Proceedings of 3rd ACM Workshop on Role-Based Access Control, Fairfax, Virginia, October 22-23, 1998

How to do Discretionary Access Control Using Roles*

Ravi Sandhu and Qamar Munawer
Laboratory for Information Security Technology and
Information and Software Engineering Department
George Mason University

Abstract

Role-based access control (RBAC) is a promising alternative to traditional discretionary access control (DAC) and mandatory access control (MAC). The central idea of RBAC is that permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. RBAC is policy neutral in that the precise policy being enforced is a consequence of how various components of RBAC—such as role hierarchies, constraints and administration of user-role and role-permission assignment—are configured. This raises the important question as to whether RBAC is sufficiently powerful to simulate DAC and MAC. Simulation of MAC in RBAC has been demonstrated earlier by Nyanchama and Osborn and by Sandhu. In this paper we demonstrate how to simulate several variations of DAC in RBAC, using the well-known RBAC96 model of Sandhu et al. In combination with earlier work we conclude that RBAC encompasses both MAC and DAC.

1 Introduction

The concept of role-based access control (RBAC) began with multi-user and multi-application on-line systems pioneered in the 1970s. The central notion of RBAC is that permissions are associated with roles, and users are assigned to appropriate roles. This greatly simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed.

Although the basic concept of RBAC has been around for some time, it is only recently that security researchers have studied it rigorously. There are different aspects to RBAC that are dealt with in different ways in different systems. This makes it difficult to arrive at a consensus definition of RBAC. Sandhu et al [SCFY96, San97] introduced the well-known RBAC96 model family which provides a general framework within which variations of RBAC can be accommodated. RBAC96 provides us the machinery to settle the kinds of questions addressed in this paper.

A major attribute of RBAC is that it is policy neutral. RBAC provides support for several important security principles (notably least privilege, privilege abstraction and separation of duties), but does not dictate how, or even if, these should be put into practice. The precise policy enforced in RBAC is a consequence of the detailed configuration of various components of RBAC, such as role hierarchies, constraints and administration of user-role and role-permission assignment.

This raises an important question regarding the expressive power of RBAC. In particular, can RBAC enforce traditional mandatory access control[1] (MAC) and discretionary access control[2] (DAC) policies? Nyanchama and Osborn, and Sandhu have earlier shown how to simulate several variations of MAC in RBAC [NO96, San96]. In this paper we show how to simulate a variety of DAC policies in RBAC, particularly in RBAC96. This result is of theoretical interest because it relates RBAC to the most dominant form of access control. It is also of practical significance because it allows DAC policies to be implemented in systems that are predomi-

*This work is partially supported by grant CCR-9503560 from the National Science Foundation at the Laboratory for Information Security Technology at George Mason University.

All correspondence should be addressed to Ravi Sandhu, ISE Department, Mail Stop 4A4, George Mason University, Fairfax, VA 22030, `sandhu@isse.gmu.edu`, `www.list.gmu.edu`.

[1]We define MAC to be equivalent to lattice-based access control such as described in [San93].

[2]We define DAC to be access control based on ownership such as described in [SS94, SS97].

nantly RBAC oriented. This could be very useful in real systems for doing some amount of DAC with respect to selected objects in an otherwise non-discretionary RBAC environment. Also, coupled with the earlier MAC to RBAC96 constructions, we can now assert that both classical forms of access control are within the purview of RBAC.

The rest of the paper is organized as follows. We begin with a quick review of RBAC96 in section 2. In section 3 we define various discretionary access control policies. Section 4 describes the simulation of these policies in RBAC96. The final section gives our conclusions.

2 The RBAC96 Model

A general family of RBAC models called RBAC96 was defined by Sandhu et al [SCFY96, San97]. Figure 1 illustrates the most general model in this family. For simplicity we use the term RBAC96 to refer to the family of models as well as its most general member.

The top half of figure 1 shows (regular) roles and permissions that regulate access to data and resources. The bottom half shows administrative roles and permissions. Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role, and a permission is an approval of a particular mode of access to one or more objects in the system or some privilege to carry out specified actions.

Roles are organized in a partial order, so that if $x > y$ then role x inherits the permissions of role y, but not vice versa. In such cases, we say x is senior to y. By obvious extension we write $x \geq y$ to mean $x > y$ or $x = y$.

