. In such scenario, the policies are enforced with the AND syntax.

4.2.2 Column-Level VPD

The Oracle VPD mechanism also allows a DBA to control access to the column data within a table or view. The basic idea is that instead of attaching a policy to the whole table or view, the policy is only attached to a subset of the columns called the security-relevant columns. The default behavior of a column-level VPD policy is same as that of a table-level VPD policy, that is, to restrict the number of rows returned for a query that references the security-relevant columns. There is another behavior, however, called the masking behavior in which all rows are returned but they contain NULL values for the security-relevant columns. In what follows, we provide an example of a column-level VPD policy.

Suppose Bob has access to the table Employees(id int, name char, salary int) whose data are shown in Table 4.3.

The business requirement is that the employees should be able to access all ids and names without any restrictions but should be able to access only their own salary figures. To support such requirement, we first create a policy function as shown in Listing 4.6.

Table 4.3. Sample data of the Employees table.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alice</td>
<td>2000</td>
</tr>
<tr>
<td>200</td>
<td>Bob</td>
<td>5000</td>
</tr>
<tr>
<td>300</td>
<td>Tim</td>
<td>10000</td>
</tr>
</tbody>
</table>

Create function sec_function(p_schema char, p_obj char)
Return char
As user CHAR;
Begin
  user := SYS_CONTEXT(userenv, SESSION_USER);
  return name = || user;
End;
Listing 4.6. Example function for a Column-level VPD policy.
We then attach the `sec_function` to the Column-level VPD policy with *default behavior* as shown in Listing 4.7.

```sql
DBMS_RLD.add_policy(
    object_schema => Sch ,
    object_name => Employees ,
    policy_name => my_policy ,
    function_schema => Sch ,
    policy_function => sec_function ,
    sec_relevant_cols => salary );
```

Listing 4.7. Example Column-level VPD policy (default behavior).

With such policy in place, the SQL statements from Bob and their results are shown in Table 4.4.

To observe the *masking behavior* of a Column-level VPD policy, we modify the policy as shown in Listing 4.8.

```sql
DBMS_RLD.add_policy(
    ....
    sec_relevant_cols => salary ,
    sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS ) ;
```

Listing 4.8. Example Column-level VPD policy (masking behavior).

With masking behavior in operation, the resulting rows for the SQL statement `{Select id, name, salary from Employee}` issued by Bob are modified as shown in Table 4.5.

### 4.3 Labeled Oracle

Oracle Label Security is an Oracle product (referred here as Labeled Oracle) that supports mandatory access control with the granularity of

<table>
<thead>
<tr>
<th>Table 4.4. Column-level VPD policy in action (default behavior).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select id, name from Employee</td>
</tr>
<tr>
<td>id     name</td>
</tr>
<tr>
<td>100    Alice</td>
</tr>
<tr>
<td>200    Bob</td>
</tr>
<tr>
<td>300    Tim</td>
</tr>
<tr>
<td>Select id, name, salary from Employee</td>
</tr>
<tr>
<td>id     name  salary</td>
</tr>
<tr>
<td>200    Bob     5000</td>
</tr>
</tbody>
</table>
Table 4.5. Column-level VPD policy in action (masking behavior).

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alice</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Bob</td>
<td>5000</td>
</tr>
<tr>
<td>300</td>
<td>Tim</td>
<td></td>
</tr>
</tbody>
</table>

the tuple; therefore different access classes, called *labels* in Oracle, can be associated with single tuples in the same relation. Labeled Oracle enables tuple-level access control based on the VPD mechanism (see Section 4.2). It controls access to the contents of a tuple by comparing that tuple’s label with a user’s label and privileges.

Access control in Labeled Oracle is based on three factors:

- The label of the tuples to which access is requested.
- The label of the user session requesting access.
- The authorizations that the user has.

Notice thus that Labeled Oracle combines mandatory access control with discretionary access control based on authorizations users receive on the data and fine-grained restrictions, based on the VPD mechanism. The diagram in Figure 4.3 shows the flow of access control.

Labels in Labeled Oracle consist of three components:

- a single level (sensitivity) ranking;
- zero or more horizontal compartments or categories;
- zero or more hierarchical group statements.

Therefore, compared with the labels of the Bell–LaPadula model (see Section 2), Oracle labels include an additional component. Notice that the compartment and group components may be missing, whereas the sensitivity component must always be present. If compartments are specified, then a user whose level would normally permit access to a tuple’s data will be forbidden from accessing this tuple unless the user’s label also contains all the compartments appearing in that tuple’s label. The group component is hierarchical and is used to reflect ownership. Suppose, for example, that an organization has
two groups of users: Administration and Engineering. Users with the label Administration cannot access data labeled Engineering (and vice versa), because they are “at the same level”. Suppose that one has a group Board of Directors (BoD). Users in this group must be allowed to access the data of both Administration and Engineering group. To this end, one can establish a group hierarchy, where BoD is the super-group of Administration and Engineering group. Figure 4.4 shows examples of labels.

A label can be any one of the following four combinations of components:

- a single level component, with no groups or compartments, such as U::;
- a level and a set of compartments with no groups, such as U:Alpha, Gamma:;
- a level and a set of groups with no compartments, such as U::ADM, EUROPE;
- a level with both compartments and groups, such as U:Alpha, Psi:EUROPE, ADM.
A user label specifies that user’s sensitivity level plus any compartments and groups that constrain the user’s access to labeled data. Each user is assigned a range of levels, compartments, and groups, and in each session can operate within that authorized range to access labeled data within that range. The session label is the particular combination of level, compartments, and groups at which a user works at any given time. The user can change the session label to any combination of components for which s/he is authorized. When a user writes data without specifying its label, a tuple label is assigned automatically, using the user’s session label. Because of the complexity of Oracle labels, the task of assigning labels to users and data can be quite cumbersome. To address this issue, Labeled Oracle provides graphical user interfaces to administer labels.

An additional interesting element of the Labeled Oracle access control mechanism is the notion of policy privileges. The policy privileges enable a user or a stored program unit to bypass certain parts of the MAC and DAC policies. An administrator can also authorize the user or program unit to perform specific actions, such as the ability of one user to assume the authorizations of a different user. Privileges can be granted to program units, authorizing the procedure, rather than the user, to perform privileged operations. For example, a user with the READ privilege can read all data protected by the policy, regardless of his discretionary authorizations or session label. The user does not
even need to have label authorizations. Such policy privilege is useful for system administrators who need to export data, but who should not be allowed to change data. The FULL privilege has the same effect and benefits as the READ privilege, with one difference: a user with FULL privilege can also write all the data. The COMPACCESS is another example of policy privilege that allows a user to access data based on the tuple label’s compartments, independent of the tuple label’s groups. If a tuple label has no compartments, then access is determined by the group authorizations. However, when compartments do exist, and access to them is authorized, then the group authorization is bypassed.

In addition to these policy privileges, Labeled Oracle provides special privileges that allows users to manipulate data labels, for example for downgrading data. The access control enforcement logic is further complicated by the presence of these policy privileges, which however are necessary in order to achieve the flexibility required in practice. We refer the reader to the Oracle documentation [68] for details about the access control logic and the Labeled Oracle access control model.

### 4.4 Summary

In this section, we have presented the different flavors of an access control system as implemented in some of the commercial DBMSs. We first presented the core access control mechanism implemented in Microsoft’s SQL Server 2008. We then described the Virtual Private Database technology in Oracle DBMS that supports fine-grained access to the data. Finally, we presented the Oracle Label security mechanism that implements the concept of mandatory access control policies in an Oracle database.
Fine-Grained Access Control Models and Mechanisms

Conventional database management systems rely on authorization mechanisms which enforce access control at the level of tables, columns or views. However, such a model represents a rather coarse-grained level of authorization. In practice, many applications require authorization to be granted differently for individual database tuples, or even for individual cells within a tuple. Such requirements call for novel approaches to access control, and bring several difficult challenges such as ensuring correctness, security and scalability of query execution. In addition, the administration of flexible authorization policies and large populations of users and data objects must be achievable in an effective manner.

Consider, for instance, the example of a human resources (HR) application, where each database user is allowed to access only his or her own salary record. Currently, applications enforce such a constraint through code that is deployed at the application layer, outside the boundaries of the DBMS. Therefore, authorization is performed by the database client applications. Given a user query, the client code retrieves from the database the entire set of matching tuples, and subsequently filters out at the client side the tuples that the current user
is not authorized to access. However, this approach has several serious drawbacks:

1. Data security may be breached if the client application is compromised. For instance, the client code may contain bugs that allow an attacker to bypass the authorization mechanism. Furthermore, if the user has administrative privileges on the client machine, s/he may be able to retrieve the data downloaded from the database either by intercepting the packets sent over the network, or by inspecting the contents of the application memory. If successful, such an attack discloses the contents of all tuples in the result set to the adversary. Ideally, only data items (e.g., tuples, cells) for which authorization exists should be released outside the boundaries of the DBMS.

2. Deploying authorization code in the client application poses serious administration issues when changes to access control policies occur. Typically, a single database may be accessed from many applications, which makes it difficult to effectively manage modifications. An outdated client can lead to problems ranging from application errors due to mismatch of policies, to unauthorized disclosure of confidential data.

3. Emerging computing paradigms, such as cloud computing, envision large-scale outsourcing of data storage to service providers. In such an environment, a single database (and even a single table) may contain data owned by several distinct entities, such as individuals, companies, organizations, etc. Application-level authorization enforcement does not properly support such scenarios. Specifically, if authorization is enforced at the application level, then a curious entity would be able to download all information in the database to the local client application, breaching data security.

To enforce fine-grained access control, it is necessary to define for each request a secure context that encapsulates information related to the executed query. The secure context may include information about
the identity of the user (or application) on behalf of whom the query is executing, the network address of the client where the query originated, etc. The authorization mechanism will subsequently use the context to evaluate whether authorization for query execution should be granted or not. Current DBMSs do provide such support for secure contexts through system functions such as `userId()` that can be accessed through a well-defined programming interface. Revisiting the earlier HR application example, the constraint enforcing access to salary information could be expressed with the help of the `userId()` function as follows:

```sql
SELECT salary FROM employees
WHERE (empid = userId());
```

In fact, authorization based on such system functions is already available in several commercial products, such as Oracle and Sybase. In Oracle’s Virtual Private Database (VPD) authorization mechanism, each relation has a set of access control functions that are specified by the administrator. Separate functions can be specified for different types of access (e.g., SELECT, INSERT, DELETE). The functions take as argument elements from the secure query context, and return predicates which are logically “AND”-ed with the WHERE condition of the SQL query. In the case of data items for which access is not allowed, VPD may eliminate tuples from the result set altogether, or replace unauthorized cells with special NULL values.\(^1\) Similarly, Sybase allows different predicates to be specified for each table column, and the predicates are appended to the WHERE condition of the queries at execution time. However, as we will discuss shortly, such approaches may have drawbacks that affect the correctness of results and may also have a negative impact on the scalability of query execution and authorization administration.

In this section, we review recent work that addresses the problem of fine-grained database authorization from several distinct perspectives: query rewriting [54, 86], SQL language extensions [24] and transparent authorization views [76].

---

\(^1\)For further details on Oracle VPD, please refer to Section 4.2.
5.1 Fine Grained Access Control through Query Rewriting

We review two prominent approaches that address the problem of fine-grained database authorization from the perspective of query rewriting. The work in [54] addresses authorization in the context of databases storing sensitive data about individuals, such as medical records, where the owner of each record has the authority to specify to whom each data tuple may be disclosed and for what purposes. Several disclosure semantics are proposed, which we review in Section 5.1.1. However, for certain types of queries, these semantics may sacrifice the soundness of the query results for the sake of data privacy. To alleviate this problem, the work in [86] identifies several essential properties that a method for fine-grained authorization must satisfy. We review these properties together with a labeling-based approach that satisfies them in Section 5.1.2.

5.1.1 Hippocratic Databases

Hippocratic databases contain sensitive data about individuals, and typically each data tuple is associated with a single person that owns the data. Databases that store health records are a representative example of hippocratic databases. The hippocratic databases domain presents two important challenges that make the authorization problem difficult. First, the access control to each data tuple is dictated by its owner. As a consequence, there is often no single, concise policy that dictates the conditions under which authorization is granted. Instead, the owner of each data record has the option to specify to which subjects access control will be granted, as well as to restrict the access based on the purpose for which the data are used, e.g., medical research, marketing of experimental drugs, etc. Second, it may not be sufficient to simply block access to unauthorized data; it is also essential to ensure that the users are not able to derive private information from the fact that their access to the data was denied.

The work in [54] assumes that each record comprises of multiple categories of data, and that these categories are disjoint. For instance, certain attributes such as Age belong to the category of personal information, whereas Address may belong to the category of billing
5.1 Fine Grained Access Control through Query Rewriting

Information. Data owners are allowed to specify a separate access control preference for each data category.

In the hippocratic database model, there are two separate sets of records: the actual data records, and the authorization metadata, which comprises of the owner-specified rules that govern data access. Each query has an associated context, which encapsulates information about the identity of the subject asking the query, as well as the purpose for which the data are requested. Before the query result is returned to the user, a filtering step enforces the access control policy specified by the authorization metadata.

Consider the example database in Figure 5.1. The actual hospital records are shown in Figure 5.1(a). The ID attribute is the primary key in the table. The remaining fields belong to three categories that comprise of disjoint sets of attributes: personal information (attribute Age), billing information (attributes Name, City and Zipcode) and diagnosis (attribute Disease).

Figures 5.1(b) and 5.1(c) capture the authorization metadata for two usage purposes, namely medical research and billing, respectively. Each disclosure metadata table has one record for every data record. Note that, the requirements of distinct data owners can be quite

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>City</th>
<th>Zipcode</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alexander</td>
<td>49</td>
<td>Indianapolis</td>
<td>38921</td>
<td>Gastritis</td>
</tr>
<tr>
<td>2</td>
<td>Charles</td>
<td>54</td>
<td>Chicago</td>
<td>62131</td>
<td>Hepatitis</td>
</tr>
<tr>
<td>3</td>
<td>Elizabeth</td>
<td>68</td>
<td>New York</td>
<td>23192</td>
<td>Flu</td>
</tr>
</tbody>
</table>

(a) Medical Records

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>City</th>
<th>Zipcode</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Deny</td>
<td>Deny</td>
<td>Permit</td>
</tr>
<tr>
<td>Deny</td>
<td>Deny</td>
<td>Deny</td>
<td>Deny</td>
<td>Deny</td>
<td>Deny</td>
</tr>
<tr>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit</td>
</tr>
</tbody>
</table>

(b) Authorization Metadata for Medical Research Usage

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>City</th>
<th>Zipcode</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Permit</td>
<td>Deny</td>
</tr>
<tr>
<td>Permit</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Permit</td>
<td>Deny</td>
</tr>
<tr>
<td>Permit</td>
<td>Permit</td>
<td>Deny</td>
<td>Permit</td>
<td>Permit</td>
<td>Permit</td>
</tr>
</tbody>
</table>

(c) Authorization Metadata for Billing Usage

Fig. 5.1 Fine grained authorization in hippocratic databases.
different, even within the authorization specification for the same purpose. Furthermore, data owners have the ability to specify authorization for the primary key: if the access to the primary key is denied, then the rest of the tuple is also denied.

