Discretionary Access Control Models

Prof.Dr. Ferucio Laurenţiu Ţiplea

Department of Computer Science
Alexandru Ioan Cuza University of Iaşi
Iaşi, Romania
E-mail: fltiplea@mail.dntis.ro

Outline

Discretionary Access Control

Basic features:

- DAC models enforce access control on the basis of the identity of requesters
- DAC models are called “discretionary” as users can be given the ability of passing on their privileges to other users
- DAC mechanisms usually include a concept of object ownership

DAC models:

- Take-grant model
- Access-matrix model
- Schematic model

The Take-grant Model

The take-grant model has been proposed in:

A.K. Jones, R.J. Lipton, L. Snyder: A Linear Time Algorithm for Deciding Security, Proc. of 17th Annual Symp. on Found. of Comp. Sci., 1976.

Basic features:

- Take-grant systems are state-transition systems
- Subjects are not objects
- States are directed graphs whose nodes are subjects and objects, and whose arcs are labeled by sets of rights
- There are two special rights: $take$ (t) and $grant$ (g):
  - if x has the right take for y, then x can "borrow" from y all his abilities (rights)
  - if x has the right grant for y, then x can "lend" to y all his abilities (rights)

Take-grant States

Example 1

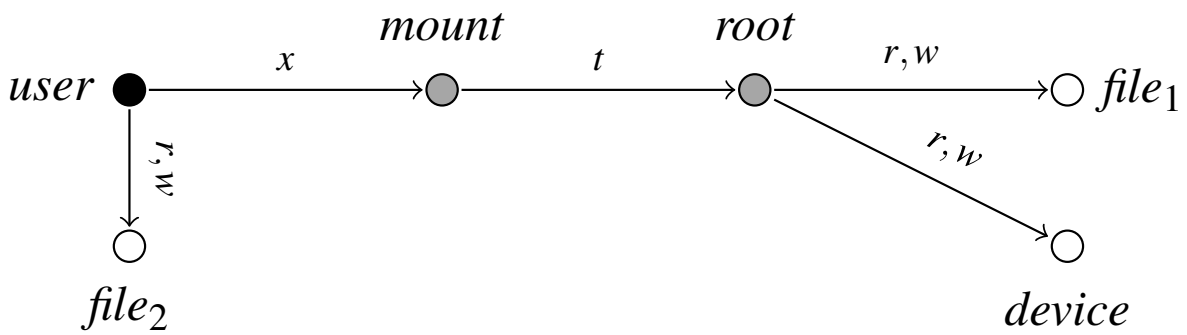


Figure: Take-grant state

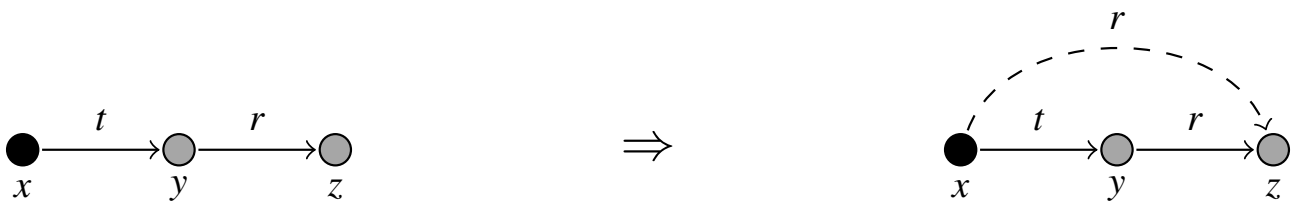
Meaning:

- Dark circles stand for subjects
- Open circles stand for objects
- Gray circles denote either a subject or an object
- An arc from x to y labeled α says that x has rights $r \in \alpha$ for y