Each session relates one user to possibly many roles. The idea is that a user establishes a session (e.g., by signing on to the system) and activates some subset of roles that he or she is a member of. In the constructions of this paper the session concept is not really used. Rather it is assumed that all roles of a user are automatically activated in every session. This assumption is consistent with RBAC96 where it amounts to a constraint on the *roles* function.

3 Variations of DAC

In this section we discuss DAC policies that will be considered in this paper. The central idea of DAC is that the owner of an object, who is usually its creator, has discretionary authority over who else can access that object [SS94, SS97]. In other words the core DAC policy is owner-based administration of access rights. There are many variations of DAC policy, particularly concerning how the owner's discretionary power can be delegated to other users and how access is revoked. This has been recognized since the earliest formulations of DAC [Lam71, GD72].

Our approach in this paper is to identify major mainstream variations of DAC and demonstrate their construction in RBAC. The constructions are such that it will be obvious how they can be extended to handle other related DAC variations. This is an intuitive, but well-founded, justification for the claim that DAC can be simulated in RBAC.[3]

The DAC policies we consider in this paper all share the following characteristics.

- The creator of an object becomes its owner.
- There is only one owner of an object.[4] In some cases ownership remains fixed with the original creator, whereas in other cases it can be transferred to another user.
- Destruction of an object can only be done by its owner.

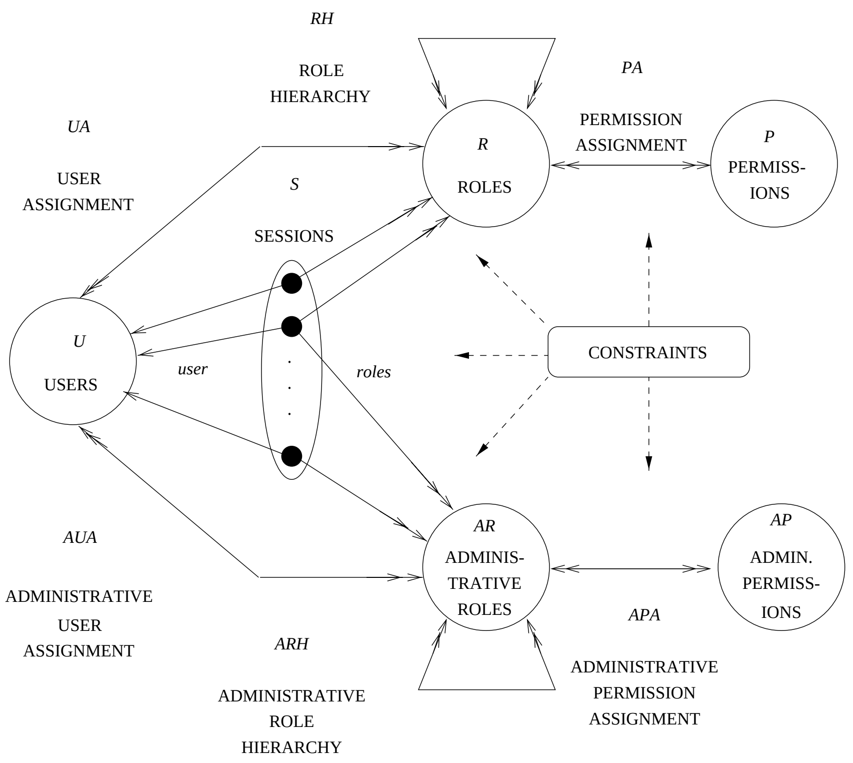
With this in mind we now define the following variations of DAC with respect to granting of access.

1. **Strict DAC** requires that the owner is the only one who has discretionary authority to grant access to an object and that ownership cannot be transferred. For example, suppose Alice has created an object (Alice is owner of the object) and grants read access to Bob. Strict DAC requires that Bob cannot propagate access to the object to another user.[5]
2. **Liberal DAC** allows the owner to delegate discretionary authority for granting access to an object to other users. We define the following variations of liberal DAC.