For medical research purposes, it is important to release diagnostic information, as well as certain personal data that may be relevant for the treated disease. For instance, Alexander allows researchers to access his age, as this could be a risk factor for gastritis. However, he does not disclose address information, as he fears that a motivated adversary may be able to trace his actual identity using city and zipcode information. On the other hand, Elizabeth, who was treated for flu (a minor condition), is not concerned that her identity may be indirectly learned, so she is allowing researchers access to address information as well (she still disallows access to the name attribute). Charles denies access to his record for medical research altogether. For billing purposes, all patients release name and address information, but only Elizabeth releases the disease as well.

In addition to purpose, owners can decide authorization based on data recipient. For instance, separate metadata tables can be defined for medical research conducted by universities, as opposed to drug companies. In general, the number of required metadata tables is \( m = r \times p \), where \( r \) denotes the number of recipient types and \( p \) the number of usage purposes. The metadata is stored in the database in the form of additional columns associated with the actual data. If there are \( c \) distinct categories of data (i.e., attribute classes), then a number of \( c \times m \) columns (corresponding to all combinations of purpose-recipient-category) need to be maintained.

At runtime, the DBMS extracts the necessary information from the secure context and rewrites the query in accordance to the permissions mandated by the data owners. To protect from unauthorized data access, the authors of [54] propose three types of disclosure semantics, as follows:

- **Strict cell-level enforcement.** This is the simplest disclosure model: every database table \( T \) in the `FROM` clause of the query is replaced with a copy \( T' \) in which every cell for which access is denied is replaced with a special “unauthorized”
5.1 Fine Grained Access Control through Query Rewriting

mark (e.g., *). Subsequently, the query is executed on top of \( T' \). Consider that the following query is issued for medical research purposes:

\[
\text{SELECT * FROM patients;}
\]

The result of the query is shown in Figure 5.2(a). This disclosure model is very easy to enforce, but can lead to situations where the results do not abide the relational database model, e.g., the record corresponding to Charles has no primary key value.

- **Table semantics disclosure model.** This model addresses the relational model violations that may occur in the cell-level enforcement model. The table semantics model requires all records for which the primary key is unauthorized to be removed from the query processing. Revisiting the case in Figure 5.2(a), the second record would not be considered in any query for medical research, so the result to the previous query would conform to the relational model. Still, if the following projection query is issued:

\[
\text{SELECT City, Zipcode FROM patients;}
\]

the result obtained (Figure 5.2(b)) still contains entries with all cells unauthorized.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>City</th>
<th>Zipcode</th>
<th>Disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>49</td>
<td>*</td>
<td>*</td>
<td>Gastritis</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>68</td>
<td>New York</td>
<td>23192</td>
<td>Flu</td>
</tr>
</tbody>
</table>

(a) Strict cell-level enforcement

<table>
<thead>
<tr>
<th>City</th>
<th>Zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>New York</td>
<td>23192</td>
</tr>
</tbody>
</table>

(b) Table semantics disclosure model

<table>
<thead>
<tr>
<th>City</th>
<th>Zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>23192</td>
</tr>
</tbody>
</table>

(c) Query semantics disclosure model

Fig. 5.2 Data disclosure semantics in hippocratic databases.
• **Query semantics disclosure model.** In this model, instead of first creating the filtered tables and then answering the query on top of the modified data, the query is processed first using the non-altered data, and strict cell-level enforcement is applied to the query result. The authorization for each field in the result is inherited from the original table authorization. For the earlier query, the result is shown in Figure 5.2(c).

### 5.1.2 Formal Correctness Criteria for Query Rewriting

The work in [54] proposed several alternatives for disclosing data in the presence of fine-grained authorization, and discussed how this can be accomplished using query rewriting techniques. However, no formal considerations about the soundness and completeness of the proposed fine-grained authorization mechanism have been given.

The work in [86] focuses on a formal treatment of fine-grained authorization, and provides several use cases in which the query rewriting mechanisms proposed in [54] can lead to undesirable outcomes in conjunction with certain types of queries. Denote by $A$ a mechanism (or algorithm) for answering queries in the presence of fine-grained authorization constraints, and let $S$ be the conventional query answering mechanism in the absence of access control (i.e., users are allowed unrestricted access to the data.) According to [86], there are three objectives that a fine grained authorization model should achieve:

1. **Soundness.** For any given query $Q$, the query result returned by $A$ should be a subset of the result returned by $S$. In other words, the information presented to the user should be *correct*. Even if not all information returned by $S$ can be made accessible to the user, the mechanism $A$ should at least not return *wrong* information. An example will illustrate shortly how some of the disclosure models in [54] violate the soundness requirement.

2. **Security.** The authorization mechanism $A$ should be secure, meaning that no information to which access is denied by the policy should be disclosed to the user.
5.1 Fine Grained Access Control through Query Rewriting

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linda</td>
<td>32(P)</td>
<td>111-1111(P)</td>
</tr>
<tr>
<td>2</td>
<td>Mary</td>
<td>29(P)</td>
<td>222-2222(P)</td>
</tr>
<tr>
<td>3</td>
<td>Nick</td>
<td>34(D)</td>
<td>333-3333(D)</td>
</tr>
<tr>
<td>4</td>
<td>Jack</td>
<td>21(P)</td>
<td>444-4444(P)</td>
</tr>
<tr>
<td>5</td>
<td>Mary</td>
<td>30(D)</td>
<td>555-5555(D)</td>
</tr>
</tbody>
</table>

(a) Original Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda</td>
<td>111-1111</td>
</tr>
<tr>
<td>Mary</td>
<td>222-2222</td>
</tr>
<tr>
<td>Nick</td>
<td>*</td>
</tr>
<tr>
<td>Jack</td>
<td>444-4444</td>
</tr>
<tr>
<td>Mary</td>
<td>*</td>
</tr>
</tbody>
</table>

(b) Q1 Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nick</td>
<td>*</td>
</tr>
<tr>
<td>Jack</td>
<td>444-4444</td>
</tr>
</tbody>
</table>

(c) Q2 Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda</td>
<td>111-1111</td>
</tr>
<tr>
<td>Mary</td>
<td>222-2222</td>
</tr>
</tbody>
</table>

(d) Q1/Q2 Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
<td>444-4444</td>
</tr>
</tbody>
</table>

(e) Correct Result

Fig. 5.3 Table semantics may violate soundness property.

(3) **Maximality.** A straightforward way to achieve both soundness and security would be not to return any result to the user. Clearly, such an approach is undesirable, since the database system would no longer provide any useful function. The authorization mechanism $A$ should return to the user as much information as possible, provided that the soundness and security requirements are met.

The following example shows why the approach in [54] does not satisfy the soundness property. Consider the table in Figure 5.3 where the authorization permissions Permit(P)/Deny(D) are included in parentheses for each data cell. Assume that the following query is issued:

```sql
SELECT name, phone FROM customer
MINUS
SELECT name, phone FROM customer
WHERE age>25;
```

If we denote by $Q_1$ and $Q_2$ the individual select statements, then the above query should return the difference between $Q_1$ and $Q_2$. According
to the table semantics model described in Section 5.1.1, the results of
$Q_1$ and $Q_2$ are as shown in Figure 5.3(b) and (c). The final result of
the query, shown in Figure 5.3(d), contains an additional tuple cor-
responding to Nick, which is not present in the result returned if no
access control restrictions were in place, shown in Figure 5.3(e). Note
that, the age of Nick is 34, which is greater than 25 therefore the result
obtained using the table semantics is incorrect.

The work in [86] uses a format for data disclosure similar to [54],
where tuples are omitted from the results altogether, or the contents
of individual cells are replaced with a special unauthorized marker.
However, a novel mechanism for authorization enforcement is proposed,
which relies on labeling to address the objectives of soundness, security
and maximality.

From the earlier example, it becomes clear that incorrect behav-
ior may arise when the same information is not consistently treated
when it appears in distinct objects. For instance, in the MINUS query
discussed earlier, the same tuples would appear in the partial results
of distinct sub-queries, but the information linking them was lost due
to masking. Another common case where incorrect results may arise
is that of join queries, where a foreign key may not be matched with
its corresponding value in a table where it is a primary key due to
unauthorized contents. The key observation of the labeling approach is
that even though unauthorized information must not be disclosed in
the final results, data relationships can be preserved during the inter-
mediate result processing.

Figure 5.4 illustrates how the labeling approach works for the same
difference query example considered earlier. During query processing,
the masking of tables proceeds as follows: for each table $T_i$, each cell
that is unauthorized is replaced with a unique random label. If the cell
is a foreign key in another table, the same value is chosen to represent
the same cell in distinct tables. This way, a secure link between tables
is created, which ensures that join operations are performed correctly.
The labels can be interpreted as a form of surrogate keys that allow
correct processing without disclosing the actual value of confidential
attributes. Figure 5.4(a) shows the labeling obtained for the consid-
ered example. The results of the two projection queries $Q_1$ and $Q_2$ are
5.2 SQL Language Extensions for Fine Grained Access Control

The work in [24] focuses on extending the SQL language specification to allow for fine-grained access control operations, such as column-level and cell-level authorization, as well as control of function and procedure execution. The main idea is to extend the existing access control grant mechanisms to handle conditions that are expressed as a function of the query execution context. Such clauses are called \textit{predicated grants}. The predicates are specified in a parametric fashion when the policies are created, and are evaluated at runtime to decide which set of tuples in the result set will be returned to the user. The actual technique of enforcement is similar to query rewriting, but the SQL language extension is specified at a declarative level, therefore no restrictions are placed on the mechanisms used to implement predicated grants.
The resulting language is a strict generalization of existing SQL authorization, and it allows for concise specification of complex access control requirements. In particular, it is powerful enough to subsume well-established models such as access control lists and multi-level security.

Similar to other techniques for fine-grained access control, the predicated grants mechanism assumes the existence of a context within which a query is executed. A predicated grant has the general form:

```
GRANT <OPERATION_TYPE> ON R
   WHERE P
   TO U;
```

where `OPERATION_TYPE` represents typical SQL operations such as select, insert, delete, or update, `P` specifies the condition on which the grant depends, and `U` is the user (or set of users) to which the grant statement applies. Note that, conventional SQL grants are special cases of predicated grants where `P` is always true, and `U` is set to `PUBLIC`, a constant that represents all users in the system.

The effect of such a predicated grant is called the `filter` semantics and is identical to the case of user `U` issuing the following SQL query:

```
SELECT * FROM R WHERE P;
```

Revisiting the earlier example of an HR application where each employee is allowed to access his or her data only, consider the following `Employee` table, with the schema

```
Employee (empid, name, deptid, dept)
```

and assume that the DBMS provides a built-in function `userId()` that extracts the identifier of the user that initiated the current query execution context. Then, a predicated grant that enforces the required access constraint to the employee table can be expressed as:

```
GRANT SELECT ON EMPLOYEE
   WHERE (empid = userId())
   TO PUBLIC;
```
5.2.1 Authorization on Columns

In addition to tuple-level filtering, the authorization model in [24] allows the specification of predicated grants at column-level. For instance, the following authorization statement

\[
\text{GRANT SELECT ON R(A) WHERE P TO PUBLIC;}
\]

allows access only to attribute A of those tuples that satisfy predicate P. Furthermore, the same user may be granted authorization on distinct columns A and B with two different predicates P1 and P2. Note that, a query may attempt to access both columns A and B. In such a case, several authorization semantics may be considered:

1. if a grant authorization exists covering both columns A and B with predicate P, then the query is authorized to return tuples that satisfy P, otherwise deny access
2. if different grants exist with predicates P1 and P2 that cover columns A and B respectively, then the query is authorized to return only tuples for which (P1 AND P2) holds
3. if different grants exist with predicates P1 and P2 that cover columns A and B respectively, then the query is authorized to return only tuples for which (P1 OR P2) holds, and nullify values in the tuple columns A, respectively B for which predicate P1, respectively P2 do not hold

The latter case is specified through a special NULLIFY clause, as follows:

\[
\text{GRANT SELECT ON R(A) WHERE P ELSE NULLIFY TO PUBLIC;}
\]

5.2.2 Making Fine-grained Authorization Scalable

This section reviews query-defined user groups and authorization groups: two mechanisms that facilitate the administration of
authorization and allow scalability to a large number of users and complex access control policies. The former mechanism applies to users with similar privileges, whereas the latter applies to related objects that are often part of the same transaction, e.g., a purchase order.

So far, grants have been assigned to all system users using the PUBLIC identifier. However, doing so may incur significant performance deterioration, since for each arriving query, a large set of grants must be evaluated (regardless of user identity). In practice, grants are usually assigned to specific sets of users with a certain privilege. For instance, consider the following managers table with the schema

\[
\text{managers (mgrid, deptid)}
\]

The following grant statement authorizes department managers to access the records of all the employees in their departments:

\[
\text{GRANT SELECT ON EMPLOYEE E}
\]
\[
\text{WHERE (E.deptid IN}
\]
\[
(\text{SELECT deptid FROM managers WHERE mgrid = userId()})
\]
\[
\text{TO PUBLIC;}
\]

Note that, the grant still applies to PUBLIC, therefore it will be evaluated even for queries issued by employees who are not managers, although the predicate can only hold for managers.

One workaround is to create a group of users corresponding to managers, and explicitly assign users who are managers to that group. Then, the grant can be assigned to that group, instead of all users. However, maintaining the group incurs high administration cost, and is not necessary since the information about which users are managers is already contained within the database. To facilitate administration, dynamic user groups defined by database queries are employed. Such groups are specified as follows:

\[
\text{CREATE GROUP managerGrp AS}
\]
\[
(\text{SELECT mgrid FROM manager});
\]

Subsequently, predicated grants can be assigned to groups. The earlier authorization condition that managers can only access the records of
employees in their department can be re-written as:

```
GRANT SELECT ON EMPLOYEE
  WHERE (deptid IN
    (SELECT deptid FROM managers WHERE mgrid = userId()))
  TO managerGrp;
```

Note that, in this case, the query filter associated with this grant will only be executed for queries that are issued by managers, as opposed to all users, improving overall performance. Furthermore, administration is facilitated by allowing roles to be assigned to query-defined groups, without need for explicit group membership maintenance.

In addition to managing users, another challenging task is setting the proper authorizations for data objects. Often, database applications involve transactions such as purchase orders, travel reservations, etc., with well-defined patterns of accessing the data. Each such transaction requires authorization to a predefined set of data objects. To facilitate the management of authorization in a scalable and consistent manner, *authorization groups* allow a single grant clause to be defined for all objects participating in a transaction.

