Introduction to
Computer Security

Lecture 3

Take Grant Model (Cont)
HRU
Schematic Protection Model

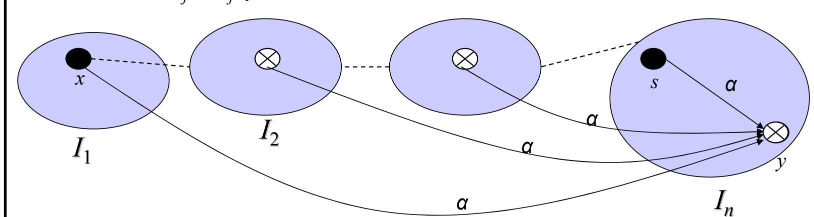
September 16, 2004

Courtesy of Professors
Chris Clifton & Matt Bishop

Theorem: Can_share(α,**x**,**y**,G_0) (for subjects)

- Subject_can_share(α, x, y, G_0) is true iff x and y are subjects and
  - there is an α edge from x to y in G_0

  OR if:
  - ∃ a subject s ∈ G_0 with an s-to-y α edge, and
  - ∃ islands $I_1, \ldots, I_n$ such that $x \in I_1$, **s** ∈ I_n, and there is a bridge from I_j to I_{j+1}

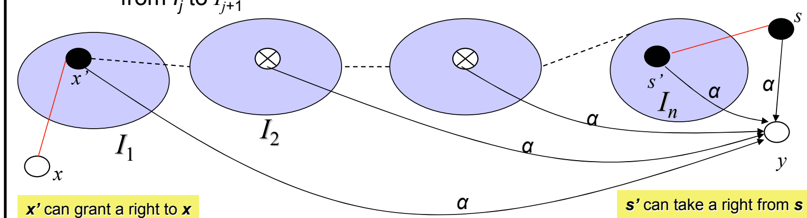


What about objects? Initial, terminal spans

- x *initially spans* to y if x is a subject and there is a *tg*-path associated with word $\{t_{\rightarrow}{}^{*}g_{\rightarrow}\}$ between them
  - x can grant a right to y
- x *terminally spans* to y if x is a subject and there is a *tg*-path associated with word $\{t_{\rightarrow}{}^{*}\}$ between them
  - x can take a right from y

Theorem: Can_share(α,**x**,**y**,G_0)

- Can_share(α,x, y,G_0) iff there is an α edge from x to y in G_0 or if:
  - ∃ a vertex $s \in G_0$ with an s to y α edge,
  - ∃ a subject x' such that $x'=x$ or x' *initially spans* to x,
  - ∃ a subject s' such that $s'=s$ or s' *terminally spans* to s, and
  - ∃ islands $I_1, \ldots, I_n$ such that $x' \in I_1$, $s' \in I_n$, and there is a bridge from I_j to I_{j+1}



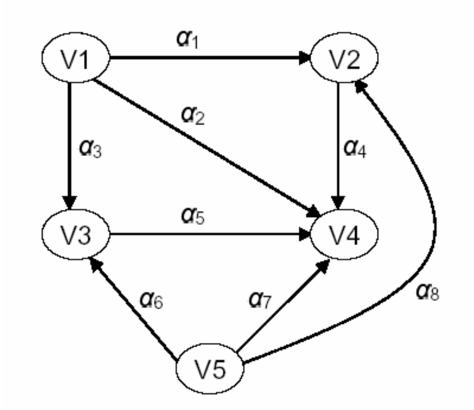
Theorem: Can_share(α,**x**,**y**,G_0)

- Corollary: There is an $O(|V|+|E|)$ algorithm to test can_share: Decidable in linear time!!
- Theorem
  - Let G_0 contain exactly one vertex and no edges,
  - R a set of rights.
  - $G_0 \vdash^* G$ iff G is a finite directed acyclic graph, with edges labeled from R, and at least one subject with no incoming edge.
  - *Only if* part: v is initial subject and $G_0 \vdash^* G$;
    - No rule allows the deletion of a vertex
    - No rule allows an incoming edge to be added to a vertex without any incoming edges. Hence, as v has no incoming edges, it cannot be assigned any