[3]A formal proof would require a formal definition of DAC encompassing all its variations, and a construction to handle all of these in RBAC96. Models such as HRU [HRU76], SPM [San88] and TAM [San92] could be used as general "DAC" models for this purpose. Such a construction would be a useful formal confirmation of our intuitive approach in this paper, but is outside the scope of the present paper.

[4]This assumption is not critical to our constructions. It will be obvious hoe mutiple owners can be handled. Assuming a single owner is convenient and simplifies our exposition.

[5]Of course, Bob can copy the contents of Alice's object into an object that he owns, and then propagate access to the copy. This is why DAC is unable to enforce information flow controls, paricularly with respect to Trojan Horses.



- U, a set of users
  R and AR, disjoint sets of (regular) roles and administrative roles
  P and AP, disjoint sets of (regular) permissions and administrative permissions
  S, a set of sessions
- $UA \subseteq U \times R$, user to role assignment relation
  $AUA \subseteq U \times AR$, user to administrative role assignment relation
- $PA \subseteq P \times R$, permission to role assignment relation
  $APA \subseteq AP \times AR$, permission to administrative role assignment relation
- $RH \subseteq R \times R$, partially ordered role hierarchy
  $ARH \subseteq AR \times AR$, partially ordered administrative role hierarchy
  (both hierarchies are written as $\geq$ in infix notation)
- $user : S \rightarrow U$, maps each session to a single user (which does not change)
  $roles : S \rightarrow 2^{R \cup AR}$ maps each session s_i to a set of roles and administrative roles $roles(s_i) \subseteq \{r \mid (\exists r' \geq r)[(user(s_i), r') \in UA \cup AUA]\}$ (which can change with time)
  session s_i has the permissions $\cup_{r \in roles(s_i)} \{p \mid (\exists r'' \leq r)[(p, r'') \in PA \cup APA]\}$
- there is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Figure 1: Summary of the RBAC96 Model

(a) **One Level Grant:** The owner can delegate grant authority to other users but they cannot further delegate this power. So Alice being the owner of object O can grant access to Bob who can grant access to Charles. But Bob cannot grant Charles the power to further grant access to Dorothy.

(b) **Two Level Grant:** In addition to a one-level grant the owner can allow some users to further delegate grant authority to other users. Thus, Alice can now authorize Bob for two-level grants, so Bob can grant access to Charles, with the power to further grant access to Dorothy. However, Bob cannot grant the two-level grant authority to Charles.[6]

(c) **Multilevel Grant:** In this case the power to delegate the power to grant implies that this authority can itself be delegated. Thus Alice can authorize Bob, who can further authorize Charlie, who can further authorize Dorothy, and so on indefinitely.

3. **DAC with Change of Ownership:** This variation allows a user to transfer ownership of an object to another user. It can be combined with strict or liberal DAC in all the above variations.

For revocation we consider two cases as follows.

1. **Grant-Independent Revocation:** Revocation is independent of the granter. Thus Bob may be granted access by Alice but have it revoked by Charles.

2. **Grant-Dependent Revocation:** Revocation is strongly tied to granter. Thus if Bob receives access from Alice, access can only be revoked by Alice.

In our constructions we will initially assume grant-independent revocation and then consider how to simulate grant-dependent revocation. In general, we will also assume that anyone with authority to grant also has authority to revoke. This coupling often occurs in practice. Where appropriate, we can decouple these in our simulations because, as we will see, they are represented by different permissions.

These DAC policies certainly do not exhaust all possibilities. Rather these are representative policies whose simulation will indicate how other variations can also be handled.

[6]More generally, we could consider a n-level grant but it will be obvious how to do this from the two level construction.

4 DAC Variations in RBAC96

To specify the above variations in RBAC it suffices to consider DAC with one operation, which we choose to be the read operation. Similar constructions for other operations such as write, execute and append, are easily possible.[7] Before considering specific DAC variations, we first describe common aspects of our constructions.

4.1 Common Aspects

The basic idea in our constructions is to simulate the owner-centric policies of DAC using roles that are associated with each object.

4.1.1 Create an Object

For every object that is created in the system we require the simultaneous creation of three administrative roles and one regular role as follows.