For instance, a purchase order may involve accessing data from several distinct tables, such as orders, parts and suppliers table. The following example shows how authorization can be granted to all such objects at once, such that when a transaction arrives the access permission can be decided upon by checking one single clause.

```
CREATE AUTHORIZATION select_purchaseorder
  WITH root order O AS (
    SELECT ON order O,
    SELECT ON part P WHERE P.part_id = O.part_id
  )
```

In addition to improving performance, authorization groups prevent errors resulting from inconsistencies in individual grant clauses set for distinct tables. If the authorization across different objects is not properly coordinated, legitimate transactions may be denied from accessing the data.
5.3 Fine Grained Access Control with Authorization Views

The work in [76] argues that query rewriting techniques have certain limitations concerning the truthfulness of the results returned to users. Note that, such limitations apply to all reviewed query rewriting techniques, as well as to mechanisms such as VPD that create authorization views, but ultimately enforce those views by transparently rewriting queries before execution.

To illustrate the limitation of query rewriting methods, consider the example of aggregate queries in the context of a table with customer records where each customer is allowed to access his/her own record only. Then, the query

\[
\text{SELECT AVG(age) FROM customers;}
\]

will be transparently rewritten as

\[
\text{SELECT AVG(age) FROM customers WHERE cust_id=userId();}
\]

hence the result will return the age of the querying customer, instead of the average customer age. This is misleading for the user, who may not know the details of the authorization policy, and may rely on the information returned by the database, although the information is not correct. To be precise, the information is correct only with respect to the subset of records accessible by the user, which is referred to in [76] as the Truman model.

Note that, it may sometimes be desirable to return no results at all when a query cannot be properly authorized, rather than returning incorrect results, as shown in the above example. The non-Truman model argues that a better approach is to define a number of authorization views, and at runtime attempt to execute the query with respect to the views that the user has authorization for. If the query cannot be answered solely based on such views, it is rejected. This approach guarantees that, whenever a result is returned to the user, it is the correct result with respect to the information present in the entire database.

The mechanism of answering queries is the following: first, the authorization views that apply to the current query are determined,
and the query context is evaluated against the requirements specified by the views. If it is determined that the query is valid, the execution of the query is allowed directly on the data. Otherwise, the query is rejected.

There are two distinct types of query validity: unconditional and conditional. Unconditional validity of a query is independent of the current state of the database, and can be determined by inspecting the syntax of the query and the authorization views only. In contrast, the evaluation of conditional validity takes into account the current database state. Continuing the earlier example, consider the same restriction requiring that customers can access their own record only, but the query is slightly modified as:

\[
\text{SELECT AVG(age) FROM customers} \\
\text{WHERE name='Alice'};
\]

If the name of the user executing the query is Alice, and there is no other user by the name of Alice in the database, then the query is valid (since the average age requested is in fact the age of Alice, as she is the only one matching the \texttt{WHERE} predicate condition). However, in a different database state (where another user by the name Alice exists) the query would not be valid.

The approach from [76] has several benefits, as well as shortcomings compared to other models such as query re-writing and predicated grants. In terms of benefits, the transparent authorization views approach works within the existing SQL model, without requiring specific extensions to the access control rules specification language. In addition, the results returned by queries are always sound.

On the other hand, the approach has the disadvantage of being too restrictive compared to alternative solutions. For instance, in many cases a large amount of information can be returned to the user if the unauthorized cells are masked, as in the approaches of [54, 86]. However, the non-Truman model would not allow such behavior, and queries would be rejected. Finally, the evaluation of the query validity is expensive, especially in the case of conditional validity. In fact, the mechanism for validity inference is not decidable in general, requiring heuristic implementations that may lead to unpredictable outcomes.
5.4 Summary

Fine-grained authorization allows specification of access control policies at the tuple or cell level, and can be enforced through several alternate mechanisms such as query rewriting or transparent authorization views. Specification of access control policies can be done with the help of pre-defined DBMS functions (e.g., `userId`), or by extending the existing SQL authorization language.

An interesting direction for future work is the study of authorization delegation. In addition, some existing solutions for fine-grained access control (e.g., [86]) focus on read access only. Extending authorization mechanisms to support update operations as well is another interesting problem to address.
The decision semantics of a typical access control system is either to permit access to a resource or to deny access based on the state of the access control system. Such coarse semantics is appropriate for most usage scenarios; however it needs to be extended when designing an intrusion response system or a continuous event-based authentication/auditing system. The access control system described in this section extends the decision semantics of access control enforcement with the notions of request suspension and request tainting. Request suspension involves suspending the request till further negotiation with the client is satisfied; request tainting, on the other hand, mandates auditing the in-progress request either in memory or on disk for further analysis. In what follows, we present a novel approach based on the concept of privilege states for achieving fine-grained decision semantics in an access control system.\footnote{The discussion in this section is primarily based on the work published in [48].}

\section{Motivation}

An access control system is typically characterized by the notion of authorizations of the form $\langle A, R, P \rangle$ where $A$ is the set of permissible
actions, $R$ is the set of protected resources, and $P$ is the set of principals that need to be authorized. The basic idea behind such traditional access control mechanism is simple. When a principal tries to access a protected resource, the access control system checks the privileges of the principal against the set of authorizations to determine whether to allow or deny the request. Note the key observation here that the decision semantics of a typical access control mechanism is to either allow or deny the requests. The main goal of this work is to extend the decision semantics of an access control system to allow more fine-grained access decisions. Consider the case of an intrusion detection system where the result of the analysis of an incoming request is not always deterministic. In such cases, the response component of the intrusion detection system may use the services of an access control system providing support for fine-grained decisions to make fine-grained responses such as request suspension and request tainting. During request suspension, further negotiation (such as a second factor of authentication) occurs with the principal before deciding to allow/deny the request. Request tainting, on the other hand, allows one to audit the request in-progress (either in-memory or on disc), thus resulting in further monitoring of the principal, and possibly in the suspension or dropping of subsequent requests by the same principal.

The important question to ask here is why should changes be made to the access control system instead of modifying the intrusion response component itself to provide such fine-grained responses. The rationale for modifying the access control system is as follows: if we model an incoming request as usage of a subset of these authorizations, it is easy to discern that to mark a request for suspension or tainting, it is sufficient to mark the authorizations used by it as suspended or tainted. Since the access control system is responsible for maintaining the set of authorizations, it makes sense to make changes to the access control system itself to provide support for suspended and tainted authorizations. Implementing this functionality in the intrusion response component shall involve duplication of maintaining the set of authorizations at the intrusion response layer as well.

Thus, the implementation of the extended decision semantics requires an access control model that supports request suspension and
request tainting. Such a model, developed in the context of a database management system (DBMS), is as follows. A request from a principal is modeled as the usage of a set of privileges in the DBMS. For example, the SQL query ‘SELECT * FROM orders, parts’ is modeled as using the select privilege on orders and parts tables. To support the extended decision semantics, we introduce the notion of privilege state. A privilege, assigned to a principal on a DBMS object, now has a state attached to it, thereby resulting in a privilege state based access control (PSAC) system. Under this approach, a privilege can exist in five states: unassign, grant, taint, suspend, and deny. The privilege state semantics and their transitions are discussed in detail in Section 6.2.

The PSAC model has been developed in the context of a role based access control (RBAC) system [78]. Extending PSAC with roles presents the main challenge of state conflict resolution, that is, deciding on the final state of a privilege when a principal receives the same privilege in different states from other principals. Moreover, additional complexity is introduced when the roles are arranged in a hierarchy where the roles higher-up in the hierarchy inherit the privileges of the lower level roles.

### 6.2 Design and Implementation

In this section, we introduce the design and the formal model underlying PSAC. In what follows, we first introduce the privilege state semantics and state transitions. We then discuss in detail how those notions have to be extended when dealing with role hierarchies.

#### 6.2.1 Privilege States Dominance Relationship

PSAC supports five different privilege states that are listed in Table 6.1. For each state, the table describes the semantics in terms of the result of an access check.

A privilege in the unassign state is equivalent to the privilege not being assigned to a principal; and a privilege in the grant state is equivalent to the privilege being granted to a principal. The deny state supports the concept of negative authorizations in which a privilege is specifically denied to a principal [18]. The suspend and the taint states
Table 6.1. Privilege states.

<table>
<thead>
<tr>
<th>State</th>
<th>Access check result semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unassign</td>
<td>The access to the resource is not granted.</td>
</tr>
<tr>
<td>Grant</td>
<td>The access to the resource is granted.</td>
</tr>
<tr>
<td>Taint</td>
<td>The access to the resource is granted; the system audits access to the resource.</td>
</tr>
<tr>
<td>Suspend</td>
<td>The access to the resource is not granted until further negotiation with the principal is satisfied.</td>
</tr>
<tr>
<td>Deny</td>
<td>The access to the resource is not granted.</td>
</tr>
</tbody>
</table>

support the fine-grained decision semantics for the result of an access check.

In most DBMSs, there are two distinct ways according to which a user/role\(^2\) can obtain a privilege \(p\) on a database object \(o\):

- **(1)** Role-assignment: the user/role is assigned a role that has been assigned \(p\).
- **(2)** Discretionary: the user is the owner of \(o\); or the user/role is assigned \(p\) by another user/role that has been assigned \(p\) with the GRANT option.\(^3\)

Because of the multiple ways by which a privilege can be obtained, conflicts are natural in cases where the same privilege, obtained from multiple sources, exists in different states. Therefore, a conflict resolution strategy must be defined to address such cases. A possible strategy is to introduce a privilege states dominance (PSD) relation (see Figure 6.1). The PSD relation imposes a total order on the set of privilege states such that any two states are comparable under the PSD relation. Note the following characteristics of the semantics of the PSD relation. First, the **deny** state overrides all the other states to support the concept of a negative authorization [18]. Second, the **suspend**, and the **taint** states override the **grant** state as they can be triggered as potential response actions to an anomalous request. Finally, the **unassign** state is overridden by all the other states thereby preserving the traditional semantics of privilege assignment.

\(^2\)From here on, we use the terms principal and user/role interchangeably.

\(^3\)A privilege granted to a principal with the GRANT option allows the principal to grant that privilege to other principals [44].
The PSD relation is the core mechanism that PSAC provides for resolving conflicts. For example, consider a user $u$ that derives its privileges by being assigned a role $r$. Suppose that a privilege $p$ is assigned to $r$ in the grant state. Now suppose we directly deny $p$ to $u$. The question is which is the state of privilege $p$ for $u$, in that $u$ has received $p$ with two different states. Such conflicts in PSAC are resolved using the PSD relation. Since in the PSD relation the deny state overrides the grant state, $p$ is denied to $u$.

### 6.2.2 Privilege State Transitions

We now turn our attention to the privilege state transitions in PSAC. Initially, when a privilege is not assigned to a principal, it is in the unassign state for that principal. Thus, the unassign state is the default (or initial) state of a privilege. The state transitions can be triggered as internal response actions by an intrusion detection system, or as ad-hoc administrative commands. In what follows, we discuss the various administrative commands available in PSAC to trigger privilege state transitions.
The GRANT command is used to assign a privilege to a principal in the *grant* state whereas the REVOKE command is used to assign a privilege to a principal in the *unassign* state. In this sense, these commands support similar functionality as the SQL-99 GRANT and REVOKE commands [44]. The DENY command assigns a privilege to a principal in the *deny* state. As part of PSAC, two new commands are introduced, namely SUSPEND and TAIN, for assigning a privilege to a principal in the *suspend* and the *taint* states, respectively. The privilege state transitions are summarized in Figure 6.2. Note the constraint that a privilege assigned to a principal on a DBMS object can only exist in one state at any given point in time.

6.2.3 Formal Model

In this section, we formally define the privilege model for PSAC in the context of a DBMS. The model is based on the following relations and functions.

Fig. 6.2 Privilege state transitions.
6.2 Design and Implementation

6.2.3.1 Relations

(1) \( U \), the set of all users in the DBMS.
(2) \( R \), the set of all roles in the DBMS.
(3) \( PR = U \cup R \), the set of principals (users/roles) in the DBMS.
(4) \( OT \), the set of all DBMS object types such as server, database, schema, table, and so forth.
(5) \( O \), the set of all DBMS objects of all object types.
(6) \( OP \), the set of all operations defined on the object types in \( OT \), such as select, insert, delete, drop, backup, disconnect, and so forth.
(7) \( S = \{ \text{deny, suspend, taint, grant, unassign} \} \), a totally ordered set of privilege states under the PSD relation.
(8) \( P \subseteq OP \times OT \), a many-to-many relation on operations and object types representing the set of all privileges. Note that not all operations are defined for all object types. For example, tuples of the form \((\text{select, server})\) or \((\text{drop, server})\) are not elements of \( P \).
(9) \( URA \subseteq U \times R \), a many-to-many user to role assignment relation.
(10) \( PRUPOSA \subseteq PR \times U \times P \times O \times S \), a principal to user to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to users (assignees) by other principals (assigners) on DBMS objects.\(^4\)
(11) \( PRROSA \subseteq PR \times R \times P \times O \times S \), a principals to role to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to roles (assignees) by principals (assigners).

\(^4\)In PSAC, a role can also be an assigner of privileges. Consider a situation when a user \( u \) gets a privilege \( p \) (with grant option) through assignment of role \( r \). If \( u \) grants \( p \) to some other user \( u' \), PSAC records \( p \) as being granted to \( u' \) by \( r \) even though the actual GRANT command was executed by \( u \).
These relations capture the state of the access control system in terms of the privilege and the role assignments. The functions defined below determine the state of a privilege assigned to a user/role on a DBMS object.

### 6.2.3.2 Functions

1. **assigned_roles (u):** \( U \rightarrow 2^R \), a function mapping a user \( u \) to its assigned roles such that \( \text{assigned_roles} \ (u) = \{ r \in R \mid (u, r) \in URA \} \). This function returns the set of roles that are assigned to a user.

2. **priv_states \((pr, r', p, o)\):** \( PR \times R \times P \times O \rightarrow 2^S \), a function mapping a principal \( pr \) (privilege assigner), a role \( r' \), a privilege \( p \), and an object \( o \) to a set of privilege states such that \( \text{priv_states} \ (pr, r', p, o) = \{ s \in S \mid (pr, r', p, o, s) \in PRRPOSA \} \). This function returns the set of states for a privilege \( p \), that is directly assigned to the role \( r' \) by the principal \( pr \), on an object \( o \).

3. **priv_states \((pr, u', p, o)\):** \( PR \times U \times P \times O \rightarrow 2^S \), a function mapping a principal \( pr \) (privilege assigner), a user \( u' \), a privilege \( p \), and an object \( o \) to a set of privilege states such that \( \text{priv_states} \ (pr, u', p, o) = \{ s \in S \mid (pr, u', p, o, s) \in PRUPOSA \} \cup_{r \in \text{assigned_roles}(u')} \text{priv_states} \ (pr, r, p, o) \). The set of states returned by this function is the union of the privilege state directly assigned to the user \( u' \) by the principal \( pr \), and the privilege states (also assigned by \( pr \)) obtained through the roles assigned to \( u' \).