Theorem: Can_share(α,**x**,**y**,G_0)

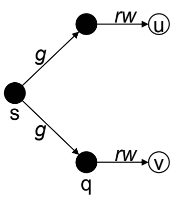
- *If* part : G meets the requirement
  - Assume v is the vertex with no incoming edge and apply rules
  1. Perform "v creates (α ∪ {g} to) new x_i" for all 2<=i <= n, and α is union of all labels on the incoming edges going into x_i in G
  2. For all pairs x, y with x α over y in G, perform "v grants (α to y) to x"
  3. If *β* is the set of rights x has over y in G, perform "v removes (α ∪ {g} - *β*) to y"

Example

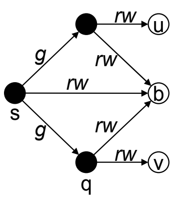


Take-Grant Model: Sharing through a Trusted Entity

- Let *p* and *q* be two processes
- Let *b* be a buffer that they share to communicate
- Let *s* be third party (e.g. operating system) that controls *b*



Witness
- S creates ({r, w}, to new object) b
- S grants ({r, w}, b) to p
- S grants ({r, w}, b) to q



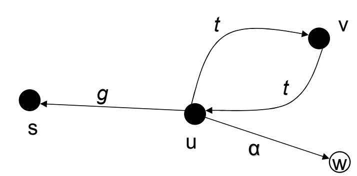
Theft in Take-Grant Model

- Can_steal(α,**x**,**y**,G_0) is true if there is no α edge from **x** to **y** in G_0 and ∃ sequence $G_1, \ldots, G_n$ s. t.:
  - ∃ α edge from **x** to **y** in G_n,
  - ∃ rules $\rho_1, \ldots, \rho_n$ that take $G_{i-1} \vdash \rho_i \; G_i$, and
  - ∀ **v**,**w** ∈ G_i, $1 \le i < n$, if ∃ α edge from **v** to **y** in G_0 then ρ_i is not "**v** grants (α to **y**) to **w**"

  - Disallows owners of α rights to y from transferring those rights
  - Does not disallow them to transfer other rights
  - This models a Trojan horse

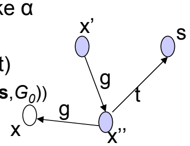
A witness to theft

- u grants (t to v) to s
- s takes (t to u) from v
- s takes (α to w) from u



Theorem: When Theft Possible

- Can_steal(α,**x**,**y**,G_0) iff there is no α edge from **x** to **y** in G_0 and ∃ G_1, …, G_n s. t.:
  - There is no α edge from **x** to **y** in G_0 ,
  - ∃ subject **x’** such that **x’=x** or **x’** *initially spans* to **x**, and
  - ∃ **s** with α edge to **y** in G_0 and can_share(t,**x**,**s**,G_0)
- Proof:
  - ⇒: Assume the three conditions hold
    - x can get t right over s (x is a subject) and then take α right over y from s
    - x’ creates a surrogate to pass α to x (x is an object)
      - X’ initially spans to x (Theorem 3.10 – can_share(t,**x’**,**s**,G_0))



Theorem: When Theft Possible

- ⇐: Assume can_steal is true:
  - No α edge from definition 3.10 in G_0.
  - Can_share(α,**x**,**y**,G_0) from definition 3.10 condition (a): α from **x** to **y** in G_n
  - **s** exists from can_share and earlier theorem
  - Show Can_share(t,**x**,**s**,G_0) holds: **s** can’t grant α (definition), someone else must get α from **s**, show that this can only be accomplished with take rule