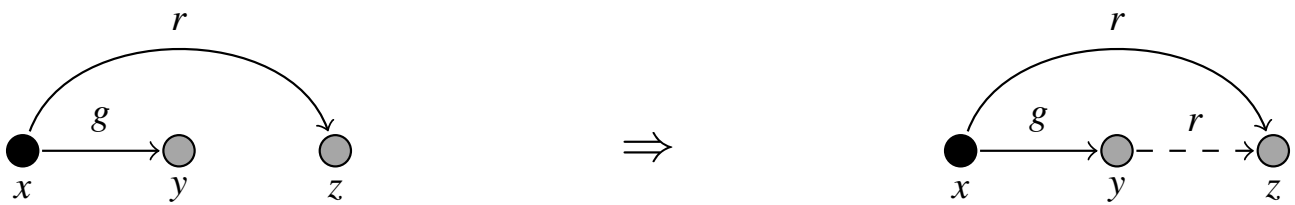
Transitions in Take-grant Systems

The transition relation in take-grant systems is guided by four rules:

- **Take**: *x take r for z from y*



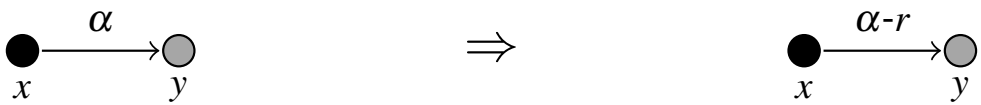
- **Grant**: *x grant r for z to y*



Transitions in Take-grant Systems

- **Create**: *x create α for new subject/object y*

- **Remove**: *x remove r for y*



Safeness in Take-grant Systems

Define the predicate $can.share(r,x,p,G)$

$$can.share(r,x,p,G) \Leftrightarrow \exists G' : G \overset{*}{\Rightarrow} G' \wedge r \in_{G'} (p,x)$$

where:

- G and G' are take-grant states
- r is a right
- x and p are nodes in G

Definition 2

Let G be a take-grant state, r a right, and x and p nodes in G. G is called safe for r and x w.r.t. p if $can.share(r,x,p,G)$ does not hold.

Connected Nodes

Definition 3

Let G be a take-grant state and x and y two nodes of G.

1. x and y are directly connected if there is an arc between them.
2. x and y are directly tg-connected if there is an arc between them with a label containing t or g.
3. A path (tg-path) is a sequence $x_0, x_1, \ldots, x_n$ of nodes such that x_i and x_{i+1} are directly connected (tg-connected), for any $0 \leq i < n$.
4. x and y are connected (tg-connected) if there is a path (tg-path) between them.

Remark 1

We emphasize that if x and y are not connected in G then they cannot become connected in any G' obtained by rewriting G. This is because no rule adds arcs between unconnected nodes.

Initial/Terminal Span, Island, Bridge

Let G be a take-grant state. With each tg-path associate one or more words over the alphabet $\{\overrightarrow{t}, \overleftarrow{t}, \overrightarrow{g}, \overleftarrow{g}\}$ in an obvious way.

Definition 4

Let G be a take-grant state.

1. An island of G is any subject-only tg-connected subgraph of G.
2. A node x initially spans to y if x is a subject and there is a tg-path between x and y with an associated word in $(\overrightarrow{t})^*\overrightarrow{g} + \lambda$.
3. A node x terminally spans to y if x is a subject and there is a tg-path between x and y with an associated word in $(\overrightarrow{t})^*$.
4. A bridge is a tg-path with endpoints both subjects and with an associated word in $(\overrightarrow{t})^* + (\overleftarrow{t})^* + (\overrightarrow{t})^*\overrightarrow{g}(\overleftarrow{t})^* + (\overrightarrow{t})^*\overleftarrow{g}(\overleftarrow{t})^*$.

Deciding Safeness

Theorem 5

Let G be a take-grant state, r a right, and x and p nodes in G. Then, $can.share(r,x,p,G)$ is true if and only if $r \in (p,x)$ or there exists a node s, two subjects p' and s', and islands $I_1,\ldots,I_n$ such that:

1. $r \in_G (s,x)$;
2. $p' = p$ or p' initially spans to p;
3. $s' = s$ or s' terminally spans to s;
4. p' is in I_1, s' is in I_n, and there is a bridge from I_j to I_{j+1}, for all $1 \leq j < n$.

