CONSTRAINTS-BASED ACCESS CONTROL

Wee Yeh Tan
School of Computing
National University of Singapore
Building S-16, Level 5, Room 05/08
3 Science Drive 2, Singapore 117543
Tel: +65 874 8850
Fax: +65 779 4580 *
tanwy@comp.nus.edu.sg

Abstract The most important aspect of security in a database after establishing the authenticity of the user is its access control mechanism. The ability of this access control mechanism to express the security policy can make or break the system.

This paper introduces constraints-based access control (CBAC) – an access control mechanism that general associations between users and permissions are specified by the rules (or constraints) governing the access rights of each user. This association is not restricted to static events but can include dynamic factors as well.

One of the many advantages of CBAC is that even a static CBAC is a generalisation of most of the access control mechanism in use today. We demonstrate how CBAC can efficiently simulate role-based access control (RBAC) and access control list (ACL). In fact, CBAC allows the introduction of any abstract concepts as one would do roles in RBAC. On top of that, CBAC also allows the users to specify interactions between these concepts.

Any flexibile access control method usually raises concerns over its time efficiency. We advocate the use of partial solutions to the access control constraints to improve the efficiency of CBAC.

Keywords: constraints, access control, security

1. INTRODUCTION

The security services does not stop after establishing the authenticity of a user. A secured system still has to determine if each user has access rights to the various resources within the system. The access control subsystem is, hence,

* Partial funding provided by a Strategic Research Programme on Computer Security funded by NSTB/MOE

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35587-0_24

M. S. Olivier et al. (eds.), *Database and Application Security XV*

tasked with limiting the activities of the authenticated user. While establishing the user's authenticity prevents unauthorized users from entering the system, the access control mechanism in the system prevents authenticated users from accessing unauthorized services or resources. This provides a finer grain of control over the use of the system.

The main issues surrounding the design of access control mechanism are its **efficiency** and **effectiveness**. For a mechanism to be effective, it need to be sufficiently expressive and at the same time help (not complicate) the management of permissions. While more advanced data structures and algorithms can be devised to improve efficiency, the question of effectiveness is better approached with a higher level specification language for programming the access control mechanism.

Several advances have been made to the area of access control specification and policies have been devised to narrow the gap between what the security adminstrators want and what the access control system can provide. The most common approaches are the **Discretionary Access Control (DAC)**, the **Lattice-based Access Control (LBAC)** and the **Role-based Access Control (RBAC)** [Sandhu and Samarati,].

DAC controls access by explicitly specifying the authorization for each user to each resource in the system. Implementations of DAC can be found in a variety of systems, from Unix's file access control, to ACL. While DAC is extremely flexible, management of access control permissions can be a pain due to the flat structure. DAC does not provide real assurance on the flow of information in a system. For example, there is no guarding against a user with the correct access rights leaking classified information to an unauthorized user.

LBAC, also known as Mandatory Access Control, enforces a direction in the flow of information in a lattice of security labels. In this approach, each user and resource is assigned a security clearance. A user can only read a resource of lower security clearance and the user can only write to a resource with a higher security clearance. While the write restriction may seem counter intuitive at first, this restriction is necessary to guarrantee that information only flows upwards in security clearance. This approach is not well accepted outside of the military due to its lack of flexibility.

The third approach, RBAC [Sandhu et al., 1996, Sandhu, 1996], models access control after the roles of the individuals in the organization. In this model, users are assigned the various roles they play in the organization and permissions are granted so that the roles can fulfill their task. Roles can be taken more broadly to include abstractions of users and permissions are to be granted to these abstractions. In [Sandhu, 1996], RBAC is shown to subsume LBAC. RBAC is widely adopted by several major databases like Oracle and Informix.

More recent development of web-based applications, Enterprise Resource Planning systems and other large scale systems that depends on a database backend requires a richer set of access control policy than is available today. Application developers are writing their own codes to provide the access control policies they wish to enforce where the access control mechanism provided by the database falls short. A typical web-application will include both codes to authenticate its users as well as codes that implements some access control policies before accessing the database on behalf of the user. More recent research like [Bertino et al., 1998] also confirms the need for a more expressive access control language.