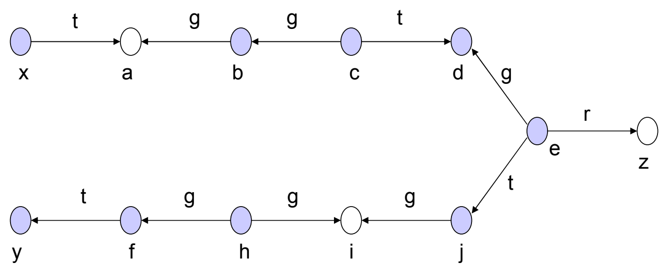
Conspiracy

- Theft indicates cooperation: which subjects are actors in a transfer of rights, and which are not?
- Next question is
  - How many subjects are needed to enable *Can_share(α,**x**,**y**,G_0)?*
- Note that a vertex y
  - Can take rights from any vertex to which it terminally spans
  - Can pass rights to any vertex to which it initially spans
- Access set A(**y**) with focus **y** (y is subject) is union of
  - set of vertices **y,**
  - vertices to which **y** initially spans, and
  - vertices to which **y** terminally spans

Conspiracy

- Deletion set δ(y,y'): All z ∈ A(y) ∩ A(y') for which
  - **y** initially spans to **z** and **y'** terminally spans to **z**
  - **y** terminally spans to **z** and **y'** initially spans to z
  - **z**=**y** & **z**=**y'**
- Conspiracy graph H of G_0:
  - Represents the paths along which subjects can transfer rights
  - For each subject in G_0, there is a corresponding vertex h(x) in H
  - if δ(**y**,**y'**) not empty, edge from h(**y**) to h(**y'**)

Example



Theorems

- I(p) =
  - contains the vertex h(p) and the se t of all vertices h(p') such that p' initially spans to p
- T(q) =
  - contains the vertex h(q) and the se t of all vertices h(q') such that q' terminally spans to q
- Theorem 3-13:
  - Can_share(α,**x**,**y**,G_0) iff there is a path from som h(p) in I(x) to some h(q) in T(y)
- Theorem 3-14:
  - Let L be the number of vertices on a shortest path between h(p) and h(q) (as in theorem 3-13), then L conspirators are necessary and sufficient to produce a witness to Can_share(α,**x**,**y**,G_0)

Back to HRU: Fundamental questions

- How can we determine that a system is secure?
  - Need to define what we mean by a system being "secure"
- Is there a generic algorithm that allows us to determine whether a computer system is secure?

Turing Machine & halting problem

- The **halting problem**:
  - *Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts (the alternative is that it runs forever without halting).*
- Reduce TM to Safety problem
  - If Safety problem is decidable then it implies that TM halts (for all inputs) – showing that the halting problem is decidable (contradiction)

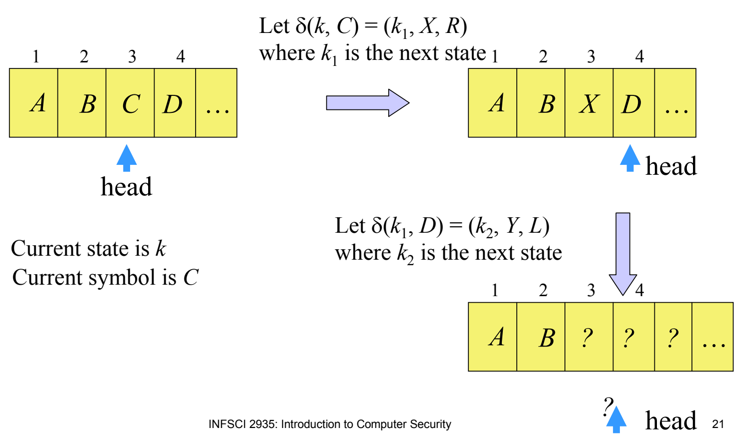
Turing Machine

- TM is an abstract model of computer
  - Alan Turing in 1936
- TM consists of
  - A tape divided into cells; infinite in one direction
  - A set of tape symbols *M*
    - M contains a special blank symbol *b*
  - A set of states *K*
  - A head that can read and write symbols
  - An action table that tells the machine
    - What symbol to write
    - How to move the head ('L' for left and 'R' for right)
    - What is the next state

Turing Machine

- The action table describes the transition function
- Transition function $\delta(k, m) = (k', m', L)$:
  - in state *k*, symbol *m* on tape location is replaced by symbol *m′*,
  - head moves to left one square, and TM enters state *k′*
- Halting state is q_f
  - TM halts when it enters this state