Corollary 6

There is an algorithm for testing $can.share$ that operates in linear time in the size of the initial state (graph).

The Access-matrix Model

The most general DAC model has been proposed in 1976:

> M.A. Harrison, W.L. Ruzzo, J.D. Ullman. *Protection in Operating Systems*, Communications of the ACM, vol.19, no. 8, 1976, 461–471.

It is called the access control matrix model or the access-matrix model or the HRU model.

Basic features:

- It is a state-transition system
- States are matrices where each row corresponds to a subject, each column corresponds to an object, and a cell specifies the rights a subject has over an object
- Transitions between states are performed by commands
- Subjects are objects too

States in the Access-matrix Model

In what follows, R denotes a non-empty finite set of rights.

Definition 7

A state over R is a triple $Q = (S, O, A)$, where S and O are non-empty finite sets of *subjects* and *objects*, respectively, and A is an $|S| \times |O|$-matrix whose elements are subsets of R.

Example 8

Let $S = \{process_1, process_2\}$, $O = \{process_1, process_2, file\}$, and A given below:

$$\begin{array}{c} \\ process_1 \\ process_2 \end{array} \begin{array}{c} \begin{array}{ccc} process_1 & process_2 & file \end{array} \\ \begin{pmatrix} \emptyset & r & r,w \\ r,x & \emptyset & r \end{pmatrix} \end{array}$$

The triple (S, O, A) is a state over $R = \{r, w, x\}$.

Primitive Operations

$\mathscr{V}_{sub}$ = set of variables of type subject, $\mathscr{V}_{ob}$ = set of variables of type object

Definition 9

A primitive operation over R is a construct of the one of the following types:

1. *enter* r *into* (X_s, X_o)
2. *delete* r *from* (X_s, X_o)
3. *create subject* X_s
4. *create object* X_o
5. *destroy subject* X_s
6. *destroy object* X_o

where $r \in R$, $X_s \in \mathscr{V}_{sub}$, and $X_o \in \mathscr{V}_{ob}$.

Commands

Definition 10

A command over R is a construct of the form:

$$\begin{array}{l} command\ \alpha(X_1,\ldots,X_k) \\ \quad op_1,\ldots,op_n \\ end \end{array}$$

$$\begin{array}{l} command\ \alpha(X_1,\ldots,X_k) \\ \quad if\ \ r_1\ in\ (X_{s_1},X_{o_1})\ and \\ \quad\quad \ldots \\ \quad\quad r_m\ in\ (X_{s_m},X_{o_m}) \\ \quad then\ op_1,\ldots,op_n \\ end \end{array}$$

where $m,n \geq 1$, $r_1,\ldots,r_m \in R$, $X_1,\ldots,X_k \in \mathscr{V}_{sub} \cup \mathscr{V}_{ob}$, $1 \leq s_1,\ldots,s_m,o_1,\ldots,o_m \leq k$, $X_{s_i} \in \mathscr{V}_{sub}$ and $X_{o_i} \in \mathscr{V}_{ob}$ for all $1 \leq i \leq m$, and $op_1,\ldots,op_n$ are operations over R whose variables are among $X_1,\ldots,X_k$.

Definition 11

A protection system over R is a finite set $\mathscr{C}$ of commands over R.

Examples of Commands

Example 12

command $CREATE(process, file)$
    create object $file$
    enter own into $(process, file)$
end

Example 13

command $CONFER\_READ(owner, friend, file)$
    if $own\ in\ (owner, file)$
    then
        enter $r\ into\ (friend, file)$
end

Examples of Commands

Example 14

command $REMOVE\_READ(owner, exfriend, file)$
    if *own in* $(owner, file)$ *and*
        r *in* $(exfriend, file)$
    then
        delete r *from* $(exfriend, file)$
end

Substitution

A substitution assigns values to variables according to their types:

- subjects to subject-type variables, and
- objects to object-type variables.