4. **priv_states \((r, p, o)\):** \( R \times P \times O \rightarrow 2^S \), a function mapping a role \( r \), a privilege \( p \), and an object \( o \) to a set of privilege states such that \( \text{priv_states} \ (r, p, o) = \{ s \in S \mid (r, p, o, s) \in PPROSA \} \). This function returns the set of states for a privilege \( p \), that is directly assigned to the role \( r \) by any principal in the DBMS, on an object \( o \).

5. **priv_states \((u', p, o)\):** \( U \times P \times O \rightarrow 2^S \), a function mapping a user \( u' \), a privilege \( p \), and an object \( o \) to a set of privilege states such that \( \text{priv_states} \ (u', p, o) = \{ s \in S \mid (u', p, o, s) \in PPRPOSA \} \). This function returns the set of states for a privilege \( p \), that is directly assigned to the user \( u' \) by any principal in the DBMS, on an object \( o \).
(pr, u', p, o). This function returns the set of states for a privilege p, that is directly assigned to the user u' by any principal in the DBMS, on an object o.

(6) PSD_state \((2^S)\): \(2^S \rightarrow S\), a function mapping a set of states \(2^S\) to a state in \(S\) such that PSD_state \((2^S) = s' \in 2^S\ |
\forall s \in 2^S|s \neq s' s' \preceq s\). This function returns the final state of a privilege using the PSD relation.

6.2.4 Role Hierarchy

Traditionally, roles can be arranged in a conceptual hierarchy using the role-to-role assignment relation. For example, if a role \(r_2\) is assigned to a role \(r_1\), then \(r_1\) becomes a parent of \(r_2\) in the conceptual role hierarchy. Such hierarchy signifies that the role \(r_1\) inherits the privileges of the role \(r_2\) and thus, is a more privileged role than \(r_2\). However, in PSAC such privilege inheritance semantics may create a problem because of a deny/suspend/taint state attached to a privilege. The problem is as follows. Suppose a privilege \(p\) is assigned to the role \(r_2\) in the deny state. The role \(r_1\) will also have such privilege in the deny state since it inherits it from the role \(r_2\). Thus, denying a privilege to a lower level role has the effect of denying that privilege to all roles that inherit from that role. This defeats the purpose of maintaining a role hierarchy in which roles higher up in the hierarchy are supposed to be more privileged than the descendant roles. To address this issue, PSAC introduces the concept of privilege orientation. Three privilege orientation modes are defined, namely, up, down, and neutral. A privilege assigned to a role in the up orientation mode means that the privilege is also assigned to its parent roles. On the other hand, a privilege assigned to a role in the down orientation mode means that the privilege is also assigned to its children roles; while the neutral orientation mode implies that the privilege is neither assigned to the parent roles nor to the children roles. The following two constraints are imposed on the assignment of orientation modes on the privileges.

- A privilege assigned to a role in the grant or in the unassign state is always in the up orientation mode thereby
maintaining the traditional privilege inheritance semantics in a role hierarchy.

- A privilege assigned to a role in the deny, taint, or suspend state may only be in the down or in the neutral orientation mode. Assigning such privilege states to a role in the down or neutral mode ensures that the role still remains more privileged than its children roles. In addition, the neutral mode is particularly useful when a privilege needs to be assigned to a role without affecting the rest of the role hierarchy (when responding to an anomaly, for example).

The privilege model of PSAC in the presence of a role hierarchy is formalized as follows:

1. \( RRA \subset R \times R \), a many-to-many role to role assignment relation. A tuple of the form \((r_1, r_2) \in R \times R\) means that the role \(r_2\) is assigned to the role \(r_1\). Thus, role \(r_1\) is a parent of role \(r_2\) in the conceptual role hierarchy.

2. \( OR = \{up, down, neutral\} \), the set of privilege orientation modes.

3. \( PRRPOSORA \subset PR \times R \times P \times O \times S \times OR \), a principal to role to privilege to object to state to orientation mode assignment relation. This relation captures the state of the access control system in terms of the privileges, their states, and their orientation modes that are directly assigned to roles by principals.

4. \( \text{assigned}\_\text{roles} (r') : R \rightarrow 2^R \), a function mapping a role \(r'\) to its assigned roles such that \( \text{assigned}\_\text{roles} (r') = \{ r \in R | (r', r) \in RRA \} \cup \text{assigned}\_\text{roles} (r) \). This function returns the set of the roles that are directly and indirectly (through the role hierarchy) assigned to a role; in other words, the set of descendant roles of a role in the hierarchy.

5. \( \text{assigned}\_\text{roles} (u) : U \rightarrow 2^R \), a function mapping a user \(u\) to its assigned roles such that \( \text{assigned}\_\text{roles} (u) = \{ r \in R | (u, r) \in URA \} \cup \text{assigned}\_\text{roles} (r) \). This function returns the set of roles that are directly and indirectly (through the role hierarchy) assigned to a user.
6.2 Design and Implementation

(6) assigned_to_roles \((r')\): \(R \rightarrow 2^R\), a function mapping a role \(r'\) to a set of roles such that assigned_to_roles \((r')\) = \(\{r \in R \mid (r, r') \in RRA\}\). This function returns the set of roles that a role is directly and indirectly (through the role hierarchy) assigned to; in other words, the set of ancestor roles of a role in the role hierarchy.

We redefine the \(priv\_states\) \((pr, r', p, o)\) function in the presence of a role hierarchy taking into account the privilege orientation constraints as follows:

(7) \(priv\_states\) \((pr, r', p, o)\): \(PR \times R \times P \times O \rightarrow 2^S\), a function mapping a principal \(pr\), a role \(r'\), a privilege \(s\), and an object \(o\) to a set of privilege states such that \(priv\_states\) \((pr, r', p, o)\) = \(\{s \in S\mid \forall or \in OR, (pr, r', p, o, s, or) \in PRRPOSORA\}\) \(\cup\) \(\{s \in \{grant, unassign\} \mid \forall r \in assigned\_to\_roles\ (r'), (pr, r, p, o, s, 'up') \in PRRPOSORA\}\) \(\cup\) \(\{s \in \{deny, suspend, taint\} \mid \forall r \in assigned\_to\_roles\ (r'), (pr, r, p, o, s, 'down') \in PRRPOSORA\}\). The set of privilege states returned by this function is the union of the privilege states directly assigned to the role \(r'\) by the principal \(pr\), the privilege states in the \(grant\) or the \(unassign\) states (also assigned by \(pr\)) obtained through the descendant roles of \(r'\), and the privilege states in the \(deny\), \(suspend\), and \(taint\) states (also assigned by \(pr\)) obtained through the roles that are the ancestor roles of \(r'\), and that are in the \(down\) orientation mode.

We now present a comprehensive example of the above introduced relations and functions in PSAC. Consider a sample role hierarchy in Figure 6.3. Table 6.2 shows the state of a sample \(PRRPOSORA\) relation.

Let the role \(r_2\) be assigned to the user \(u_1\). To determine the final state of the \(select\) privilege on the table \(t_1\) for the user \(u_1\), we evaluate \(priv\_states\) \((u_1, select, t_1)\) as follows:

\[
\begin{align*}
priv\_states(u_1, select, t_1) &= priv\_states(SU_1, u_1, select, t_1) \cup \nonumber \\
& \quad priv\_states(SU_2, u_1, select, t_1)
\end{align*}
\]
The final state is determined using the PSD_state() function as follows:

\[
\text{PSD}_\text{state}(\text{taint}, \text{grant}, \text{suspend}) = \text{suspend}
\]

6.2.5 Implementation Details

In this section, we present the details on how to extend the PostgreSQL 8.3 open-source object-relational DBMS [84] with PSAC. In the rest of
the section, we use the term PSAC:PostgreSQL to indicate PostgreSQL extended with PSAC, and BASE:PostgreSQL to indicate the official PostgreSQL 8.3 release.

6.2.5.1 PSAC:PostgreSQL

Access control in BASE:PostgreSQL is enforced using access control lists (ACLs). Every DBMS object has an ACL associated with it. An ACL in BASE:PostgreSQL is a one-dimensional array; the elements of such an array have values of the ACLItem data type. An ACLItem is the basic unit for managing privileges of an object. An ACLItem is implemented as a structure with the following fields: granter, the user/role granting the privileges; grantee, the user/role to which the privileges are granted; and privs, a 32 bit integer (on 32 bit machines) managed as a bit-vector to indicate the privileges granted to the grantee. A new ACLItem is created for every unique pair of granter and grantee. There are 11 pre-defined privileges in BASE:PostgreSQL with a bit-mask associated with each privilege [73]. As shown in Figure 6.4, the lower 16 bits of the privs field are used to represent the granted privileges, while the upper 16 are used to indicate the grant option.\(^5\) If the \(k\)th bit is set to 1 \((0 \leq k < 15)\), privilege \(p_k\) is granted to the user/role. If the \((k + 16)\)th bit is also set to 1, then the user/role has the grant option on privilege \(p_k\).

6.2.5.2 Design Details

In BASE:PostgreSQL, the \texttt{pg\_class} system catalog is used to store the metadata information for database objects such as tables, views, indexes and sequences. This catalog also stores the ACL for an object.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
31 & 30 & \multicolumn{4}{|c|}{17 & 16 & 15 & 14} & 1 & 0 \\
\hline
\end{tabular}
\caption{ACLItem privs field.}
\end{figure}

\(^5\)Recall that the grant option is used to indicate that the granted privilege may be granted by the grantee to other users/roles.
in the acl column that is an array of ACLItems. The pg_class system catalog has been extended to maintain privilege states by adding four new columns, namely: the acltaint column to maintain the tainted privileges; the aclsuspend column to maintain the suspended privileges; the acldeny column to maintain the denied privileges; and the aclneut column to indicate if the privilege is in the neutral orientation mode. Those state columns and the aclneut column are of the same datatype as the acl column, that is, an array of ACLItems. The lower 16 bits of the privs field in those state and aclneut columns are used to indicate the privilege states and the orientation mode respectively. This strategy allows one to use the existing privilege bit-masks for retrieving the privilege state and orientation mode from these columns. The upper 16 bits are kept unused. Table 6.3 is the truth table capturing the semantics of the privs field bit-vector in PSAC:PostgreSQL.

### 6.2.5.3 Authorization Commands

The BASE:PostgreSQL GRANT and REVOKE authorization commands have been modified to implement the privilege state transitions. Three new authorization commands have been introduced, that is, the DENY, the SUSPEND, and the TAINT commands. The DENY command moves a privilege to the deny state, the SUSPEND command moves a privilege to the suspend state, and the TAINT command moves a privilege to the taint state. The default privilege orientation mode

<table>
<thead>
<tr>
<th>acl kth bit</th>
<th>acl taint kth bit</th>
<th>acl suspend kth bit</th>
<th>acl deny kth bit</th>
<th>acl neut kth bit</th>
<th>pk State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>unassign/up</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>grant/up</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>taint/down</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>suspend/down</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>deny/down</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>taint/neutral</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>suspend/neutral</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>deny/neutral</td>
</tr>
</tbody>
</table>

All other combinations are not allowed by the system.
for these commands is the down mode with the option to override that by specifying the neutral orientation mode. The administrative model for these commands is similar to that of the SQL-99 GRANT command, that is, a DENY/SUSPEND/TAINT command can be executed on privilege $p$ for object $o$ by a user $u$ iff $u$ has the grant option set on $p$ for $o$ or $u$ is the owner of $o$. The syntax for the commands is reported in Table 6.4. Note that in the current version of PSAC:PostgreSQL, the new commands are applicable on the database objects whose metadata are stored in the $pg\_class$ system catalog.

### 6.3 Summary

In this section, we have presented the design, the formal model, and the implementation of a privilege state based access control (PSAC) system tailored for a DBMS. The fundamental idea in PSAC is that a privilege, assigned to a principal on an object, has a state attached to it. There are five states in which a privilege can exist namely, unassign, grant, taint, suspend and deny. A privilege state transition to either the taint or the suspend state acts as a fine-grained response to a database anomaly. An interesting open direction is to extend the notion of state to access control models, other than RBAC, and also to investigate the use of state for supporting obligations. Obligations are actions to be executed upon the access execution by the user executing the access or other parties.
The issue of insider threats is inherently a very hard problem to address. Employees in an organization are trusted by the organization to carry out its business in a professional and legal manner; and thus they possess the necessary authorizations to access much of the company’s proprietary and sensitive data. Such authorization can easily be misused by malicious or disgruntled employees for a variety of reasons. The threat from such insiders is even greater if they possess superuser privileges to an information system such as the database administrators in the case of a DBMS. In this section, we discuss two different approaches to addressing this problem in the context of a DBMS. The first mechanism that we discuss is the Oracle Data Vault (Section 7.1) that mandates creating policies that curtail the privileges of the DBAs by restricting their access to the application data. The second mechanism is the joint-threshold administration model (JTAM) (Section 7.2) that applies the security principle of separation-of-duty to execution of sensitive commands in order to mitigate the risk from malicious DBAs.¹

¹The discussion in Section 7.2 is primarily based on the work published in [47].
7.1 Oracle Database Vault

*Oracle Database Vault* (ODV) is a security mechanism introduced in Oracle Database 11g [32] to mitigate the risk of insider threats by using policies that prevent access to specific areas in an Oracle database from any user, including the highly privileged database administrators (DBAs). For example, using ODV’s security controls, administrative access may be restricted to performing only administration tasks such as backup, recovery, schema management, and so forth; and no access may be granted to the sensitive application data such as employee salaries, patient medical records, and so forth. In what follows, we first discuss the key components of the ODV mechanism. We then discuss the ODV administration model.

7.1.1 ODV Components

ODV’s security controls are based on the concept of *realms, command rules, factors, and rule sets.*

**Realms.** A realm is a functional grouping of database objects (schemas, roles, etc.) that need to be secured. After the objects have been grouped into a realm, the realm may be used to control the use of system privileges to specific users or roles. This enables fine-grained access control for anyone who wants to use the database objects that are grouped into a realm.

**Command Rules.** A command rule is a rule created to control how an SQL command (DDL/DML) may be executed. Command rules work in conjunction with other rules in a rule set to determine whether a statement executed by a user is allowed or not.

**Factors.** A factor is an attribute, such as client IP address, database IP address, database name, session user, and so forth, that ODV can recognize. Factors are used in rule sets for creating filtering logic to restrict access based on the factor identities.

**Rule Sets.** A rule set is a collection of rules that can be associated with a realm, a command rule, or a factor assignment. A rule within a rule set is a PL/SQL expression that evaluates to true or false. The
A rule set may evaluate to true or false depending upon the evaluation of each rule in the rule set. The rule sets may be used for the following purposes:

- To define the conditions under which a realm authorization is active.
- To define when a command rule is allowed.
- To define when an identity may be assigned to a factor.