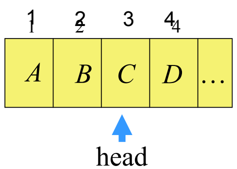
Turing Machine



General Safety Problem

- Theorem: It is undecidable if a given state of a given protection system is safe for a given generic right
- Proof: Reduce TM to safety problem
  - Symbols, States $\Rightarrow$ rights
  - Tape cell $\Rightarrow$ subject
  - Cell s_i has *A* $\Rightarrow$ s_i has *A* rights on itself
  - Cell s_k $\Rightarrow$ s_k has end rights on itself
  - State *p*, head at s_i $\Rightarrow$ s_i has *p* rights on itself
  - Distinguished Right *own*:
    - s_i *owns* s_i*+1* for $1 \le i < k$

Mapping



Current state is k
Current symbol is C

	s_1	s_2	s_3	s_4	
s_1	A	*own*			
s_2		B	*own*		
s_3			C k	*own*	
s_4				D end	

Command Mapping (Left move)

$\delta(k, \mathrm{C}) = (k_1, \mathrm{X}, \mathrm{L})$

command $c_{k,C}(s_i, s_{i-1})$
if *own* **in** $a[s_{i-1}, s_i]$ **and** k **in** $a[s_i, s_i]$ **and** C **in** $a[s_i, s_i]$
then
  delete k **from** $A[s_i,s_i]$;
  delete C **from** $A[s_i,s_i]$;
  enter X **into** $A[s_i,s_i]$;
  enter k_1 **into** $A[s_{i-1}, s_{i-1}]$;
end

Mapping (Left Move)

1	2	3	4	
A	*B*	*X*	*D*	…

head (under *D*)

	s_1	s_2	s_3	s_4	
s_1	A	*own*			
s_2		B k_1	*own*		
s_3			X	*own*	
s_4				D *end*	

After $\delta(k, \text{C}) = (k_1, \text{X}, \text{L})$ where k is the current state and k_1 the next state

Mapping (Initial)

1	2	3	4	
A	*B*	*C*	*D*	…

head (under *C*)

	s_1	s_2	s_3	s_4	
s_1	A	*own*			
s_2		B	*own*		
s_3			C k	*own*	
s_4				D end	

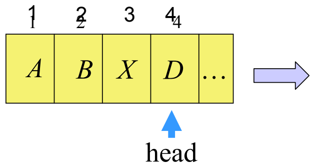
Current state is k
Current symbol is C

Command Mapping (Right move)

$\delta(k, \mathrm{C}) = (k_1, \mathrm{X}, \mathrm{R})$

command $c_{k,\mathrm{C}}(s_i, s_{i+1})$
if *own* **in** $a[s_i, s_{i+1}]$ **and** k **in** $a[s_i, s_i]$ **and** C **in** $a[s_i, s_i]$
then
  delete k **from** $A[s_i,s_i]$;
  delete C **from** $A[s_i,s_i]$;
  enter X **into** $A[s_i,s_i]$;
  enter k_1 **into** $A[s_{i+1}, s_{i+1}]$;
end

Mapping



	s_1	s_2	s_3	s_4	
s_1	A	*own*			
s_2		B	*own*		
s_3			X	*own*	
s_4				D k_1 end	

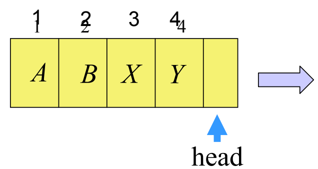
After $\delta(k, \mathrm{C}) = (k_1, \mathrm{X}, \mathrm{R})$ where k is the current state and k_1 the next state

Command Mapping (Rightmost move)

$\delta(k_1, \text{D}) = (k_2, \text{Y}, \text{R})$ at end becomes

command crightmost$_{k,\text{C}}(s_i, s_{i+1})$
if *end* **in** $a[s_i, s_i]$ **and** k_1 **in** $a[s_i, s_i]$ **and** D **in** $a[s_i, s_i]$
then
  delete *end* **from** $a[s_i, s_i]$;
  create subject s_{i+1};
  enter *own* **into** $a[s_i, s_{i+1}]$;
  enter *end* **into** $a[s_{i+1}, s_{i+1}]$;
  delete k_1 **from** $a[s_i, s_i]$;
  delete D **from** $a[s_i, s_i]$;
  enter Y **into** $a[s_i, s_i]$;
  enter k_2 **into** $A[s_i, s_i]$;
end