Substitutions can homomorphically be applied to primitive operations and commands.

Example 15

Let S be a set of subjects and O be a set of objects. If $\sigma(X) = s \in S$ and $\sigma(X') = o \in O$, then

$$\sigma(enter\ r\ into\ (X, X')) = enter\ r\ into\ (s, o)$$

Transition Relation

Given an operation op and a substitution σ, define the binary relation $\Rightarrow_{\sigma(op)}$ on states by

$$(S,O,A)\Rightarrow_{\sigma(op)}(S',O',A')$$

if and only if one of the following properties holds:

1. if $op = enter\ r\ into\ (X,Y)$, then $\sigma(X)\in S$, $\sigma(Y)\in O$, $S'=S$, $O'=O$, and

$$A'(s,o)=\begin{cases}A(s,o)\cup\{r\}, & \text{if } (\sigma(X),\sigma(Y))=(s,o)\\ A(\sigma(X),\sigma(Y)), & \text{otherwise}\end{cases}$$

2. if $op = delete\ r\ from\ (X,Y)$, then $\sigma(X)\in S$, $\sigma(Y)\in O$, $S'=S$, $O'=O$, and

$$A'(s,o)=\begin{cases}A(s,o)-\{r\}, & \text{if } (\sigma(X),\sigma(Y))=(s,o)\\ A(\sigma(X),\sigma(Y)), & \text{otherwise}\end{cases}$$

Transition Relation

3. if $op = create\ subject\ X$, then $\sigma(X) \notin O$, $S' = S \cup \{\sigma(X)\}$, $O' = O \cup \{\sigma(X)\}$, and
$$A'(s,o) = \begin{cases} A(s,o), & \text{if } (s,o) \in S \times O \\ \emptyset, & \text{otherwise} \end{cases}$$
4. if $op = create\ object\ Y$, then $\sigma(Y) \notin O$, $S' = S$, $O' = O \cup \{\sigma(Y)\}$, and
$$A'(s,o) = \begin{cases} A(s,o), & \text{if } (s,o) \in S \times O \\ \emptyset, & \text{otherwise} \end{cases}$$
5. if $op = destroy\ subject\ X$, then $\sigma(X) \in S$, $S' = S - \{\sigma(X)\}$, $O' = O - \{\sigma(X)\}$, and $A'(s,o) = A(s,o)$, for all $(s,o) \in S' \times O'$;
6. if $op = destroy\ object\ Y$, then $\sigma(Y) \in O - S$, $S' = S$, $O' = O - \{\sigma(X)\}$, and $A'(s,o) = A(s,o)$, for all $(s,o) \in S' \times O'$.

Define now

$$(S,O,A) \Rightarrow_{op} (S',O',A') \ \Leftrightarrow\ \exists \sigma :\ (S,O,A) \Rightarrow_{\sigma(op)} (S',O',A')$$

Transition Relation

Given a command α and a substitution σ, define the binary relation $\Rightarrow_{\sigma(\alpha)}$ on states by

$$(S,O,A)\Rightarrow_{\sigma(\alpha)}(S',O',A')$$

if and only if one of the following properties holds:

1. if the test of $\sigma(\alpha)$ is not satisfied at (S,O,A), then $(S',O',A')=(S,O,A)$;
2. if the test of $\sigma(\alpha)$ is satisfied at (S,O,A), then there exist $Q_0,Q_1,\ldots,Q_n$ such that

$$(S,O,A)=Q_0\Rightarrow_{\sigma(op_1)}Q_1\Rightarrow_{\sigma(op_2)}\cdots\Rightarrow_{\sigma(op_n)}Q_n=(S',O',A')$$

   where $op_1,\ldots,op_n$ is the body of α.