This paper introduces Constraints based Access Control (CBAC), our approach to extensible access control. Constraints are used to specify general associations between the users and their authorizations. Further constraints may be placed between the environmental factors, e.g. time of day, and these factors can affect the outcome of the authorization.

The following section will introduce CBAC formally and explain the rational behind a constraints based approach. We will then explore the expressiveness of CBAC by simulating both ACL and RBAC, further explore the use of the rich syntax of CBAC, and introduce a flexible grant/revoke semantics supported directly by CBAC. Section 4 discusses implementation issues and presents an implementation model for developing the CBAC mechanism and the database server seperately. A list of future work is presented before we conclude the paper.

We conclude this introduction with a motivating example. In CBAC Inc, a pay clerk can access the pay table but only within working hours, and the access must be within an hour of the clearance given by the database adminstrator. The following constraints is imposed to the access of the said table:

- The user must play the role of the pay clerk
- The time of day must be within the working hours
- DBADM has given the clearance
- Clearance is within 1 hour.

2. CONSTRAINTS-BASED ACCESS CONTROL

Constraint programming is a declarative form of programming where the programmer specifies the problem in terms of constraints and an automatic solver finds solutions to the problem within the specified constraints [Marriott and Stuckey, 1998]. The mathematical nature of constraints allows the automatic solver to prune the search space by ensuring consistency between the available solutions. This feature also allows the solver to detect contradictions

in constraints without iterating through potential assignments. Several systems have been developed from Constraints Programming research and a handful, like iLog, has been successfully deployed in the industry.

There are several reasons constraints are a natural choice for specifying access control policy:

- Constraints can be used directly to specify relationships between entities, events and authorizations.
- Constraints has a strong theoretical foundation and has been applied to program verification. The same principles can likely be applied directly to policy verification.
- Constraint programming languages are turing-complete.
- Advances in the field of automatic constraint solving will have a direct and beneficial impact on this application.
- Techniques like constraint simplification can help weed out redundant rules. The simplified rules are likely to be easier to analyse.
- With the recent interest in Constraint Databases, a constraint based approach to access control like CBAC's matches exactly the tuple generating rules in CDB.

Direct specification of all access control policies in CBAC leads to better consistency in the final authorization with respect to the security policy. In essence, the security adminstrator will specify the policy as a set of constraints and the access control system, in this case the constraint solver, ensures that all accesses complies with the specified constraints. When a user attempts to access a resource, the constraint solver is invoked to check his permissions against the specified constraints.

Definition 1 *We define CBAC as follows:*

- *U, O, and P (users, resources and permissions respectively)*
- *$auth : U \times O \times P \rightarrow \{yes, no, partial\}$, a function that answers if the user is given some permission for a particular resource. 'partial' represents a partial solution when the system cannot determine at the said time whether the user should be allowed to access the resource. I.e. further information that is not available at the time of check is required to give an final answer.*

For purpose of presentation, we use a CLP based system [Jaffar and Lassez, 1987] and syntax to demonstrate the constraints in CBAC *. An authorization constraint, is a predicate in the form

```
auth(User, Resource, Permission) :- C_1, C_2, ... , C_n.
```

where `C_i` is some generic constraints. This predicate is invoked with the necessary parameters when an authorization is required.

The following are some examples of the use of CBAC/CLP.

- User 'adminstrator' is allowed access to every resource.

```
auth(adminstrator, _, _).
```

- All users are allowed all permissions to the tables they own.

```
auth(User, Resource, _) :-
   is_dbtable(Resource), own(User, Resource).
```

- A user can only read from or write to a row that contains his username from table 'particulars'.

```
auth(User, Tuple, [read, write]) :-
  Tuple = particulars(User, ... ),
```

- A pay clerk can access the pay table but only within working hours, and the access must be within an hour of the clearance given by the database adminstrator.

```
auth (User, Object, _) :-
   role(User, pay_clerk),
   date_time(Time), in_working_hour(Time),
   clearance(dbadm, Clearance),
   time_of_clearance(Clearance, T_Clear),
   time_diff(Time, T_Clear, T_Diff), T_Diff <= 60.
```

- List all resources that can be read by User `judge_dred`.

```
auth (judge_dred, Resource, read).
```

3. EXPRESSIVENESS OF CBAC

This section discusses the expressiveness of CBAC. In sections 3.1 and 3.2, we shall show that CBAC subsumes both ACL and RBAC by simulation. We also describe in these sections how the simulations can be done efficiently. Lastly, we shall discuss how CBAC can be used to handle policies like control flow and how GRANT/REVOKE can be supported natively by CBAC.