Mapping



	s_1	s_2	s_3	s_4	s_5
s_1	A	*own*			
s_2		B	*own*		
s_3			X	*own*	
s_4				Y	*own*
s_5					b k_2 end

After $\delta(k_1, \text{D}) = (k_2, \text{Y}, \text{R})$ where k_1 is the current state and k_2 the next state

Rest of Proof

- Protection system exactly simulates a TM
  - Exactly 1 *end* right in ACM
  - 1 right corresponds to a state
  - Thus, at most 1 applicable command in each configuration of the TM
- If TM enters state q_f, then right has leaked
- If safety question decidable, then represent TM as above and determine if q_f leaks
  - Leaks halting state ⇒ halting state in the matrix ⇒ Halting state reached
- Conclusion: safety question undecidable

Other theorems

- Set of unsafe systems is recursively enumerable
  - Recursively enumerable?
- For protection system without the create primitives, (i.e., delete **create** primitive); the safety question is complete in **P-SPACE**
- It is undecidable whether a given configuration of a given monotonic protection system is safe for a given generic right
  - Delete **destroy**, **delete** primitives;
  - The system becomes monotonic as they only increase in size and complexity

Other theorems

- The safety question for biconditional monotonic protection systems is undecidable
- The safety question for monoconditional, monotonic protection systems is decidable
- The safety question for monoconditional protection systems with **create**, **enter**, **delete** (and no **destroy**) is decidable.
- Observations
  - Safety is undecidable for the generic case
  - Safety becomes decidable when restrictions are applied

Schematic Protection Model

- Key idea is to use the notion of a protection type
  - Label that determines how control rights affect an entity
  - Take-Grant:
    - *subject* and *object* are different protection types
  - TS and TO represent subject type set and object set
  - τ(X) is the type of entity X
- A *ticket* describes a right
  - Consists of an *entity name* and a *right symbol*: **X**/*z*
    - Possessor of the ticket **X**/*z* has right *r* over entity **X**
    - Y has tickets **X**/*r*, **X**/*w* -> Y has tickets **X**/*rw*
  - Each entity **X** has a set *dom*(**X**) of tickets **Y**/*z*
  - τ(X/*r:c*) = τ(X)/*r:c* is the type of a ticket

Schematic Protection Model

- Inert right vs. Control right
  - Inert right doesn't affect protection state, e.g. *read* right
  - *take* right in Take-Grant model is a control right
- Copy flag c
  - Every right *r* has an associated copyable right *rc*
  - *r:c* means *r* or *rc*
- Manipulation of rights
  - A link predicate
    - Determines if a source and target of a transfer are "connected"
  - A filter function
    - Determines if a transfer is authorized

Transferring Rights

- *dom*(**X**) : set of tickets that X has
- Link predicate: $link_i(\mathbf{X},\mathbf{Y})$
  - conjunction or disjunction of the following terms
    - $\mathbf{X}/z \in dom(\mathbf{X})$; $\mathbf{X}/z \in dom(\mathbf{Y})$;
    - $\mathbf{Y}/z \in dom(\mathbf{X})$; $\mathbf{Y}/z \in dom(\mathbf{Y})$
    - **true**
  - Determines if **X** and **Y** "connected" to transfer right
  - Examples:
    - Take-Grant: $link(\mathbf{X}, \mathbf{Y}) = \mathbf{Y}/g \in dom(\mathbf{X}) \vee \mathbf{X}/t \in dom(\mathbf{Y})$
    - Broadcast: $link(\mathbf{X}, \mathbf{Y}) = \mathbf{X}/b \in dom(\mathbf{X})$
    - Pull: $link(\mathbf{X}, \mathbf{Y}) = \mathbf{Y}/p \in dom(\mathbf{Y})$
    - Universal: $link(\mathbf{X}, \mathbf{Y}) =$ **true**
- *Scheme*: a finite set of link predicates is called a scheme