- Three administrative roles: OWN_O, PARENT_O and PARENTwithGRANT_O
- One regular role: READ_O

The relationship between these roles is shown in figure 2. Figure 2(a) indicates that the role OWN_O can add users to the role PARENTwithGRANT_O which in turn can add users to the role PARENT_O and so on.[8] Each role also has the power to revoke users from the following role in this chain. Figure 2(b) shows the seniority relation between the three administrative roles, so OWN_O inherits all permissions of PARENTwithGRANT_O which in turn inherits permissions of PARENT_O.

In addition we require simultaneous creation of the following eight permissions along with creation of each object O.

- canRead_O: authorizes the read operation on object O. It is assigned to the role READ_O.
- destroyObject_O: authorizes deletion of the object. It is assigned to the role OWN_O.

[7]The copy operation can be viewed as a read of the original object and a write (and possibly creation) of the copy. It can be useful to associate some default permissions with the copy. For example, the copy may start with access related to that of the original object or it may start with some other default. Specific policies here could be simulated by extending our constructions. As we have said earlier, we need to work with some formal model of DAC such [HRU76, San88, San92] to argue that all possible extensions have been considered.

[8]Strictly speaking we mean that users who are members of the role can carry out the indicated action, but for simplicity we say the role carries out the actions.

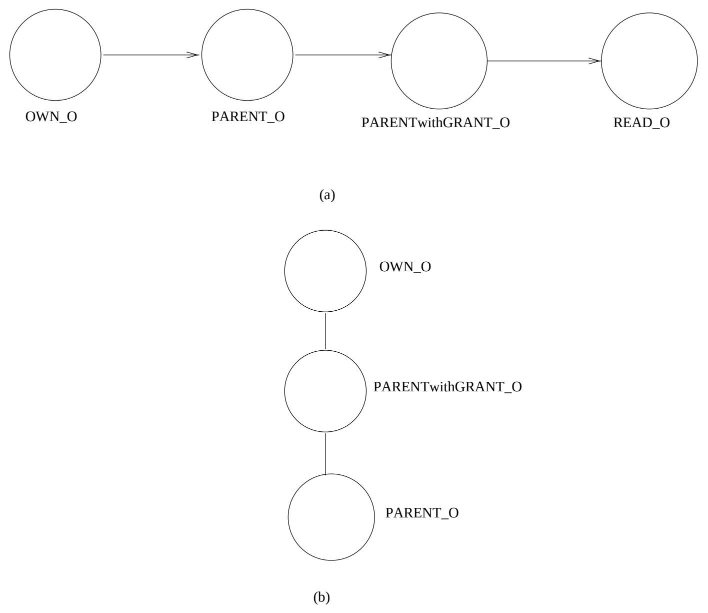


Figure 2: (a)Administration of roles associated with an object (b) Administrative role hierarchy

- addReadUser_O, deleteReadUser_O: respectively authorize the operations to add users to the role READ_O and remove them from this role. They are assigned to the role PARENT_O.
- addParent_O, deleteParent_O: respectively authorize the operations to add users to the role PARENT_O and remove them from this role. They are assigned to the role PARENTwithGRANT_O.
- addParentWithGrant_O, deleteParentWithGrant_O: respectively authorize the operations to add users to the role PARENT_O and remove them from this role. They are assigned to the role OWN_O.

These permissions are assigned to the indicated roles when the object is created and thereafter they cannot be removed from these roles or assigned to other roles.

In RBAC96 the behavior described above would be enforced by the constraints mechanism. For example, one constraint on the permissions would be that they are automatically associated with the roles at the time of their creation. Another constraint would be that the three administrative roles associated with object O have the seniority relationship shown in figure 2(b).

4.1.2 Destroy an Object

Destroying an object O requires deletion of the four roles namely OWN_O, PARENT_O, PARENTwithGRANT_O and READ_O and the eight permissions (in addition to destroying the object itself). This can be done only by the owner, by virtue of exercising the destroyObject_O permission.

4.2 Strict DAC

In strict DAC only the owner can grant/revoke read access to/from other users. The creator is the owner of the object. By virtue of membership (via seniority) in PARENT_O and PARENTwithGRANT_O, the owner can change assignments of the role READ_O. Membership of the three administrative roles cannot change, so only the owner will have this power. This policy can be enforced in RBAC96 by imposing a cardinality constraint of 1 on OWN_O and of 0 on PARENT_O and PARENTwithGRANT_O.