Define now

$$(S,O,A)\Rightarrow_{\alpha}(S',O',A') \;\Leftrightarrow\; \exists\sigma:\ (S,O,A)\Rightarrow_{\sigma(\alpha)}(S',O',A')$$

and

$$(S,O,A)\Rightarrow(S',O',A') \;\Leftrightarrow\; \exists\alpha:\ (S,O,A)\Rightarrow_{\alpha}(S',O',A')$$

Safety

Definition 16

Let $\mathscr{C}$ be a protection system over R, Q a state of $\mathscr{C}$, $r \in R$, and α a command of $\mathscr{C}$. We say that α leaks r from Q if there exists a substitution σ such that:

1. the test of $\sigma(\alpha)$ is satisfied at Q;
2. there exist $Q_0, Q_1, \ldots, Q_i$ such that:
   - $Q_0 = (S_0, O_0, A_0) \Rightarrow_{\sigma(op_1)} Q_1 = (S_1, O_1, A_1) \Rightarrow_{\sigma(op_2)} \cdots \Rightarrow_{\sigma(op_i)} Q_i = (S_i, O_i, A_i)$;
   - $r \in A_i(s,o) - A_{i-1}(s,o)$ for some s and o,

   where $op_1, \ldots, op_i, \ldots, op_n$ is the body of α and $1 \leq i \leq n$.

Definition 17

Let $\mathscr{C}$ be a protection system over R, Q a state of $\mathscr{C}$, and $r \in R$. We say that $\mathscr{C}$ leaks r from Q if there exists a command of $\mathscr{C}$ that leaks r from Q.

Safety

Definition 18

Let $\mathscr{C}$ be a protection system over R, Q a state of $\mathscr{C}$, and $r \in R$. We say that Q is unsafe for r if there exists a reachable state Q' from Q such that $\mathscr{C}$ leaks r from Q'.

We say that Q is safe for r if it is not unsafe for r.

The safety problem for protection systems is the problem to decide, given a protection system over some set R of rights, a state Q of $\mathscr{C}$, and a right $r \in R$, whether Q is safe for r.

Remark 2

We emphasize that "leaks" are not necessarily "bad". Any interesting protection system has commands able to leak some rights. However, these leaks should not occur at unauthorized states.

Deciding safety

Theorem 19

The safety problem for bi-conditional (i.e., at most two conditions) monotonic (i.e., without delete and destroy operations) protection systems is undecidable.

Theorem 20

The safety problem for mono-conditional protection systems without destroy-operations is decidable.

Most practical systems require multi-conditional commands !

Implementation

Access control matrix implementations do not scale well: a bank with 50,000 staff and 300 applications would have an access control matrix of 15 million entries !

We need compact ways of storing and managing access control matrices.

Two main ways of doing this are:

1. use groups (roles) to manage the privileges (rights) of large sets of users simultaneously (role-based access control - RBAC);
2. store the matrix either by columns (access control lists - ACL) or rows (capability lists).

Access Control Lists

Definition 21

An access control list (ACL) is a column of the access control matrix (therefore, associated to an object - the ACL associated to o is denoted ACL_o, and it is stored along with o).

Advantages and disadvantages of ACLs:

- suited to environments where users manage their own file security;
- less suited where the user population is large and constantly changing;
- less suited where users want to be able to delegate their authority to run a particular program to another user for some set period of time;

Access Control Lists

Advantages and disadvantages of ACLs (continued):

- simple to implement;
- security checking at runtime is difficult (usually, the operating system knows which user is running a particular program, rather than which files it has been authorized to access);
- tedious to find all the files to which a user has access;
- tedious to run system wide checks, such as verifying that no files have been left world-writable by users whose access was revoked.

Access Control Lists in Unix

In Unix:

- every file or folder has associated access permissions. There are three types of permissions:
  - read access
  - write access
  - execute access
- permissions are defined for three types of users:
  - the owner of the file
  - the group that the owner belongs to
  - anyone else (world)

  Each permission type has exactly two values, allowed or denied, specified by a bit.

Access Control Lists in Unix

Example 22

ACL for a file:

```
-rw-r----- Alice Accounts
```

The first bit specifies that the ACL is for a file, the next three bits give the access rights for the owner, the next three bits for the group, and the last three bits for anyone else. It follows then the owner name and the group name.