*CBAC does not limit the use of constraint solver to CLP

3.1. A CBAC simulation of ACL

ACL, or access control list, is a list bound to each resource in the system. The list indicates, for the particular resource, the access rights of each users.

Definition 2 *ACL can be viewed as a function defined formally as follows*

- *U, O, and P (users, resource, and permission respectively)*
- *$ACL : O \times U \rightarrow P$, a function mapping the resource O and user U to his access authorizations P.*

A user is allowed to access the resource O if the permission P returned by ACL is equivalent to or dominates[†] the authorization required for his current actions. In actual implementation, the ACL might be a list attached to the resource. Upon access, the access control system retrieves this list and looks up the user's permission in the list. This is matched against the action the user is trying to perform to determine if the access is to be granted.

Given that ACL is a list that is optimised for searching, a direct simulation will have the same semantics but will not enjoy the same optimization. Instead, we introduce a rule as follows:

```
auth(U, O, P) :-  /* User, Resource and Permission */
  auth_acl(O, U, P).
```

The term `auth_acl` is introduced so we can store the authorization list as a constraint fact with Resource O, User U and Permission P as the first, second and last term of the fact respectively. This is to take advantage of the indexing of facts and rules that is normally done on CLP systems. The abstraction is also allows us to introduce more authorization constraints that applies across the access control system.

We can include terms in the ACL in CBAC using constraint facts as follows:

$$\forall ACL(O,U,P), \texttt{assert(auth\_acl(O, U, P))}.$$

The execution sequence of the CBAC program proceeds as follows in resolving the read authorization for User `judge_dred` on Resource `pilot_table`:

1 `auth(judge_dred, pilot_table, read)` resolves to

2 `auth_acl(pilot_table, judge_dred, read)`.

This term is looked up in both the rules as well as the facts database and the system returns *yes* if there is a solution to this binding and *no* otherwise.

[†] In the case of a hirerachy of permissions

3.2. A CBAC simulation of RBAC

Role-Based Access Control works by associating permissions to roles and roles to users. This design models the security policies of organizations with the assumptions the roles sufficiently determines the access authorization of the users. RBAC is recently adopted by several well known databases like Oracle, Sybase and Informix. Due to the varying definitions of RBAC, we took the definition off [Sandhu, 1996].

Definition 3 *The $RBAC_0$ model consists of the following components:*

- *U, R, P, and S (users, roles, permissions and sessions respectively)*
- *$PA \subseteq P \times R$, a many-to-many permission to role assignment relation,*
- *$UA \subseteq U \times R$, a many-to-many user to role assignment relation*
- *$user : S \rightarrow U$, a function mapping each session s_i to the single $user(s_i)$ (constant for the session's lifetime), and*
- *$roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r|(user(s_i), r) \in UA\}$ (user's role may change with session) and session s_i has the permissions $\cup_{r \in roles(s_i)}\{p|(p, r) \in PA\}$.*

For this simulation, we define the following predicates:

- `ua(U, R)`, UA binding as defined above,
- `pa(R, O, P)`, binding between roles, resource and permission,
- `user(S, U)`, mapping from session S to user U. Fails if user has no current session.
- `roles(S, R)`, roles R that are active for session S.

The authorization function `auth` is defined as:

```
auth(U, O, P) :- auth_rbac(U, O, P).
auth_rbac(U, O, P) :-
  user(S, U), roles(S, R), /* R = active roles of user U */
  pa(R, O, P).             /* Get P for O,R */
```

Once again, we abstract `auth_rbac` from `auth` to enable introduction of further constraints at that level. We also assume that role-assignment is consulted prior to the role activation. In other words, the user is not allowed to activate an unauthorized role so there is no need to do a check after retrieving the activated role.