We next show a simple example of how these concepts work together to protect against unauthorized access to the sensitive application data. To be specific, we demonstrate how a table (salary table for example) in a schema (finance schema for example) may be protected from access by a DBA. The first step in the process is to create a new realm (called the demo realm) and include the salary table and the finance schema into the new realm. The next step is to authorize the required database users or roles to access the salary table in the demo realm. After this step is done, only the authorized database users or roles may access the salary table data in the demo realm. Such access may further be restricted by associating a rule set with the demo realm.

### 7.1.2 ODV Administration

The key idea of the ODV’s administration model is to change some of the privilege semantics in Oracle to limit the privileges of the DBAs and require a separate set of users (other than the DBAs) to manage the ODV objects. A special protected schema called the DV SYS schema holds the ODV objects. The ODV objects store the ODV configuration information and also support the administration of run-time processing of ODV. The DV SYS protected schema guards the schema against improper use of system privileges such as SELECT ANY TABLE, DROP ANY, and so forth. Only the DV SYS user and other database vault roles can have the privileges to modify objects in the DV SYS schema. The powerful ANY system privileges for database definition language and data manipulation language commands are also restricted in the DV SYS protected schema.
7.2 Joint-Threshold Administration

The Joint Threshold Administration Model (JTAM) may be used for performing certain critical and sensitive database operations (at production sites) such as user or role creation, deletion, or modification, grant or revoke of permissions, management of ODV objects, and so forth. For the sake of brevity, let us call such operations that use the joint administration model as \textit{JTAM-ops}. The organizational setting in which JTAM may be applied is as follows. Suppose there are a few database administrators (DBAs) in an organization managing tens of databases. It is not possible for the organization to divide the duties of its DBAs specifically since there are a large number of tasks to be performed and only a few DBAs to carry out the tasks. Thus, every DBA is granted the same level of privileges in the DBMS such that it has the capability to perform any action as necessary to ensure the smooth operation of the database. The risk of privilege abuse from malicious insiders is substantial in such a scenario since not all the DBAs enjoy the same level of trust. The goal of JTAM is to mitigate such risk from malicious DBAs and also prevent accidental mistakes when performing the \textit{JTAM-ops} in such an organizational setting.

The JTAM approach is based on the security principle of separation of duty (SoD). As a security principle, the primary objective of SoD is prevention of fraud and user generated errors. This objective is traditionally achieved by dividing the task among multiple users. \textit{The fundamental premise of this approach is that a single DBA is not trusted to perform a JTAM-op, and the threat is substantially mitigated if the trust is distributed among multiple DBAs.} The key idea is that a \textit{JTAM-op} is not executed by the DBMS unless it has been authorized by at least \( k - 1 \) additional DBAs where \( k \) must be specified when creating the \textit{JTAM-op}.

A key requirement for the JTAM approach to be successful and usable is that the authorization process must be asynchronous such that all the \( k - 1 \) DBAs need not be logged into the database simultaneously to authorize a \textit{JTAM-op}. To achieve this requirement, the details of an in-progress \textit{JTAM-op} must be stored in a database table, so they can be used and updated as the operation is authorized by the other DBAs.
When the threshold number of DBAs have authorized the operation, it is then executed by the DBMS. However, note that the DBAs have unlimited privileges to the DBMS; hence they also have the ability to execute arbitrary malicious SQL update commands on the table storing the in-progress JTAM-op details. Such actions are possible even if the details are stored in a system catalog since many popular DBMSs allow DBAs to make ad-hoc updates to the system catalogs [71, 84].

To address this issue, the approach that is followed in JTAM is to create a digital signature on the hash of the in-progress JTAM-op details. The JTAM-op is executed by the DBMS only when a valid digital signature can be created on the JTAM-op details. One of the key assumptions in JTAM is that the DBMS is not assumed to be in possession of a secret key for generating such signatures. If the DBMS were to possess such key, it could create a HMAC (Hashed Message Authentication Code) for each in-progress JTAM-op using the secret key, and later use the same key to verify the integrity of the JTAM-op. However, management of such secret key is again an issue since the key may not be assumed to be hidden from a malicious DBA. Therefore, the JTAM approach instead applies the techniques of threshold cryptography signatures to create a digital signature on a JTAM-op.

**Threshold Signatures.** A $k$ out of $l$ threshold signature scheme is a protocol that allows any subset of $k$ users out of $l$ users to generate a valid signature, but that disallows the creation of a valid signature if fewer than $k$ users participate in the protocol [82]. The basic paradigm of most well-known threshold signature schemes is as follows [39]. Each user $U_i$ has a secret key share $s_i$ corresponding to the signature key $d$. Each user participating in the signature generation protocol generates a signature share that takes as input the message $m$ (or the hash of the message) that needs to be signed, his/her secret key share, and other public information. Signature shares from different users are then combined to form the final valid signature on $m$.

Thus, in JTAM, a DBA authorizes an in-progress JTAM-op by submitting a signature share on the hash of the in-progress JTAM-op details. At least $k$ signature shares are required to form a valid
final signature. Once a valid final signature is formed on an in-progress JTAM-op, the operation is executed by the DBMS.

### 7.2.1 Design Details

JTAM uses the Practical Threshold Signature scheme by Victor Shoup [82] to implement the threshold signatures. The main reasons for using this scheme are as follows. First, it allows for the signature share generation procedure to be asynchronous, and the signature share combining operation to be completely non-interactive satisfying the two main requirements of JTAM. Second, the signature shares generated by this scheme can be made public without compromising their security. This allows one to store them in a public-readable database table without any security concerns. Third, the signature verification mechanism in this scheme is very efficient (since it is based on RSA digital signatures), thereby minimizing the overhead on the DBMS operations due to signature verification. In what follows, we present the details of Shoup’s protocol in the context performing a JTAM-op.

#### 7.2.1.1 Registration Phase

The parameter \( l \) is set greater than or equal to the number of DBAs registered with the DBMS. Such requirement allows any DBA to generate a valid signature share, thereby making the approach very flexible. Next, for every \( k \) such that \( 2 \leq k \leq l - 1 \), the DBMS generates the RSA public key, and the secret key shares, \( s_i \), for each of the \( l \) DBAs. Note that these secret shares are generated for every \( k \) value, and not for every JTAM-op. Also note that \( k \geq 2 \) means that at least 1 additional DBA needs to authorize a JTAM-op before it can be executed.

In JTAM, some security parameters need to be registered with the DBMS as part of a one-time registration phase. The details of the registration phase are as follows: the parameter \( l \) is set equal to the number of DBAs registered with the DBMS.\(^2\) Such requirement allows any DBA

---

\(^2\)The registration of the DBAs (including assigning initial passwords) will be typically handled by a DBA itself. The security parameters needed for JTAM operations are presented as DBMS configuration options that may also be set by any DBA. The scenario that is assumed here is that there are multiple administrators, each holding the DBA role,
to generate a valid signature share on a policy object, thereby making the approach very flexible. Shoup’s scheme also requires a trusted dealer to generate the security parameters. This is because it relies on a special property of the RSA modulus, namely, that it must be the product of two safe primes. We assume the DBMS to be the trusted component that generates the security parameters.\footnote{In practice, only a small portion of the DBMS code base that deals with JTAM operations needs to be trusted.} For all values of \( k \), such that \( 2 \leq k \leq l - 1 \),\footnote{Note that \( k \geq 2 \) means that at least 1 additional DBA needs to authorize a JTAM-op before it can be executed.} the DBMS generates the following parameters:

- **RSA Public-Private Keys.** The DBMS chooses \( p, q \) as two large prime numbers such that
  \[
  p = 2p' + 1 \quad \text{and} \quad q = 2q' + 1
  \]
  where \( p' \) and \( q' \) are themselves large primes. Let \( n = p \times q \) be the RSA modulus. Let \( m = p' \times q' \). The DBMS also chooses \( e \) as the RSA public exponent such that \( e > l \). Thus, the RSA public key is \( PK = (n, e) \). The server also computes the private key \( d \in \mathbb{Z} \) such that \( de \equiv 1 \mod m \).

- **Secret Key Shares.** The next step is to create the secret key shares for each of the \( l \) DBAs. For this purpose, the DBMS sets \( a_0 = d \) and randomly assigns \( a_i \) from \( \{0, \ldots, m - 1\} \) for \( 1 \leq i \leq k - 1 \). The numbers \( \{a_0, \ldots, a_{k-1}\} \) define the unique polynomial \( p(x) \) of degree \( k - 1 \), \( p(x) = \sum_{i=0}^{k-1} a_i x^i \). For \( 1 \leq i \leq l \), the server computes the secret share, \( s_i \), of each DBA, \( DBA_i \), as follows:
  \[
  s_i = p(i) \mod m.
  \]

The secret shares can be stored in a *smart card* or a *token* for every DBA, and submitted to the DBMS when required to sign a JTAM-op. The other alternative, that is implemented in JTAM, is to let the DBMS store the shares in the database encrypted with keys generated out of

and thus having the same level of privileges. We assume that the DBAs are individually trusted to perform the administration tasks such as registration of the DBAs, database configuration, etc.
7.2 Joint-Threshold Administration

the DBA’s passwords. Thus, during the registration phase, each DBA must submit its password to the DBMS for encrypting its secret key shares. Using such strategy, whenever a DBA needs to sign a JTAM-op for authorization, it submits its password which is used by the DBMS to decrypt the DBA’s secret share, and use that to generate the correct signature share. Frequent changes of passwords do not represent a security issue since the password change procedure of the DBMS can be enhanced to re-encrypt existing secret shares of the DBA with the new password.

7.2.1.2 Life cycle of a JTAM enabled command

We present the details of a JTAM-op in the context of the Create User SQL command.

Command Creation. The Create User JTAM-op has the following format: Create User […] Cmd Definition […] Joint Adm By k Users. Suppose that DBA$_1$ issues such command and that $k = 3$. The DBMS, upon receiving such command, prompts DBA$_1$ to input its password, uses the password to decrypt the secret share, $s_1$, of DBA$_1$ corresponding to $k = 3$, and generates a signature share on the hash, $H(Cmd)$, of the command. The hash is taken on all the command attributes that need to be protected from malicious modifications. For the Create User command, such attributes are user name, user permissions, and so forth. Thus, $H(Cmd) = \text{Hash}(… \text{Cmd Definition} …, k)$. Let $x = H(Cmd)$. The signature share of DBA$_1$, is $W(DBA_1) = x^{2\Delta s_1} \in Q_n$, where $\Delta = l!$, and $Q_n$ is the subgroup of squares in $Z_n^*$. $W(DBA_1)$ does not leak any information about the secret share $s_1$ because of the intractability of the generalized discrete logarithm problem [60]. The DBMS also generates a unique command ID, CmdID, for the in-progress command. The command definition and the CmdID along with the signature share, $W(DBA_1)$, and $H(Cmd)$ are stored in the $pgm\_jtam\_cmds$ system catalog. The column $r$ stores the number of DBAs that have yet to authorize the command. The initial value of $r$ after completion of the command creation step is $k - 1 = 2$. Table 7.1 shows the state of the $pg\_jtam\_cmds$ catalog after a JTAM enabled command has been created.
Command Authorization. The JTAM enabled Create User authorization command has the following format: Authorize Create User Id={CmdID}. The DBMS, upon receiving such command, prompts the DBA to input its password, uses the password to decrypt the secret share of the DBA corresponding to the \(k\) value associated with the \{Cmd Id\}, and generates a signature share on the hash, \(H(Cmd)\), of the command. The attributes for recreating \(H(Cmd)\) are read from the \texttt{pgm\_jtam\_cmds} catalog. When \(k − 1\) administrators have authorized the command, the signature combining and verification algorithms are executed. The final signature on the command, \(W_{\text{final}}\), is verified using the RSA public key corresponding to the \(k\) value associated with the command. The detailed signature combining and verification procedure is as follows:

Let \(S\) be the set of DBAs that have submitted the signatures shares on a policy; \(S = \{i_1, \ldots, i_k\} \subset \{1, \ldots, l\}\). Let \(x = H(Cmd) \in \mathbb{Z}_n^\times\), and \(x_{ij}^2 = W(U_{ij})^2 = x^{4\Delta s_{ij}}\). To combine the signature shares, we compute \(w\) such that

\[
w = x_{i_1}^{2\lambda_{0,i_1}^S}, \ldots, x_{i_k}^{2\lambda_{0,i_k}^S} = x^{4\Delta(\sum_{j \in S} \lambda_{0,j}^S s_{ij})}
\]

where the \(\lambda\)'s are the integers defined as follows:

\[
\lambda_{i,j}^S = \Delta \prod_{j \in S \setminus \{j\}} \frac{(i - j)}{j \in S \setminus \{j\} \prod (j - j')} \in \mathbb{Z}, \quad i \in \{0, \ldots, l\} \setminus S, \quad j \in S.
\]

These values of \(\lambda\) are derived from the standard Lagrange polynomial interpolation formula [52]. Using the Lagrange interpolation formula, we have

\[
\Delta \cdot f(i) \equiv \sum_{j \in S} \lambda_{i,j}^S f(j) \mod m
\]
Thus,

\[ w^e = x^{4\Delta(\sum_{j \in S} \lambda^S_{0,j} \cdot t_j)} = x^{4\Delta(\sum_{j \in S} \lambda^S_{0,j} f(j) \mod m)} = x^{4\Delta f(0) \mod m} = x^{4\Delta^2 (de \mod m)} = x^{e'} \]

where \( e' = 4\Delta^2 \) since \( de \mod m \equiv 1 \) (RSA property). Since Shoup’s scheme is based on RSA threshold signatures, the final signature is the classical RSA signature [60]. To compute the final signature \( W_{\text{final}} = y \) such that \( y^e = x \), we set \( y = w^a x^b \) where \( a \) and \( b \) are integers such that \( e'a + eb = 1 \). This is possible since \( \gcd(e, e') = 1 \). The values of \( a \) and \( b \) are obtained from the standard Euclidean algorithm on \( e \) and \( e' \) [60].

The final signature, \( W_{\text{final}} \), is verified using the public key \((n, e)\) corresponding to the value of \( k \). If \((W_{\text{final}})^e = H(Cmd)\), the signature is considered to be valid.

If the final signature is found to be valid, the DBMS performs the following steps as a transaction. First, it executes the traditional \textit{Create User} command to create the database user. Second, it copies the final signature to the database users system catalog (where database user details are stored) along with the \( k \) value associated with the command, and attaches it to the newly created user entry. Third, it deletes the command entry from the \textit{pgm_jtam_cmds} catalog.

Note that an in-progress command may be modified by a malicious DBA using the SQL update statement on the \textit{pgm_jtam_cmds} catalog before it has been authorized. However, all command definition attributes that need to be protected from malicious modifications are included in the command hash, \( H(Cmd) \); therefore any modification to such attributes through the SQL update command will force an invalid final signature to be generated on the command thus preventing the command execution.

\textbf{Signature Verification Process.} The reason for copying the final signature to the database users catalog is to prevent a malicious DBA from modifying the user details directly by using an SQL update command on the database users catalog. JTAM creates a \textit{signature verification}
database server process that periodically polls the database users catalog to validate the signature on its tuples.\footnote{The hash for validating the signature is generated out of the command details present in the user tuple and the $k$ value that was copied over during the command execution.} If a malicious DBA tries to modify any of the tuples directly using SQL update, the signature is invalidated and an alert is generated by the signature verification process.