Example 23

ACL for a folder:

```
drwxrwxrwx Alice Accounts
```

The first bit specifies that the ACL is for a folder (directory); the next bits have the same meaning as above.

Access Control Lists in Unix

How is associated an ACL to a program in Unix?

Unix does not provide any direct method for doing this. However, there are two attributes, `suid` (set user id) and `sgid` (set group id), which help for this:

- the owner of the program mark the program as `suid` (the bit `x` in owner ACL is set to `s` meaning both `x` and `suid`, or to `S` meaning only `suid`);
- then, the program is placed in some folder where some user Alice has access;
- Alice can run the program with the privilege of its owner.

(things are similar for `sgid`).

This method leads to serious security holes.

Access Control Lists in Windows NT

In Windows NT, the access control is richer than Unix, but not fundamentally different:

- There are six types of permissions:
  - read access
  - write access
  - execute access
  - delete
  - change permissions (i.e., modify the ACL)
  - take ownership (make current account the new owner)
- permissions are defined for users and groups. Each permission type has three values, Access denied, Access allowed, and System audit.

ACLs are associated to items (i.e., files or directors), and each ACL is a list of entries of the form

$$\cdots \quad \texttt{(user/group,permissions)} \quad \cdots$$

Capability Lists

Definition 24

An capability list (C-list) is a row of the access control matrix (therefore, associated to a subject - the C-list associated to s is denoted C_s, and it is stored along with s).

In practice, it is more convenient to store a C-list C_s as a list of pairs (o, r), where o is an object and r is a right (permission). Such a pair will be called a capability; then, C_s becomes a list of capabilities. Each capability acts like a ticket for s to access o with permission r. Therefore, capabilities are authentication tags.

This technique is used in EROS (Extremely Reliable Operating System), Hydra operating system (CMU), IBM System/38 and AS/400, Amoeba distributed operating system etc.

Capability Lists

Problems with capabilities:

- how to represent object o in capability (o, r)?
  The use of o's address might not be a good idea if the address changes.
  A solution would be to use random bit strings, hash tables, and translation techniques (naming schemes);
- How to make capabilities unforgeable? There are a number of possibilities:
  - hardware tags: 1-bit tag associated to the capability, showing that the capability can/cannot be changed or copied;
  - protected address space: store capabilities in parts of memory that are not accessible to programs;
  - language-based security: use of a programming language to enforce restrictions on access and modification to capabilities;
  - cryptography: use encryption.

The Schematic Model

The schematic model has been proposed in:

R. S. Sandhu: The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes, Journal of the ACM 35(2), 1988, 404–432.

Why this model ?

To fill the gap between the richness in expressive power of the HRU model and its intractability with respect to the safety question as compared with the limited applicability of the take-grant model but efficient decidability of safety.

How ?

The schematic model provides considerably more structure than HRU.

The Schematic Model: Types and Rights

Types:

- subject types: TS
- object types: TO
- $T = TS \cup TO$
- $\tau(x)$: type of the entity x
- subjects and objects are distinct entities

Rights:

- inert rights: RI (do not affect the protection state)
- control rights: RC (may change the protection state)
- $R = RI \cup RC$
- copy flag c: rc means "r is copyable", while r means "r is not copyable"
- $r : c$ denotes r or rc (rc subsumes r !)

The Schematic Model: Tickets

Tickets:

- A ticket is a pair $(x, r : c)$ (often written as $x/r : c$)
- Abbreviation: $x/r_1 r_2 c$ means $\{x/r_1 c, x/r_2 c\}$
- $\tau(x/r : c)$ is defined as $\tau(x)/r : c$
- for a subject x, $dom(x)$ is a set of tickets that x has

Presence of x/rc in $dom(y)$ subsumes presence of x/r, but not vice versa !