This policy could be simulated using just two roles OWN_O and READ_O, and giving the addReadUser_O and deleteReadUser_O permissions directly to OWN_O at creation of O. For consistency with subsequent variations we have introduced all required roles from the start.

4.3 Liberal DAC

The three variations of liberal DAC described in section 3 are now considered in turn.

4.3.1 One-Level Grant

The one-level grant DAC policy can be simulated by removing the cardinality constraint of strict DAC on membership in PARENT_O. The owner can assign users to PARENT_O role who in turn can assign users to the READ_O role. But the cardinality constraint of 0 on PARENTwithGRANT_O remains.

4.3.2 Two-Level Grant

In the two level grant DAC policy the cardinality constraint on PARENTwithGRANT_O is also removed. Now the owner can assign users to PARENTwithGRANT_O who can further assign users to PARENT_O. Note that members of PARENTwithGRANT_O can also assign users directly to READ_O, so they have discretion in this regard. Similarly the owner can assign users to PARENTwithGRANT_O, PARENT_O or READ_O as deemed appropriate.[9]

4.3.3 Multilevel Grant

To grant access beyond two levels we authorize the role PARENTwithGRANT_O to assign users to PARENTwithGRANT_O. We achieve this by assigning the addParentWithGrant_O permission to the role PARENTwithGRANT_O when object O is created. As per our general policy of coupling grant and revoke authority, we also assign the deleteParentWithGrant_O to the role PARENTwithGRANT_O when O is created. This coupling policy is arguably unreasonable in context of grant-independent revoke, so the deleteParentWithGrant_O permission could be retained only with the OWN_O role if so desired. For grant-dependent revoke the coupling is more reasonable.

4.4 DAC with Change of Ownership

Change of ownership can be easily accomplished by suitable redefinition of the administrative authority of a member of OWN_O. The transfer capability can be easily specified in RBAC96.

4.5 Multiple Ownership

Multiple ownership can also be accommodated by allowing users to be added to OWN_O. Since all members of OWN_O have identical power, including the ability to revoke other owners, it would be appropriate with grant-independent revoke to distinguish the original owner. Alternately, we can have grant-dependent revoke of ownership.

[9]N-level grants can be similarly simulated by having N roles, PARENTwithGRANT_O^{N-1}, PARENTwithGRANT_O^{N-2}, ..., PARENTwithGRANT_O, PARENT_O.

4.6 Grant-Dependent Revoke

So far we have considered grant-independent revocation where revocation is independent of granter. Now finally we consider how to simulate grant-dependent revoke in RBAC96. In this case only the user who has granted access to another user can revoke the access (with possible exception of the owner who is allowed to revoke everything).

Specifically, let us consider the one level grant DAC policy simulated earlier by allowing members of PARENT_O role to assign users to READ_O role. To simulate grant-dependent revocation with this one level grant policy we need a different administrative role U_PARENT_O and a different regular role U_READ_O for each user U authorized to do a one-level grant by the owner. These roles are automatically created when the owner authorizes user U. We also need two new administrative permissions created at the same time as follows.

- addU_ReadUser_O, deleteU_ReadUser_O: respectively authorize the operations to add users to the role U_READ_O and remove them from this role. They are assigned to the role U_PARENT_O.

Thereby, Ui_PARENT_O manages the membership assignments of Ui_READ_O role as indicated in figure 3 for users U1, U2, ..., Un. The cardinality constraint of U_PARENT_O is one. Moreover, its membership cannot be changed. Thus user U will be the only one granting and revoking users from U_READ_O role. The U_READ_O role itself is assigned the permission canRead_O at the moment of creation. As before all of this enforced by RBAC96 constraints.

We can allow the owner to revoke users from the U_READ_O role by making U_PARENT_O junior to OWN_O. Simulation of grant-dependent revocation can be similarly simulated with respect to PARENT_O and PARENTwithGRANT_O roles. Extension to multiple ownership is also possible.

4.7 Discussion

The nature of the RBAC96 simulations described above suggests that many other DAC variations could be similarly simulated. Some of these, such as multiple ownership with each owner having identical and autonomous authority, have been mentioned along the way. Other variations could include groups of users who are granted access as a single unit.