### 7.3 Summary

There are considerable risks to an organization’s sensitive and proprietary data from malicious insiders. In this section, we have described two very different approaches to addressing the insider threat problem in the context of a DBMS. The first mechanism that we discussed is Oracle Data Vault that mandates creating policies that restrict a database administrator’s access to the application data. The second mechanism is the joint-threshold administration model (JTAM) that applies the security principle of separation-of-duty to sensitive command execution to mitigate the risk from malicious DBAs.
Though the relational database technology has today a central role in the data management arena, in the past 20 years, numerous extensions to this technology have been proposed and incorporated in commercial products. These extensions have been driven on one hand by the need of managing complex, multimedia objects, and on the other hand by the widespread use of Internet and Web-based applications, that have fueled the development of interoperability approaches, like XML and Web services, and also made possible to share data across different administration domains. A major requirement underlying all those extended data management systems and tools is a demand for adequate security and, in particular, tailored access control models and systems. In what follows, we first discuss relevant requirements. We then discuss in detail the authorization model of the Orion object DBMS [51], as this is the most significant access control model for object databases. We then briefly discuss issues concerning the application of MAC models to object databases, and we conclude by covering access control models and systems for XML data, for geographical data, and for digital libraries. The material in this section is partially based on the material in the paper by Bertino and Sandhu [19].
8.1 Requirements

Relevant features that an access control system for advanced database applications should provide include:

**Fine-grained flexible authorization models for complex, multimedia objects.** Most advanced (post-relational) applications are characterized by objects whose structure is far more complex than the simple flat structure typical of relational data. This is the case, for example, of XML data and multimedia data. Because applications may directly access data at various granularity levels from sets of data objects to specific portions of a single data object, mechanisms are needed to control access at varying granularity levels and to be able, at the same time, to support concise specification of authorizations. Extensions that have been proposed to address such requirements include the notions of positive/negative authorizations, and implicit/explicit authorizations [74] that we discuss in the context of access control models for object-based systems.

**Flexible user specification mechanisms based on user credentials and profiles.** Most Web-based applications are characterized by a user population which is far more heterogeneous and dynamic than the user population typical of conventional information systems. In such a scenario, traditional identity mechanisms, based on login or user names, for qualifying the subjects to which a policy applies are no longer appropriate in that they would require the specification and management of a large number of policies. There is thus the need of using other properties of subjects (e.g., age, nationality, job position) besides their login names, in the specification and enforcement of access control policies. Such properties, that can be considered a form of partial digital identity, are often encoded into user profiles and certified by means of credentials and attribute certificates. The RBAC model provides some initial approach, in that we can consider roles as a property of subjects; however we need generalized mechanisms efficiently supporting the use of multiple digital identity properties in access control policies (expressed in languages like XACML).
Access control mechanisms tailored to data dissemination strategies and third party publishing architectures. An important requirement of today’s Web-based applications is to support a variety of data dissemination strategies. A dissemination strategy regulates how a data source delivers data to subjects. In conventional database systems, data are delivered according to a strategy known as pull strategy. According to such a strategy, data are delivered to subjects upon an explicit request by the subjects. However, in a Web environment, an alternative strategy can be adopted, which is more suitable when data have to be delivered to a large community of subjects. According to such a strategy, referred to as push strategy or as publish-subscribe, the data source periodically (or when some predefined events happen) sends data to authorized subjects, without the need of an explicit access request by the subjects. In some cases, the data that are sent to subjects also depend on the specific subject interests, that are recorded in some special subject profiles managed by the data source. Supporting different dissemination strategies may require the adoption of different access control techniques depending on the data dissemination strategy adopted. A comprehensive access control system should thus provide a large variety of access control techniques able to enforce a given policy under a variety of dissemination strategies.

8.2 The Orion Authorization Model

The first and most comprehensive discretionary access control model has been defined for the Orion object-oriented DBMS. The access control model of Orion has introduced a number of novel concepts.

Varying object protection granularity. The Orion access control model established the idea that when dealing with complex objects, such as composite objects and versions, access can be allowed to entire objects or only to portions of objects. As such the access control model makes available a large variety of protection object types, based on the conceptual structures of the object data model, including classes, composite objects, versions, and class hierarchies.
Explicit and implicit authorizations. An access right for a subject on an object can be explicitly specified through a grant command, referred to as *explicit authorization*, or can be implicitly derived from other explicit authorizations, referred to as *implicit authorization*. Authorization derivation can occur along all the three domains of authorizations, that is, objects, subjects, and modes. In particular, implication rules on objects support the derivation of authorizations from an object to all objects semantically related to it; the semantic relationships are based on the conceptual structures of the model. For example, a read authorization on the root of a version hierarchy implies read authorizations on all the versions in the hierarchy. However, it is also possible for an authorization to be granted on a single version of an object. The use of implication rules is instrumental in providing varying granularity levels of protection without performance penalties.

Positive and negative authorizations. The Orion access control model has been one of the first models to introduce the notion of negative authorization. The main purpose of this type of authorization is the support for exceptions in implicit authorizations. The combined use of implicit and negative authorizations allows one to concisely express a large number of access control policies. For example, consider a class with 2000 instances; suppose that a subject has to be authorized to access all those instances except one. Under a conventional access control model one would have to issue 1999 authorizations. Under the Orion model, one would need to enter only two authorizations, that is, a positive authorization on the class, which would automatically propagate to all instances, and a negative authorization on the instance to be excluded. Also exceptions may occur at different levels within the protection object hierarchical organizations, and therefore it is possible for exceptions to have exceptions.

Weak and strong authorizations. Authorizations in most access control models are *strong* in the sense that the authorizations implied by some authorization cannot be overridden. A weak authorization allows one to make exceptions in the implicit authorizations. A strong authorization guarantees that all of the authorizations it implies cannot be
overridden, whereas authorizations implied by weak authorizations can be overridden by other (strong or weak) authorizations.

**Protection of schema objects.** The access control model of Orion also established that the definition of objects, such as class schemas, can represent sensitive information and thus has to be protected.

Figure 8.1 illustrates the interplay of weak and strong authorizations and implicit authorizations. In this example, there is an explicit negative authorization on database[Customer]; this authorization is weak and is thus overridden by a strong authorization on class-hierarchy[ForeignCustomer]. This strong authorization propagates to all classes in the class hierarchy. Notice that the implicit authorization on class[DomesticCustomer] is not overridden by the strong authorization, since this class does not belong to class-hierarchy [ForeignCustomer].

A formal model for the Orion access control model has been defined and we refer the reader to [74] for details. Some key definitions from this model are as follows. Let $S$, $O$, and $A$ be the set of subjects, protected objects, and access modes.

- **Positive authorization:** a triplet $(s, o, a)$ with $s \in S$, $o \in O$, and $a \in A$. 
• **Negative authorization**: a triplet \((s, o, \neg a)\) with \(s \in S\), \(o \in O\), and \(a \in A\).

• **Authorization**: a positive or negative authorization.

• **Strong Authorization Base (SAB)**: a set of explicit authorizations.

• **Weak Authorization Base (WAB)**: a set of explicit authorizations.

• **Scope of the SAB**: the set of authorizations that are implied, that is, derived, from the set of authorizations in the SAB.

• **Scope of the WAB**: the set of authorizations that are implied, that is, derived from the set of authorization in the WAB and that are not overridden by exceptions.

• **Access Control Function**: it determines if an authorization \((s, o, a)\) is true or false: \(f : S \times O \times A \rightarrow \{\text{True, False}\}\).

In addition to these basic definitions, the Orion access control model includes a number of invariants. The first two invariants are defined for the SAB and require the SAB to be consistent, and non-redundant. Consistency requires that the SAB and its scope do not contain conflicting authorizations, that is, both a negative and positive authorization for the same subject, object, and access mode. Non-redundancy requires that if an authorization can be derived from another authorization in the SAB, then the former should not be in the SAB. Notice, however, that this invariant can be relaxed for performance reasons.

The other two invariants are defined for the WAB and require that the WAB be consistent and complete. The consistency invariant has the same meaning as the consistency invariant for the SAB. The completeness invariant requires that the entire space \(S \times O \times A\) be covered by some weak authorization; this ensures that the access control function is able to always return a decision. The last invariant concerns both SAB and WAB and states that the authorizations in the WAB and its scope are to be used only when it is not possible to return an access control decision based on the SAB and its scope. Notice that this invariant basically states that strong authorizations have always the precedence with respect to weak authorizations.
The formal definition of function $f$ is based on the definition of two auxiliary functions:

- Function $i(s,o,a)$: this function, given an authorization, determines whether this authorization exists in the $SAB$ or in its scope; it is defined as $i: S \times O \times A \rightarrow \{\text{True, False, Undecided}\}$.
  
  The definition of this function clearly states that there could be authorizations that are not in the $SAB$ nor can be derived from the $SAB$ and thus the $SAB$ may not always be able to return a decision concerning an access request.

- Function $d(s,o,a)$: this function, given an authorization, determines whether this authorization exists in the $WAB$ or in its scope; it is defined as $d: S \times O \times A \rightarrow \{\text{True, False}\}$.
  
  Note, that unlike function $i$, function $d$ states that a decision must always be returned. In order to achieve this, a special default authorization is used.

Based on these two functions, the semantics of the $f$ function is defined as follows.

\[
  f(s,o,a) = \begin{cases} 
  i(s,o,a) & \text{if } i(s,o,a) = \text{Undecided} \\
  d(s,o,a) & \text{else}
  \end{cases}
\]

Therefore, in order to check the authorization $(s,o,a)$ we first determine from the $SAB$ if the authorization is True or False. If this cannot be determined from the $SAB$, we use the $WAB$ to determine if the corresponding weak authorization is True or False.

An interesting issue in this model is which are the set of access modes. With a large number of data object types, one may have to introduce a large number of corresponding access modes. The design choice made for the Orion model was to minimize the set of access modes, by using a small number of access modes and interpreting them depending on the objects to which they were applied. The set of access modes defined for the Orion model is thus: \{Read (R), Write (W), Generate (G), Read Definition (RD)\}. In particular, the $G$ access mode allows one to generate an object from another object; for example, when applied to a set of instances associated with a class $c$, it allows a user to create an instance from $c$. The $RD$ access allows one to read the
Access Control for Object Databases, XML Data and Novel Applications

definition of an object. It has many different meanings; for example when it is applied to a class object allows one to read the definition of the class, whereas when it is applied to a versioned object allows one to access the version tree of the object.

All access modes, except for the \( G \) access mode, are defined for all possible object types. The set of access modes is partially ordered. Figure 8.2 shows the partial order by using a graph representation. Each node in the graph denotes an access mode; there is an edge from the node representing access mode \( a \) to the node representing access mode \( a' \) if \( a \) precedes \( a' \) in the partial order. Authorizations also propagate according to the partial order defined on the access mode set. For example, an authorization to write an object implies the authorization to read the object and its definition; as such, the latter two authorizations do not need to be issued if the former is issued.

Another interesting notion part of the Orion access control model is the notion of authorization object schema (AOS). Such schema, modeled as a graph, represents all database protection granularity units, modeled as nodes, and structural relationships among these granularity types, modeled as edges. Figure 8.3 presents the AOS for the Orion object DBMS. Notice that the AOS includes all semantic modeling constructs of the object model supported by Orion, such as composite objects and versions, as well as other protected objects, like indexes. The AOS determines how authorizations on one object \( o \) propagate to authorizations on objects semantically related to \( o \). Authorization propagation according to the AOS, however, depends on the type of access mode and whether the authorization is negative and positive. The main rationale for the propagation is to reduce the number of

![Diagram of partial order for the access modes of the Orion authorization model.](image-url)
8.2 The Orion Authorization Model

Fig. 8.3 Authorization object schema for the Orion object database.

authorizations to be granted for a subject to access an object. Therefore access modes are grouped into three sets: modes that propagate down in the AOS ($R, W$), modes that propagate up in the AOS ($RD$), and modes that do not propagate ($G$). The propagation of the $RD$ access mode represents an interesting case. For example consider the authorization to read an instance of a class. The $R$ access mode implies the $RD$ access mode (see Figure 8.2). However since the definition of an instance is given by its class, the authorization to read the definition of an instance automatically implies the authorization to read the definition of its class.

The above discussion clearly shows that implications along different authorization domains are combined when performing access control. As an example, suppose that the following authorization is in the $SAB$: $(Bob, database[Customer], W)$. Suppose that the Customer database contains a class called ‘ForeignCustomer’. Suppose that Bob would like to access the i-th instance of this class, denoted as $ForeignCustomer[i]$. The following derivation steps are performed: along domain $O$, we can derive the authorization $(Bob, ForeignCustomer, W)$ and from this the
authorization (Bob, ForeignCustomer[i], W). Based on this authorization, along domain A we can derive the authorization (Bob, ForeignCustomer[i], R). Therefore, Bob is allowed the requested access.

8.3 MAC Models for Object Databases

The application of MAC models to object databases is quite challenging, due to the semantic richness of object data models. Moreover, the differences both in theory and implementation among the various object DBMSs make it very difficult to define sound and general principles upon which a suitable MAC model for object DBMSs can be based.

MAC models proposed for object databases can be classified into single-level models and multilevel models. Models in the first category require that an object and all its features, e.g., attributes and methods, be classified at the same access class. Models in the second category do not impose such a restriction; however, they are rather difficult to implement in practice. Most proposed models are thus single-level. The main reason is the simplicity of such an approach and its compatibility with a security kernelized architecture. By using an underlying security kernel for the enforcement of MAC properties, the layer implementing the object data management system need not be trusted. The main drawback of single-level models, despite their simplicity of implementation, is that applications often need objects that are multilevel. In order to accommodate such applications, the most common approach is to use a single-level object system and map the multilevel application objects onto several single-level objects. This approach, first proposed by Thuraisingham in a seminal paper [85] and referred to as multilevel object view approach, has two variants depending on whether inheritance or aggregation is used to support the multilevel view. Real multilevel object models are more difficult to handle and no satisfactory approach has been proposed.

8.4 Access Control Models for XML Data

XML [89] is widely used in a large variety of applications and industry products as it has become the standard for data exchange on the Web.
The most important feature of XML is the notion of *semantic tags*, allowing one to identify different semantic portions, called *elements*, of a given document and to assign to them names that are semantically meaningful. Elements may in turn contain other elements, called *subelements*; thus, an XML document is often characterized by a nested organization. An element may also have associated attributes, whose purpose is to provide additional information on the element. XML documents can be interconnected through some special attributes, e.g., IDREF/URI attributes. Additional key features of XML are the notions of Document Type Definition (DTD) and XMLSchema, used for specifying document structures. Note that, unlike relational data, an XML document does not necessarily have a DTD or XMLSchema of which it is an instance. An XML document which is an instance of some DTD (XMLSchema) is said to conform to the DTD (XMLSchema).