Given the same indexing method for facts described in section 3.1, the worst case analysis is

$$O(\Sigma_R|\texttt{pa(R, O, P)}|).$$

3.3. Beyond the Simulations

Consider organization CBAC Inc again. The leave clerk should be authorized to access the leave table of all employees in order to accomplish his job. The above is perfectly captured in RBAC model, "Any user with the role of a leave clerk should be allowed to modify the leave table". However, we may want to exert a little more control over the access to the leave table. We might want to say that this authorization to modify the leave table should be given only when the employee has taken leave. This workflow is not captured in RBAC. On the other hand, it is not difficult to see how CBAC can naturally model this workflow step by checking for an assertion of some employee taking leave before allowing the leave clerk through.

```
auth(U, O, P) :-
   in_dbtable(O, leave),       /* O is from leave table */
   O = leave(Employee, ...),   /* break up leave tuple */
   leave_applied(Employee), /* Employee applied for leave */
   auth_rbac(U, O, P),       /* RBAC authorization */
```

Next, we might chose to employ different kinds of access control mechanism for different kinds of tables. CBAC captures them within the same syntax. E.g. we may want to have an rbac-based control from some tables but acl based control for others. The above can be handled with the disjunction of constraints.

```
auth(U, O, P) :-
   acl_controlled(O),
   auth_acl(O, U, P).

auth(U, O, P) :-
   rbac_controlled(O),
   auth_rbac(O, U, P).
```

The above examples are aimed at demonstrating how expressive CBAC can be. In fact, CBAC/CLP provides the full expressive power of the underlying CLP language which is Turing-Complete.

3.4. Grant and Revoke Semantics

The GRANT/REVOKE semantics is a heavily debated issue in implementing any access control mechanism. The question usually revolves around what happens to the access rights of a user whose right is granted by another user when the latter's rights are revoked. In a nutshell, any of the following scenarios may be the answer, depending on the system.

1 *Revoke the rights of all the users down the pipeline.* This gives rise to a cascading REVOKE where a right granted may be revoked if the grantor's rights are removed. Removing the access rights of a user implicitly means

removal of access rights granted by this user. It is easy to see how this model gives rise to better control over the access rights.

2 *Revoke the rights of only the users specified, leaving the implicated users alone.* Here, we honour any rights that has been granted and requires explicit removal of any rights. This model is more flexible than the former but requires more work in tracking down the rights that needs removal. This is the approach taken by today's implementation of SQL.

In [Rosenthal and Sciore, 2000], Rosenthal et. al. proposed to extend the SQL grant/revoke model to allow grantors to impose limitations on use of granted rights as well as a flexible revocation scheme. The approach falls between the two scenarios described above. It gives the adminstrators control over the revocation of access priviledges via revoke with limitation predicates, as well as extends the grantor control via predicated grants. This is not unlike the use of constraint rules in describing authorization except that CBAC allows the description of more complex relationship beyond simple predicates.

Use of constraint rules in CBAC should reduce the need to delegate GRANT priviledges to users. Where necessarily, the adminstrators can in fact provide rules that asserts authorisation constraints. Such rules can be as straight forward as an SQL GRANT or support cascading revocation as described in the first senario, or even a predicated GRANT as proposed in Rosenthal et. al. Revocation of user rights can be done through similiar interfaces provided by the adminstrator.

The following example shows how the predicated grant and flexible revocation can be acheived in CBAC.

```
auth(U, O, P) :-
  granted(U, O, P, Grantor),
  not revoked(U, O, P, Grantor),
  auth(Grantor, O, P).
```

Authorisation is given if granted permission is not revoked and grantor still has rights to the resource. The grantors can be allowed free play in asserting their own granted and revoked rules thereby acheiving predicated grants and flexible revocation. Grants and Revokes by different grantors are independent of each other so a grantor cannot revoke the rights granted by another.

4. CBAC: IMPLEMENTATION ISSUES

Efficiency and effectiveness are the 2 main considerations surrounding the design of access control mechanism. We have explore how CBAC can help management of permissions as well as demonstrated the expressiveness of CBAC in the previous sections. The concern about efficiency will be addressed in the sections that follows.

First, the bad news. CBAC allows access if there exists an assignment of variables such that `auth(...)` is realized. The general problem is thus to find an assignment to a problem that is subjected to some constraints, in our case, access control constraints. This is a constraint satisfaction problem (CSP) that is NP-Complete in general [Tsang, 1993]. In other words, a huge system of access control constraints might run into an search that is potentially very expensive. An access control system running on exponential time[‡] will be unusable since it will take too long to respond with the appropriate access authorization.