Filter Function

- Filter function:
  - Imposes conditions on when tickets can be transferred
  - f_i: $TS \times TS \rightarrow 2^{TxR}$ (range is copyable rights)
- **X**/*r:c* can be copied from *dom*(**Y**) to *dom*(**Z**) iff $\exists i$ s. t. the following are true:
  - **X**/*rc* $\in$ *dom*(**Y**)
  - $link_i$(**Y**, **Z**)
  - τ(X)/*r:c* $\in f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$
- Examples:
  - If $f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$ = T x R then any rights are transferable
  - If $f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$ = T x RI then only inert rights are transferable
  - If $f_i(\tau(\mathbf{Y}), \tau(\mathbf{Z}))$ = Θ then no tickets are transferable
- One filter function is defined for each link predicate

SCM Example 1

- Owner-based policy
  - Subject U can authorize subject V to access an object F iff U owns F
  - Types: TS= {*user*}, TO = {*file*}
  - Ownership is viewed as copy attributes
    - If U owns F, all its tickets for F are copyable
  - RI: { *r:c*, *w:c*, *a:c*, *x:c* }; RC is empty
    - read, write, append, execute; copy on each
  - $\forall$ **U**, **V** $\in$ user, *link*(**U**, **V**) = **true**
    - Anyone can grant a right to anyone else if they posses the right to do so (copy)
  - *f*(*user*, *user*) = { *file*/*r*, *file*/*w*, *file*/a, *file*/*x* }
    - Can copy read, write, append, execute

SPM Example 1

- *Peter* owns file *Doom*; can he give *Paul* execute permission over *Doom*?
  1. τ(*Peter*) is *user and* τ(*Paul*) is *user*
  2. τ(*Doom*) is *file*
  3. *Doom*/*xc* $\in$ *dom*(*Peter*)
  4. *Link*(*Peter*, *Paul*) = TRUE
  5. τ(*Doom*)/*x* $\in$ *f*(τ(*Peter*), τ(*Paul*)) - because of 1 and 2

  Therefore, Peter can give ticket *Doom*/*xc* to Paul

SPM Example2

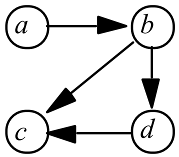
- Take-Grant Protection Model
  - *TS* = { subjects }, *TO* = { objects }
  - *RC* = {*tc*, *gc*}, *RI* = {*rc*, *wc*}
    - Note that all rights can be copied in T-G model
  - *link*(**p**, **q**) = **p**/*t* $\in$ *dom*(**q**) v **q**/*t* $\in$ *dom*(**p**)
  - *f*(*subject*, *subject*) = { *subject*, *object* } $\times$ { *tc*, *gc*, *rc*, *wc* }
    - Note that any rights can be transferred in T-G model

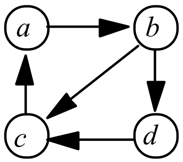
Demand

- A subject can demand a right from another entity
  - Demand function $d{:}TS \rightarrow 2^{T \times R}$
  - Let *a* and *b* be types
    - $a/r{:}c \in d(b)$: every subject of type *b* can demand a ticket $X/r{:}c$ for all *X* such that $\tau(X) = a$
  - A sophisticated construction eliminates the need for the demand operation – hence omitted

Create Operation

- Need to handle
  - type of the created entity, &
  - tickets added by the creation
- Relation $can\bullet create(a, b) \subseteq TS \times T$
  - A subject of type *a* can create an entity of type *b*
- Rule of *acyclic creates*
  - Limits the membership in $can\bullet create(a, b)$
  - If a subject of type *a* can create a subject of type *b*, then none of the descendants can create a subject of type *a*