The main requirements for an access control system for XML arise from the nested structure of XML documents and from the main context for their use, that is, Web-based environments. Like object data, the nested structure of XML documents requires a flexible protection object granularity. The system must be able to support a wide spectrum of protection granularity units, identified on the basis of both the document structure and contents. Examples of protection granularity units are a single document, a set of documents, an element of a document, and an attribute of a document. Moreover, it must be possible to exploit the intended description given by a DTD or XMLSchema in the specification of protection objects. For example, it must be possible to specify access control policies at the DTD/XMLSchema level, which apply to all data conforming to that DTD/XMLSchema. To address such requirement, the same techniques proposed for access control in object DBMS have been adopted. Most of the proposed XML access control systems thus support positive/negative authorizations and explicit/implicit authorizations that can associated with a DTD, a single data, or to specific portions (elements, subelements, attributes) of a document. Authorization propagation, typical of implicit authorization mechanisms, can apply to various types of semantic relationships among protection objects (for instance, element-to-subelement and element-to-attribute/link relationships). With respect to protection objects,
however, an important difference between object databases and XML data is that in the former each object is necessarily an instance of some class and, thus, if authorizations are specified at class level, each database object is “controlled” by some authorizations. By contrast, in an XML data source, not necessarily each data is an instance of some DTD (or XMLSchema); it may happen, for example, that a source imports XML data for which no DTD (or XMLSchema) is specified. Thus, not every data in an XML source is necessarily covered by some access control policy. If the system uses a closed world access control policy, users may unnecessarily be denied access to some data items. To date, this problem has not been investigated much and the only proposed solutions are those by the Author-X system [9].

The Author-X system includes other relevant features, namely the possibility of specifying subjects in authorizations by predicates again subject properties. Any subject that satisfies the subject predicates associated with an authorization is implicitly granted the access right in the authorization. It also supports an encryption-based mechanism to enforce access control based on the use of a hierarchical key management scheme [9], distributed cooperative updates through the combination of hash functions, digital signature techniques and digital certificates [59], specification and enforcement of data flow policies, and third-party data publishing [8], through the use of the Merkle hash tree technique.

8.5 Access Control Models for Geographical Data

Geographical data are crucial in several applications, like homeland security, marketing analysis tools and environmental risks control procedures [5]. Most applications in these areas require a fine-granularity flexible access control to geographical information. Conventional access control models, however, are not suitable for geographical databases, because of the peculiarities of geographical maps. Objects in these maps can be represented with different dimensions and the accesses can also be driven by the reference space (i.e., authorization to access only data concerning geographical entities in a given region). Moreover, geographical data can be represented using different approaches. The users of a
geographical information system (GIS) usually recognize geographical data from the existence of a geometry describing the shape, extension and location of some geographical objects (features). However, geographical data can be represented also in other forms, for example by using a set of topological relations (topological representation), that specify the adjacency, the disjointness or other kind of interaction between two features. Those various representations must be taken into account when defining an access control model.

Despite its significance, the problem of access control for geographical data has not been much investigated and very few models have been developed to date. The first such model has been defined by Bertino and Damiani [13]. In this model, spatial data consist of objects having a geometry compliant with the OpenGIS simple features model [27]. Authorizations on spatial objects can be applied to limited areas (windows) within the reference space. As an example, a user may be authorized to insert road objects only if the roads are located in a given region. Windows define the geographical scope of authorizations, thus making authorizations themselves geographical objects which occupy a position in the reference space. This model has however several limitations, including the use of a simple map model and authorization propagation model. Such drawbacks have been addressed by the comprehensive model by Belussi et al. [5]. This access control model has been defined for geographical maps, admitting multiple (vector-based and topological) representations. The key concepts of the access control model by Belussi et al. are derived from the Orion authorization model, including the notion of authorization derivation and negative and positive authorizations. However, because of the nature of spatial objects, additional derivation rules have been defined by Belussi et al. in order to take into account object dimension and spatial layer. Such information is contained in the granted privileges. The most informative layer is the geometric one, since topological information can be computed from it but the converse is not true. Thus, an authorization granting a privilege for the geometric layer has to be propagated to the topological layer. On the other hand, an authorization denying a privilege for the topological layer has to be propagated to the geometric one. Similar rules are defined by considering object dimension.
Indeed, an authorization granting a privilege to objects with a certain dimension has to be propagated to objects with lower dimension (e.g., if a user can select regions, the user can also select lines and points). On the other hand, an authorization denying a privilege to objects with a certain dimension has to be propagated to objects with higher dimension (e.g., if a user cannot select points, the user cannot select neither lines nor regions). The resulting access control model is thus quite complex and, even though this model is an important reference model for GIS data, research is needed to determine which features should be incorporated in an access control model suitable for deployment in applications.

8.6 Access Control Models for Digital Libraries

Digital libraries can be defined as collections of multimedia documents. Such documents are often unstructured or semistructured and contain textual and image components. The main challenge in access control models for such data is how to support content-based access control. Content-based access control is typically based on predicates against the content of data. However whereas for relational databases expressing such predicates is easy as predicates are expressed against the attributes of relations, expressing such predicates for multimedia documents is more complex because of their lack of structure. A possible approach to addressing this problem is to express conditions against the concepts associated with the documents. The rationale is that concepts typically well represent the contents of documents. The use of document concepts for expressing content-based access control requires however addressing several issues. First, it is necessary to have some mechanism in place to extract concepts from documents. Then, the relationships that may exist among concepts in a given domain must be taken into account (for instance, a concept may subsume other concepts and so on). Because of such issues, content-based access control models for digital libraries are effective only if they rely on effective mechanisms for correctly identifying the concepts associated with documents. To date, only one such model has been developed based on a sophisticated methodology for concept extraction from documents in
a specialized application domain. We refer the reader to the paper by Ferrari et al. [38] for details about this model.

### 8.7 Summary

The discussion in this section has shown that developing access control models for complex object databases is quite difficult because of the complexity of the data models. However, initial approaches have identified relevant requirements and formulated interesting solutions. These approaches have been extended in different directions. For example, the notion of derived authorizations has been extended, based on concepts from logic programming, to support arbitrary authorization derivation rules, not necessarily based only on the structural relationships among objects [11]. An important research issue is the development of content-based access control for multimedia data based on effective content identification techniques.
Conventional database access control relies on a “pull” approach, whereby data are stored on dedicated machines under the ownership and control of the same entity that owns the data. Users send their queries to database servers that determine whether the requests are authorized or not based on some pre-defined access control policy. However, many emerging applications, such as Internet-scale data delivery through publish–subscribe systems, rely on different data sharing paradigms. In such a context, having the users directly connect to the database and ask queries may not scale well, and a “push”-based approach to data dissemination is more suitable. Instead of answering user queries, data owners choose to release the data, and users process their own queries on a local copy. Nevertheless, the access of users to data must be thoroughly controlled even in this setting.

Several approaches consider access control enforcement through encryption [14, 61]. Any user can get a copy of the encrypted data, but only authorized users have the key and are able to decrypt data contents. Since data published at the Internet scale must comply with common formats accessible to a large population of users, the focus is often on flexible formats such as XML that facilitate
9.1 Encryption-based Access Control for XML Documents

Sharing. In Section 9.1, we review two prominent solutions that address encryption-based access control for XML data.

In addition to protecting the data, in some scenarios it is equally important to protect the anonymity of users requesting data access. Consider the example of healthcare applications where results of medical research are published online. If a service provides access to the data on a subscription-only basis, only authorized users may retrieve data. Conventional systems require users to authenticate through certain identity attributes to prove that they are authorized. However, in the process the data owner may learn private details about the user based on the accessed data items. If a subscriber requests access to documents related to AIDS medication, there is a good chance that the subscriber may be infected with HIV. In Section 9.2, we review work [22, 81] which addresses the development of access control mechanisms that protect the privacy of principals.

Note that, although in this work we focus on XML databases, many commercial applications require encryption based access control in the context of relational DBMS. For instance, payment information such as credit card numbers, or personally-identifying information such as social security numbers, must always be stored in encrypted form. Standards such as Payment Card Industry Data Security Standard (PCI DSS) specify compliance requirements that must be met when storing payment information.

9.1 Encryption-based Access Control for XML Documents

XML documents consist of a hierarchy of elements, and each element can have a number of attributes. Some elements and attributes are public, whereas others are confidential, and should be accessed only by authorized users. In the “pull”-based access control model, enforcing access control is as simple as filtering out the protected elements and attributes before returning the documents to users. However, such an approach is not possible in the data publishing model. To address the “push”-based data dissemination model, the work in [14, 61] restricts access to data using encryption. Specifically, the data security policies are enforced by encrypting the protected sections of a document.
Encryption-based Access Control

Figure 9.1 illustrates the system architecture used in [61]. In this setting, the access control policies are specified through policy queries, expressed in a language that derives from XQuery. The set of policy queries and the raw (unprotected) XML data are fed into a policy query evaluator, which creates a logical XML protected tree, comprising of an XML document with guarded nodes. Each guarded node has annotated information such as the keys that should be used to encrypt and decrypt the corresponding document sub-tree. The evaluator is also in charge of generating the required keys, as well as optimizing the protection tree structure to avoid duplication of ciphertexts and to reduce the number of required keys.

The protected XML tree is then fed into an encryption module, which encrypts the guarded elements as required by the protection tree. In [61], a symmetric encryption algorithm is used, namely AES encryption with 128-bit keys. However, any other symmetric key encryption algorithms can be used within the framework. The encrypted tree is released, and users subsequently request keys through a secure channel.

The access control policies are specified using policy queries, expressed in a language that derives from XQuery. The following example illustrates a policy query that restricts access to student grades to
faculty members only:

```
SUFFICIENT
FOR $x$ in /students/subjects
KEY getKey("FacultyKey")
TARGET $x$/grades
```

The SUFFICIENT condition specifies that the condition stipulated by the subsequent policy query is sufficient to access the grades objects. However, the condition is not necessary. For instance, another query may specify that secretaries in each department may also be allowed access to grades. Or, with a more complex policy query, students may be authorized to access their own grades, while they are forbidden to access grades of other students. The language also permits the declaration of a policy query as NECESSARY, which means that the data element cannot be accessed if the key stipulated in the query is not held by the principal. Such queries are useful when an application requires strict auditing of access to a certain data object, e.g., hospital patients with HIV.

The FOR clause identifies the objects (i.e., XML elements) to which the policy applies, whereas TARGET indicates which exact elements within the qualifying objects are encrypted. Finally, the KEY clause specifies the key that is required to access encrypted data. The getKey function instructs the subject to retrieve the key with identifier “FacultyKey” from the data owner. Before granting access to the key, the data owner will verify (using a conventional access control mechanism, for instance) that the principal has the proper credentials to receive the key (e.g., the principal is an actual faculty member).

The keys that are used for encryption can belong to three categories: exchange keys, data keys and inner keys. To understand how encryption and key discovery is performed, consider the example in Figure 9.2. The exchange keys are the simplest case, and they are retrieved by connecting to the data owner, as shown in the previous student grades policy query. Denote by $k_0$ the key reserved for faculty members. Data keys allow more flexibility to access control. For instance, consider that the data owner wants to control the access pattern of operations. Continuing the previous example, the data owner may allow faculty members
to browse through the dataset of grades, i.e., retrieve the entire set of student grades. On the other hand, a student should only access their own grades. Since key management may become an issue with a large number of students, a better approach is to protect the document using information that only individual students know. For instance, the key may be derived from the matriculation number and the date of birth (DOB) of the student. (It is assumed that the combination of matriculation number and DOB is not known by other students, and that the space of these identifiers is sparse enough to disallow a brute-force attack where an adversary tries all identifiers in sequence.) The following query illustrates this case:

```
SUFFICIENT
FOR $x in /students/subjects
KEY $x/Matriculation/text()
   $x/DOB/text()
TARGET $x/grades
```

Data keys are derived directly from the data at encryption time. For instance, the matriculation number together with day of birth may be concatenated and then split into blocks of 128 bits. The resulting blocks are then logically XOR’d, and the XOR result represents the encryption key. Authorized students can reconstruct the key without having to interact with the data owner, and access their own grade. In Figure 9.2, the key $k_s$ is computed based on the two shares $s_1 \oplus s_2$. 
9.1 Encryption-based Access Control for XML Documents

The previous use case scenario considered two categories of subjects: faculty that can access all grades, and students that can access their own grades only. However, the two categories of subjects hold different keys, and the key that is used by faculty cannot be shared with students. One approach would be to include the grades information twice in the published data, encrypted with two different keys. However, this method is not scalable, since, in general, many different subjects (or types of subjects, e.g., roles) may need to access the same data object. The use of inner keys solves this problem. The main idea behind inner keys is that each data object is encrypted only once with a key \( k \), but there are multiple ways to get in the possession of \( k \). Continuing the earlier example, each student grade is encrypted with a key \( k_s \) obtained from the student matriculation number and day of birth. The document tree is enlarged with a metadata element that contains the value of \( k_s \) encrypted with key \( k_0 \), which corresponds to the faculty key. In other words, each student grade is encrypted exactly once (with a distinct key for each student), and a master key \( k_0 \) is used to decrypt each individual key. The key \( k_s \) acts as an inner key, since it is encapsulated (in encrypted form) inside the document. Note that, in this example, \( k_s \) is both a data key and an inner key. In other cases where subjects need to access the same document sub-tree using distinct credentials, it is possible to have “pure” inner keys, which cannot be derived from the data.

Since the access control policies that govern access to data tend to be complex in large-scale environments such as the Internet, it is important to develop an authorization enforcement mechanism that is both effective and efficient. As a result of rich policy queries, data elements may be protected by complex combinations of keys that can be expressed in boolean form. For instance, expression \( k_1 \lor (k_2 \land k_3) \) specifies that the corresponding element can be accessed by a principal that possesses either key \( k_1 \), or both keys \( k_2 \) and \( k_3 \). For space efficiency, there should be no duplication of ciphertexts (i.e., the same data element should not appear twice encrypted with two different keys). Furthermore, the encryption algorithms take as input a single key, so the protection key expression must be reduced to an atomic key. To that extent, each logical tree is normalized, i.e., brought to a simplified form composed of
atomic keys only. Subsequently, metadata tree nodes are added that help enforcing the required policy with the use of inner keys.

The work in [14] also addresses the problem of secure publishing of XML documents, and proposes a formal framework that addresses issues ranging from minimizing the number of required keys to supporting key distribution based on principal attributes (e.g., roles) in addition to principal identities. Specifically, the authors formulate a number of requirements for access control in data publishing, including:

- **Varying granularity of protection levels.** The same access control policies may apply to several distinct documents, and at different levels in the hierarchy. For instance, in some cases the policy applies to the entire document, whereas in others it applies only to selected elements.

- **Content-based access control.** Often, the document structure or element types are not sufficient to characterize the degree of sensitivity associated with a document. Instead, the document contents may dictate the level of protection required.