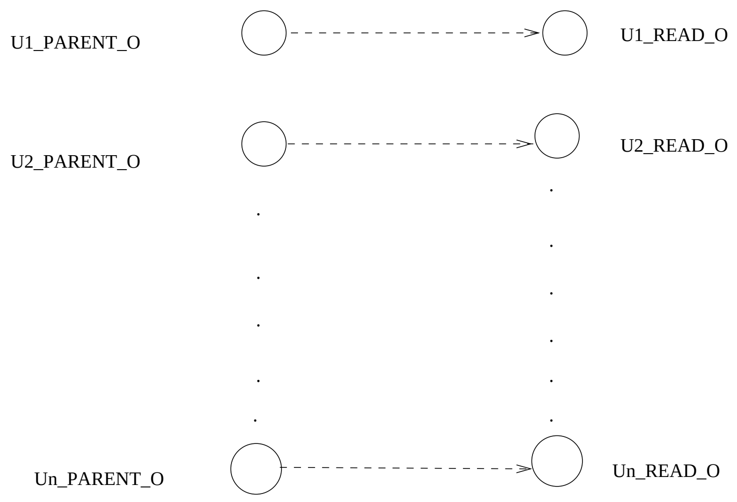


Figure 3: Read_O Roles associated with members of PARENT_O

Our constructions have been described rather informally in keeping with the intuitive simplicity of RBAC. These constructions can be formalized using RBAC96 notation, but there is probably not much benefit to be gained by this.

The number of roles in our constructions is high because several roles are needed for each object. This is due to separate discretion at the granularity of individual objects. In practice objects can be grouped together to overcome this. For instance, if discretionary authority over all objects owned by a user is uniformly granted to others, we need have only one set of administrative PARENT roles for that user. Grant-dependent DAC is even more complex.

Generally, DAC appears to be more complex to simulate in RBAC96 than MAC. While we do not expect DAC to be simulated in RBAC in this fine-grained manner in the normal course, our results tell us that theoretically this can be done in unusual circumstances as required. For practical applications DAC can be realistically provided within RBAC, but on reasonably sized collections of objects rather than on individual objects. Our results do confirm that RBAC is policy neutral and can certainly accommodate DAC and MAC, although it is more suited for the latter than the former.

5 Conclusion

In this paper we have shown how to simulate a variety of DAC policies in RBAC96. This fact is theoretically important, especially in conjunction with earlier results showing how to do MAC using roles [NO96, San96]. RBAC therefore subsumes both traditional forms of access control. The results of this paper also have practical significance, because they show how DAC can be accommodated within a RBAC oriented system. In particular DAC could apply to selected objects (such as a user's private objects) and not to others (such as objects belonging to the enterprise).

Finally, there is no generally accepted definition of DAC. Models such as HRU [HRU76], SPM [San88] and TAM [San92] could be used as general "DAC" models for this purpose. Reduction of one of these models to RBAC96 would be a valuable exercise to confirm the intuitive arguments and insights of this paper.

References

[GD72] G.S. Graham and P.J. Denning. Protection – principles and practice. In *AFIPS Spring Joint Computer Conference*, pages 40:417–429, 1972.

[HRU76] M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[Lam71] B.W. Lampson. Protection. In *5th Princeton Symposium on Information Science and Systems*, pages 437–443, 1971. Reprinted in *ACM Operating Systems Review* 8(1):18–24, 1974.

[NO96] Matunda Nyanchama and Sylvia Osborn. Modeling mandatory access control in role-based security systems. In *Database Security VIII: Status and Prospects*. Chapman-Hall, 1996.

[San88] Ravi S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432, April 1988.

[San92] Ravi S. Sandhu. The typed access matrix model. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, May 1992.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[San96] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In Elisa Bertino, editor, *Proc. Fourth European Symposium on Research in Computer Security*. Springer-Verlag, Rome, Italy, 1996. Published as *Lecture Notes in Computer Science, Computer Security–ESORICS96*.

[San97] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SS94] Ravi Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9):40–48, 1994.

[SS97] Ravi S. Sandhu and Pierangela Samarati. Authentication, access control and intrusion detection. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 1929–1948. CRC Press, 1997.