Create operation Distinct Types

- create rule *cr*(*a*, *b*) specifies the
  - tickets introduced when a subject of type *a* creates an entity of type *b*
- **B** object: $cr(a, b) \subseteq \{ b/r{:}c \in RI \}$
  - Only inert rights can be created
  - A gets B/r:c iff $b/r{:}c \in cr(a, b)$
- **B** subject: *cr*(*a*, *b*) has two parts
  - $cr_P(a, b)$ added to **A**, $cr_C(a, b)$ added to **B**
  - **A** gets **B**/*r*:*c* if *b*/*r*:*c* in $cr_P(a, b)$
  - **B** gets **A**/*r*:*c* if *a*/*r*:*c* in $cr_C(a, b)$

Non-Distinct Types

- *cr*(*a*, *a*): who gets what?
  - *self*/*r*:*c* are tickets for creator
  - *a*/*r*:*c* tickets for the created
- $cr(a, a) = \{ a/r{:}c, self/r{:}c \mid r{:}c \in R \}$
- $cr(a, a) = cr_C(a, b) | cr_P(a, b)$ is attenuating if:
  1. $cr_C(a, b) \subseteq cr_P(a, b)$ and
  2. $a/r{:}c \in cr_P(a, b) \Rightarrow self/r{:}c \in cr_P(a, b)$
- A scheme is attenuating if,
  - *For all types a*, $cc(a, a) \rightarrow cr(a, a)$ is attenuating

Examples

- Owner-based policy
  - Users can create files: *cc*(*user, file*) holds
  - Creator can give itself any inert rights: *cr*(*user*, *file*) = {*file*/*r*:*c*| *r* ∈ *RI*}
- Take-Grant model
  - A subject can create a subject or an object
    - *cc*(*subject*, *subject*) *and cc*(*subject*, *object*) hold
  - Subject can give itself any rights over the vertices it creates but the subject does not give the created subject any rights (although grant can be used later)
    - $cr_C(a, b) = \Theta$; $cr_P(a, b)$ = {*sub/tc, sub/gc, sub/rc, sub/wc*}

    Hence,
    - *cr*(*sub*, *sub*) = {*sub/tc, sub/gc, sub/rc, sub/wc*} | Θ
    - *cr*(*sub*, *obj*) = {*obj/tc, obj/gc, obj/rc, obj/wc*} | Θ

Safety Analysis in SPM

- Idea: derive *maximal state* where changes don't affect analysis
  - Indicates all the tickets that can be transferred from one subject to another
  - Indicates what the maximum rights of a subject is in a system
- Theorems:
  - A maximal state exists for every system
  - If parent gives child only rights parent has (conditions somewhat more complex), can easily derive maximal state
  - Safety: If the scheme is acyclic and attenuating, the safety question is decidable

Typed Access Matrix Model

- Finite set *T* of types (*TS* $\subseteq$ *T* for subjects)
- Protection State: (*S*, *O*, τ, *A*)
  - $\tau : O \rightarrow T$ is a type function
  - Operations same as in HRU model except create adds type
- τ is child type iff command create creates subject/object of type τ
- If parent/child graph from all commands acyclic, then:
  - Safety is decidable
  - Safety is NP-Hard
  - Safety is polynomial if all commands limited to three parameters

HRU *vs*. SPM

- SPM more abstract
  - Analyses focus on limits of model, not details of representation
- HRU allows revocation
  - SPM has no equivalent to delete, destroy
- HRU allows multiparent creates, SPM does not
  - SPM cannot express multiparent creates easily, and not at all if the parents are of different types because *can•create* allows for only one type of creator
  - Suggests SPM is less expressive than HRU

Comparing Models

- Expressive Power
  - HRU/Access Control Matrix subsumes Take-Grant
  - HRU subsumes Typed Access Control Matrix
  - SPM subsumes
    - Take-Grant
    - Multilevel security
    - Integrity models
- What about SPM and HRU?
  - SPM has no revocation (delete/destroy)
- HRU without delete/destroy (monotonic HRU)
  - MTAM subsumes monotonic mono-operational HRU

Extended Schematic Protection Model

- Adds “joint create”: new node has multiple parents
  - Allows more natural representation of sharing between mutually suspicious parties
    - Create joint node for sharing
- Monotonic ESPM and Monotonic HRU are equivalent