- **Heterogeneity of principals.** Large scale dissemination of data implies that a large number of principals with diverse characteristics will access the data. Therefore, it is important to devise mechanisms that enforce access control based on principal properties, or attributes, rather than individual identities.

The system architecture is similar to [61], and consists of three steps: (i) a document marking step in which the protected portions are identified and the necessary keys are created, (ii) the actual encryption step and (iii) the document dissemination and key retrieval step. Within this framework, the access control policies can be specified either with respect to individual documents, a collection of documents, or with respect to a document schema (e.g., DTD), which in turn establishes the policy for all instances of the schema. In addition, the propagation of access control permissions can be fine-tuned. For instance, in a hierarchical, interlinked structure of XML documents and DTDs, permissions may be inherited from items at a higher level of the hierarchy, but this aspect is not mandatory. Furthermore, the number of propagation levels
within the hierarchy can also be controlled using a *depth* parameter. Specifically, if the depth level is equal to zero, then no propagation is performed, and the permissions are applied only to the current element in the hierarchy. In contrast, if the propagation depth is set to *, then propagation is performed all the way down to the leaf level. Depth can also take any integer value, setting the number of propagation levels to that value.

An important contribution of [14] is the consideration of credentials and credential types for authorization in data publishing. Credentials consist of attribute-value pairs, where attributes have well-defined types, and are associated with concepts such as the role of a principal within an organization, nationality of an individual, etc. Similar to documents, credentials are also organized in a hierarchy. Credentials can be used in expressions, based on operations such as exact match, inequality, string similarity, etc.

Finally, the key distribution mechanism is approached in [14] from both an on-line and an off-line perspective. The *on-line distribution* mode comprises of two alternative approaches: key packaging and secure channels. In the former case, the keys are encapsulated in the same message with the document, but the values of the keys are encrypted with the public key of the principal for which the keys are intended.\(^1\) However, sending the keys over an unsecure channel (even in encrypted form) may not be desirable. Therefore, the latter approach uses a separate secure channel for disseminating keys. In the off-line distribution mode, a directory service such as LDAP is used. The directory maintains the hierarchical structure of credentials, and in each of the leaf nodes of the directory the appropriate keys are stored. Principals get access to keys only if their credentials are properly verified at each level of the directory.

### 9.2 Privacy-preserving Access Control Mechanisms

The cryptographic access control methods discussed so far protect the confidentiality of the data. However, in most cases, principals have

---

\(^1\) Note that, public-key encryption is necessary in this case.
to explicitly request the encryption key from the data owner (with the exception of data keys, but the assumption that principals already know part of the data holds for restricted cases only). Since the keys are directly related to the corresponding data items, the data owner will immediately learn what data items a principal is interested in based on the requested keys. Such disclosure of principal interests may be detrimental in many situations.

Consider the example of a DNA database that stores characteristics of chromosome sequences and associated medical disorders. Such databases are extremely valuable, and researchers pay expensive fees to access the records. Therefore, access is most likely done on a record-by-record basis, and each record must be encrypted with a distinct key. However, if a principal requests a certain key, the data owner can immediately learn which DNA sequence the principal plans to access. This can lead to further disclosure of confidential details about the research that the querying principal is conducting.

The work in [22] addresses the difficult challenge of enforcing access control while at the same time not allowing the data owner to learn which data item is being accessed. At first sight, such an endeavour seems quite impossible, since authorization depends on the object that is accessed. Not knowing what the principal is asking for means that the data owner does not know what result to return, let alone which authorization rule to apply. Fortunately, advances in cryptographic protocols enable oblivious transfer [63], i.e., private retrieval of data items from a server. The oblivious access control protocol that we present, called $AC$-$OT$, relies on elliptic curve public key cryptography using bilinear maps. $AC$-$OT$ is secure under the bilinear Diffie–Hellman exponent and the strong Diffie–Hellman assumptions (see Section 2.2).

Similar to the work in [14], the access control model in [22] grants authorization based on the attributes of the principal. Specifically, there is a finite set $C$ that represents the universe of possible credentials, and each principal is associated with a maximum number $\ell$ of credentials in $C$.

Note that, within the oblivious access control framework, the data owner cannot issue itself credentials to principals, since this would violate principal anonymity. Therefore, a third party trusted credential
issuer entity is required. The issuer (I for short) sets up before-hand a public–private key pair \((pk_I, sk_I)\). Principals that require access to the data contact the issuer first, and they present to the issuer their credentials, as shown in Figure 9.3(a). The issuer verifies the credentials and encrypts them using the secret key \(sk_I\). The transformation performed by the issuer also acts as a credential signature, which the database later checks to ensure that the credentials are valid, and that they have been verified by the issuer. The encrypted credential of principal \(P\) is denoted by \(cred_P\). Note that, it is computationally unfeasible to recover the plaintext credentials of \(P\) from \(cred_P\) without knowing the issuer’s secret key \(sk_I\).

The database comprises of a set of indexed records \(\{R_i\}_{1 \leq i \leq N}\). Note that, this does not restrict applicability, since non-relational data (e.g., XML documents) can be broken down to atomic units with the same level of protection, and each such unit can represent a data item. As shown in Figure 9.3(b), each data item has an associated access control
list which comprises of a set of at most $\ell$ credentials. The list for data item $i$ is represented as $ACL_i = \{C_1, \ldots, C_\ell\} \in C^\ell$. The authorization semantics is such that a principal is allowed access to data item $i$ only if she/he possesses all credentials within $ACL_i$. The data owner generates a public-private key pair $(pk_{DB}, sk_{DB})$ and encrypts the data with the private key. The public key of the issuer $pk_I$ is also used in the encryption process, to facilitate data transfer as shown later. The owner publishes the encrypted version of the data which contains for each record $i$ the record ciphertext $ER_i$. Note that, the encrypted version encapsulates the access control information.

After obtaining the (encrypted) credentials from the issuer, the principal engages in a protocol with the data owner to retrieve the contents of the required records (Figure 9.3(c)). Note that, the encrypted database is already available to the users, since it is published. However, the cleartext contents are not. Assume that the principal wishes to access record $i$. The input of the protocol executed by the user is the encrypted record $ER_i$, and the public key of the database $pk_{DB}$. The database first checks the signed credentials, which prove that the principal has the proper authorization to access some record (unknown to the database). Next, the principal and the database engage in an interactive zero-knowledge proof of knowledge (ZKPK) protocol (see Section 2.2) following which the database owner assists the principal in decrypting record $R_i$, without knowing the actual $ER_i$. Due to the signature of the issuer, which is also an input to the protocol, the principal is only able to access the record she/he has credentials for. If she tries to access a different record, the ZKPK protocol will not correctly decrypt that record.

The work in [81] considers the same scenario as [22], i.e., it provides both data confidentiality and privacy of principals. In addition, [81] proposes a novel and efficient mechanism for key management, which does not require the expensive ZKPK step executed by the previous approach for each individual access.

The system architecture is similar to the one in Figure 9.3, where the issuer functionality is achieved through an identity manager (IdMgr) component. The environment envisioned by [81] is that of publish-subscribe systems: a publisher (Pub) releases a set of documents, and
9.2 Privacy-preserving Access Control Mechanisms

Each document comprises of several sub-documents, or sections. Each section may have its own confidentiality requirements, and is therefore encrypted with a distinct key. In fact, it is possible for multiple users with different credentials to require access to the same document subsection. For this reason, the Pub keeps a reverse mapping of subscribers for each section. However, the Pub does not know the real identities of principals, but only their pseudonyms.

Principals, or subscribers (Subs), are authorized based on their identity attributes. Specifically, each principal has an associated list of attribute name-value pairs, e.g., Age = 34, or Role = Professor. Authorization constraints are expressed as access control policies (acp), where each policy is represented by a conjunction of attribute conditions, e.g., Age > 30.

Figure 9.4 shows the structure of a document $D$. There are multiple sections $D_1, \ldots, D_3$, and each section is associated with one or more policies. The set of acp for each section abide an “OR” semantics, i.e., a principal must satisfy the constraints in any of the acp to gain access to the corresponding section. The set of policies associated with a certain document section $D_i$ is denoted as the policy configuration of $D_i$. For each policy configuration a symmetric key is created, which is used to encrypt all sections having that configuration. For instance, sections $D_1$ and $D_3$ share the same configurations $\{acp_1, acp_3\}$.

Principals need to learn the values of the keys for sections they are authorized to access. However, principals do not want to release their identity attributes to the Pub. Therefore, they obtain encrypted and signed versions of their credentials from IdMgr. The IdMgr creates a Pedersen commitment \cite{72} for each identity attribute of the principal.

<table>
<thead>
<tr>
<th>Document Sections</th>
<th>Policy Configurations</th>
<th>Access Control Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$: acp, acp_3</td>
<td>${acp, acp_3}$</td>
<td>acp: Age&gt;30</td>
</tr>
<tr>
<td>$D_2$: acp, acp_3</td>
<td>${acp, acp_3}$</td>
<td>acp: Age&lt;40 &amp; Role=Mgr</td>
</tr>
<tr>
<td>$D_3$: acp, acp_3</td>
<td></td>
<td>acp: Salary &gt; 100,000</td>
</tr>
</tbody>
</table>

Fig. 9.4 Document sections, policies and policy configurations.
The commitments are generated within a finite group $G$ of order $p$. Denote by $g$ and $h$ two generators of $G$ such that $h = g^\alpha$, where $\alpha$ is a secret known only to $IdMgr$. Note that, due to the intractability of finding discrete logarithms (see Section 2.2), the value of $\alpha$ can not be recovered by any entity other than $IdMgr$. An identity attribute with value $x$ is encoded as

$$c = g^x h^r$$

where $r$ is a random number in $\{0, 1, \ldots, p - 1\}$. The principal receives an identity token that consists of a unique pseudonym, the value of the commitment, and signature $\sigma$ that proves that the token was indeed issued by the $IdMgr$:

$$IT = (\text{pseudonym}, \text{AttributeName}, c, \sigma)$$

To retrieve a particular key $K$ associated with an identity attribute condition, the principal contacts the publisher and together they execute a Oblivious Commitment-Based Envelope (OCBE) protocol [55] that allows the principal to extract the key information from the messages received from the publisher based on the value of commitment $c$. OCBE supports typical operations used in attribute conditions such as $=, \neq, \leq, \geq$.

The work in [81] proposes an innovative mechanism for the management of keys. Recall that according to the document and acp organization in Figure 9.4, only the key associated with one of the policy configurations of section $D_i$ is necessary to access it. Therefore, instead of running the key retrieval protocol for all keys that a principal is entitled to, it suffices to retrieve a select set of keys that ensure access to all sections for which the principal is authorized.

### 9.3 Summary

Protection of published data (e.g., XML documents) is enforced through encryption of selected document sections, and distribution of keys to authorized users only. Principals may explicitly request keys, and obtain access to protected data based on their credentials, e.g., identity attributes. Furthermore, in cases where protection of data access patterns is required, principals have the ability to privately
retrieve decryption keys with the help of zero-knowledge proof of knowledge protocols.

An interesting direction for future work is mediating heterogeneous access control policies when documents contain data from multiple sources. A related research topic worth investigating is authorization in the presence of distributed modification of documents. In the domain of privacy-preserving authorization, enhancing the performance of the authorization mechanism is an important problem to study, since currently-employed encryption functions with advanced functionality may require an expensive overhead in terms of both computational and communication cost.
Concluding Remarks and Research Directions

Data security and in particular protection of data from unauthorized accesses remain important goals of any DBMS. In this monograph, we have covered the fundamentals of access control for database systems and access control systems provided as part of current DBMSs. We have also discussed research approaches motivated by new security requirements, such as protection from insider threats, and new contexts for data usage, such as data dissemination and publishing.

An important direction is represented by access control for mobile users. Users are today increasingly mobile and have a large variety of devices available to them. Moreover, the deployment of computing power and sensors in every-day environments will make it possible for users to be always connected, sometimes without even being aware of it. In such contexts, several issues are relevant. Users execute many more activities online; information about user identities, profiles, credentials, and permissions is more frequently required. Such information needs to be secure and reliable; reliable user identification is increasingly crucial. It is thus important on one side to develop techniques for efficient storage of access control information on small devices; a relevant example in this respect is represented by the notion of portable access rights
recently proposed by Bykova and Atallah [20]. On the other side, it is important that access control mechanisms be integrated with standards for identity management [57] as well as with trust negotiation techniques [15].

Because large-sized streams of data are generated in such environments, efficient techniques for access control must also be devised and integrated with processing techniques for continuous queries. A preliminary “stream-centric” approach has been proposed by Nehme et al. [64] for streaming data management systems (DSMSs). In this approach authorizations are not persistently stored on the DSMS server, but rather streamed together with the data. Here, authorizations are expressed via security constraints (called security punctuations) and embedded into data streams. Therefore, the party streaming the data, that is, the data source, is able to dynamically change authorizations and stream the authorizations together with the data. These authorizations are then used for access control at the server side. This approach is highly flexible and dynamic and, as shown by experimental results, also efficient. However, it only supports a simple access control model based on RBAC. It would be important to support security punctuations able to encode access control policies expressed in more complex languages, like XACML. The use of a language like XACML in the context of streaming data requires being able to encode XACML policies in structures that require very little space and are thus amenable for embedding in data streams. Also efficient techniques for enforcing these policies at the server side are required, as a conventional approach based on invoking an external XACML engine would not be viable.

A second important direction is represented by the development of access control systems specifically tailored to support privacy-sensitive information. In addition to approaches to fine-grained access control that we have discussed in Section 5, other preliminary approaches have been proposed dealing with access control systems specifically tailored to enforce privacy policies, such as the policies that can be expressed by using the P3P standard [88]. An important feature of such approaches is the use of privacy metadata consisting of privacy policies and privacy authorizations stored in privacy-policy tables and privacy-authorization tables, respectively. The privacy policies define
the intended use, the external-recipients, and retention period for each attribute of a table, while the privacy authorizations define the authorized users. Such approaches also add a special attribute, “purpose,” to each table, which encodes the set of purposes users, to whom the data are referred, agree with during the data collection process. It is important to note that purpose is a different notion with respect to the notion of role, in that purposes characterize actions executed on data whereas roles characterize users. Moreover, though purposes may be considered a form of labels to be associated with data and, thus, similar to labels used in multi-level databases, approaches to purpose management, like the approach by Byun et al. [21], have some important differences with respect to label-based approaches developed as part of MLS. These approaches support the association of multiple purposes with the same data item and, thus, are not restricted to a single label, and the specification of negative purposes, specifying that certain data items should not be used for a given set of purposes. Despite these initial approaches, access control mechanisms need extensions, such as support for retention and privacy obligations, in order to be effective for data privacy.

We conclude by mentioning that even though access control represents an important security mechanism, comprehensive approaches to data security have to address not only data confidentiality and privacy, but also data integrity and availability. Also novel computing platforms, such as cloud computing, require specialized security techniques. Substantial research efforts are thus required to develop effective and efficient data security solutions that address all the security requirements in different computing platforms and environments.
References

References


References


References


References