4.1. Efficiency Considerations

While the general CSP may be NP-Complete, our access control check need not be. In reality, the access control requires only very small constraint programs without deep searches. The typical use of CBAC would be to compute fairly straight forward access authorizations which can be done very efficiently. As we have demonstrated in the section 3, CBAC indeed works efficiently for simple authorization. Complex authorizations would have to come at a cost anyway. Only very few special authorization would need more computation.

The consistency checking mechanism in most constraint solvers are important to ensuring the performance when applying constraints to access control. Constradictions, defined as constraints that will yield no solutions, can be identified by the consistency checking algorithm before further search is invoked. The same consistency checking algorithm will help prune the search space of wrong solutions. Automatic constraint solvers have been known to perform better with a more streamlined set of rules than a larger set of more explicit rules. The former translates to a smaller set of access control rules that facilitates checking and encourages a more unified set of access control policies than patching.

We do not attempt to conclude that CBAC would be optimal for all complex authorizations but its use of a constraint engine that can generate partial solutions can in fact help even in speeding up the database search.

Partial solutions generated by constraint solvers when it cannot give a definite answer may be useful in limiting SQL searches. In a typical system, normal users may be allowed to access only their own record. This check is usually done through a frontend that accesses the database on the user's behalf. A smarter application programmer would realise that in this senario, the `username` can be used to limit the SQL search. For example, if user 'NormalUser' tries to access the system to retrieve his payroll statement, the SQL query formulated by the application can be:

```
select * from payroll where employee='NormalUser';
```

In CBAC, the access control will be written as:

[‡]Until we can prove that $P = NP$

```
auth(U, O, P) :- O = payroll(U, ...).
```

Since `payroll(U, ...)` is a non-entailing but linear constraint, the constraints system will return it as the partial solution. This partial solution can be passed on the database backend as an additional condition yielding the same effect as the smart programmer. In fact, we can do this for any number of linear constraints.

The advantage of CBAC stands out when there are many such applications.

4.2. Implementation Model

This section discusses a priliminary implementation model for CBAC on existing databases. The model is designed so that the CBAC layer can be developed independently from the database server. This philosophy will allow us to use existing databases as well as existing constraint solvers. The assumptions for the model area as follows:

- The database backend need not have constraint solving capabilities
- All connections to the database has to pass through the CBAC layer
- Existing access control system may continue to be in place

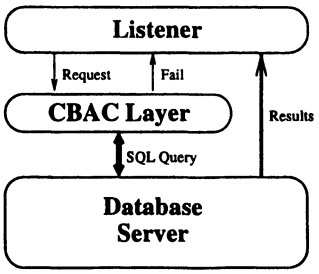

Figure 1. Incorporating CBAC into existing databases

Our proposed model seperates the system into 3 layers as shown in figure 1.

- **Listener.** The listener listens for incoming connection and does the authentication as required by the database server.
- **CBAC layer.** The CBAC layer will store all the access control constraints and apply them on incoming queries before passing the queries to the database.
- **Database Server.** This is the existing database server.

In this model, all queries will pass thru the CBAC layer. Queries may be modified by the CBAC layer to the security specifications before being passed to the database backend. There are also mechanisms that will allow the CBAC layer to "check" with the database backend so that proper access control status can be determined.

Results from the database server is sent directly to the frontend.

The CBAC layer works as follows:

- Any queries that can be cleared directly is passed unchanged to the database backend.
- Queries that fail the security specifications are dropped and CBAC will return an "Access Violation" to the frontend.
- The use of constraint based specification allows for queries that can only be cleared pending some conditions. If these conditions can be answered by the database backend, e.g. linear constraints, the partial solution is added as extra conditions to the original query and the modified query is passed to the backend as cleared.
- Otherwise, the CBAC layer may need to formulate its own queries to the database until any of the above conditions prevails.

The CBAC layer will be compatible with whatever access control strategies the existing database server employs since the SQL queries still need to pass through the existing access control mechanism of the database. In fact, if the existing database server uses RBAC, the CBAC layer may pass the RBAC checks to the database instead.

5. FUTURE WORK

This paper presents a fresh approach to specifying access control directed at database systems. Several issues are yet to be smoothened out.