The Schematic Model: States

States:

- A state consists of a (finite) set of typed entities (subjects and objects) together with their domains

Operations that change the current state:

- copy: moves a copy of a ticket from the domain of one subject to the domain of another, leaving the original ticket intact
- create: introduces new subjects and objects in the system

The original formulation of the model included a third operation, demand. However, this is formally redundant:

R. S. Sandhu: The Demand Operation in the Schematic Protection Model, Information Processing Letters 32(4), 1989, 213–219.

The Schematic Model: The Copy Operation

The copy operation is guided by a link predicate $link_i$ and a filter function f_i:

- $link_i(x,y)$ is a disjunction or conjunction of atomic terms:

$$x/r \in dom(x),\ x/r \in dom(y),\ y/r \in dom(x),\ y/r \in dom(y),$$

  where $r \in RC$;
- $f_i : TS \times TS \rightarrow \mathscr{P}(T \times R)$

One filter function is associated to each link predicate !

A ticket $z/r : c$ can be copied from $dom(x)$ to $dom(y)$ iff there exists i such that:

- $z/rc \in dom(x)$;
- $link_i(x,y)$ evaluates to true;
- $\tau(z)/r : c \in f_i(\tau(x), \tau(y))$

The Schematic Model: The Create Operation

The create operation is guided by two predicates: cc (can create) and cr:

- $cc \subseteq TS \times T$

  $(a,b) \in cc$ a subject of type a can create an entity of type b

- cr handles the tickets
  - object creation: $cr(a,b) \subseteq \{b/r : c | r \in RI\}$

    (when a subject x of type a creates on object y of type b, x gets $y/r : c$ iff $b/r : c \in cr(a,b)$)
  - subject creation:
    - $a \neq b$: $cr(a,b) = cr_p(a,b) \cup cr_c(a,b)$

      (as above, x gets $y/r : c$ iff $b/r : c \in cr_p(a,b)$, and y gets $x/r : c$ iff $a/r : c \in cr_c(a,b)$)
    - $a = b$: $cr(a,b) \subseteq \{a/r : c, self/r : c | r : c \in RI\}$

      (as above, where $self/r : c$ denotes tickets for the creator)

The Schematic Model: Examples

Example 25 (Owner-based policy)

Subject u can authorize subject v to access an object z iff u owns z:

- $TS = \{user\}$, $TO = \{file\}$
- $RI = \{r : c, w : c, a : c, x : c\}$ (read, write, append, execute)
- $RC = \emptyset$
- $link(u, v) = true$, for all u and v of type user
- $f(user, user) = \{file/xc\}$
- $cc = \{(user, file)\}$
- $cr(user, file) = \{file/rc, file/wc, file/xc\}$

The Schematic Model: Examples

Example 26 (The basic take-grant model)

- $TS = \{user\}$, $TO = \{file\}$
- $RI = \{x : c\}$ (execute)
- $RC = \{t : c, g : c\}$
- $link(u, v) = true$ iff $v/g \in dom(u)$ or $u/t \in dom(v)$
- $f(user, user) = T \times R$
- $cc = \{(user, file), (user, user)\}$
- $cr(user, file) = \{file/xc\}$
- $cr(user, user) = \{user/tgc, self/tgc\}$

The Schematic Model: Results

The schematic model subsumes several well-known protection models in terms of expressive power and safety analysis, such as:

- the Bell-LaPadula multi-level security model
- take-grant models
- the grammatical protection model

R. S. Sandhu: Expressive Power of the Schematic Protection Model, Journal of Computer Security 1(1), 1992, 59–98.

Concluding Remarks

- DAC policies enforce access control on the basis of the identity of the requester and explicit access rules
- DAC policies ignore the distinction between users and subjects and evaluate all requests submitted by a process (subject) running on behalf of some user against the authorizations of the user
- DAC policies are vulnerable from processes executing malicious programs (such as Trojan Horses) exploiting the authorizations of the user on behalf of whom they are executing
- DAC policies do not enforce any control on the flow of information once this information is acquired by a process

A more precise examination of the access control problem shows the utility of separating users from subjects and controlling the flow of information !