- *Further implications of applying CBAC to databases.* Most access control mechanism build on top of SQL which is familiar to most database adminstrator but CBAC introduces a rule-based language which may not be familiar to most users.
- *Further implications of the GRANT/REVOKE mechanism.* While we paint a rosy picture about the GRANT/REVOKE mechanism and are comfortable about that CBAC can support predicated grants and conditional revocation, the impact of loose use of GRANT is yet to be known. One of the problems that are not addressed is forming of infinite loops through the grant semantics – the grantor grants himself. The flexible

grant/revoke mechanism discussed in section 3.4 introduced an additional level of search that will have an impact on the efficiency of the entire system if loosely used.

- *Customising the constraint solver for CBAC.* Most constraint solvers are built to be general engines and will remain as such. Some work can be done to customise constraint solvers to speed up the evaluation of constraints that are frequently used by CBAC.

- *CBAC dependency on the database when unresolved authorization constraints are non-linear is not clear.* We are certain this can happen due when complex constraints are involved but are not sure the communication costs for the resolution.

- *A tighter integration with existing databases.* The layered approach will only serve as a demonstration of how the CBAC engine and the database server can be seperately developed. A tighter integration is required to reduce the communication cost between the two systems.

- *Use of validation libraries on CBAC rules.* The expressiveness of CBAC allows users to write authorization rules without introducing another frontend check. This permits us to compute a closure of the access control of the system without having to worry about any further access mechanism at work outside of CBAC. There are validation tools that work with constraint solvers and they may be applied to validate the access control constraints.

6. CONCLUSION

In this paper, we have presented CBAC, a radically different approach to database access control that is still work-in-progress. This constraints based approach allows the users the full flexibility of the constraints language.

Specifying access control policies using a constraints based language has many advantages. Most of the time, access control policies are specified in rules that can be translated directly into constraint rules. The expressiveness of the constraints language removes the reliance on access proxies. This greatly simplifies the job of access control analysis since all information is available at a single point instead of being spread across several programs and possibly different languages. The use of constraint rules also help the analysis.

We showed how CBAC can simulate both ACL and RBAC and went further to demonstrate CBAC with a workflow example. We also discussed the grant/revoke semantics that can be acheived within CBAC. The database adminstrator now can specify exactly how grant/revoke is used with the same language that authorizes the user-rights of his database.

CBAC casts the access control graph into a CSP problem which is NP-Complete. However, as demonstrated by the examples, the use of CBAC would typically be restricted to fairly straight forward computations that will not result in deep searches. Partial solutions generated by the constraint solvers can be added to the final SQL query to limit the database search.

An implementation model that keeps the development of CBAC and the database server seperate is also presented.

In summary, CBAC presents a rich syntax for describing access control policies. It is important to note that CBAC still needs a lot of work to deployment as we have highlighted in section 5. Despite the followups required, we believe that CBAC deserves attention, not only from the database developers, but also from developers of massive systems where access control policies are important.

References

[Bertino et al., 1998] Bertino, E., Bettini, C., Ferrari, E., and Samarati, P. (1998). An access control model supporting periodicity constraints and temporal reasoning. In *ACM Transactions on Database Systems*, volume 23, pages 231–285.

[Bertino et al., 1993] Bertino, E., Samarati, P., and Jajodia, S. (1993). Authorizations in relational database management systems. In *1st ACM Conference on Computer and Communications Security*, pages 130–139.

[Castano et al., 1994] Castano, S., Fugini, M., Martella, G., and Samarati, P. (1994). *Database Security*. Addison Wesley.

[Jaffar and Lassez, 1987] Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Principles of Programming Languages*.

[Marriott and Stuckey, 1998] Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints*. The MIT Press.

[Rosenthal and Sciore, 2000] Rosenthal, A. and Sciore, E. (2000). Extending sql grant and revoke operations to limit and reactive privileges. In *IFIP Working Conference on Database Security*.

[Sandhu, 1996] Sandhu, R. S. (1996). Role-based access control. Technical report, Laboratory for Information Security Technology, Geore Mason University.

[Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

[Sandhu and Samarati,] Sandhu, R. S. and Samarati, P. Access control: Principles and practice. www.isse.gmu.edu/faculty/sandhu.

[Tsang, 1993] Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.