Computer Security, CSI 424/524

Lecture 3 - Decidability and Foundational Results
Secure Computer Systems: Some Fundamental Questions

How can we determine if a given computer system is secure?

- We need to define secure.
  - Want a broad definition.
  - Use ACM model without special rights copy and own.
  - Enforce attenuation of privilege.
  - Let $R$ be the set of rights in the system.

Definition 1. [Leaked Right] When a generic right $r \in R$ is added to an element of the access control matrix not already containing $r$, right $r$ is said to be leaked.

- Note: Subjects with authorized transfers can be treated as “trusted” and removed from the system.
- Leaking a right effectively enters the system into an unauthorized state.

Is there an algorithm that can check to see if a system is secure?

- This is a decidability problem.
Safety and Security

Let a computer system begin in state $s_0$.

Definition 2. [Safe/Unsafe with respect to right $r$] If a system can never leak the right $r$ (including the initial state $s_0$) the system is called *safe with respect to the right* $r$. Otherwise, the system can leak the right $r$, and is said to be *unsafe with respect to the right* $r$.

Safety refers to the model, security refers to the implementation.

- Safety is necessary but not sufficient for a secure system.

Definition 3. [Safety Question] The *safety question* asks: Is there an algorithm for determining if a given protection system with initial state $s_0$ is safe with respect to a generic right $r$?
Results for Mono-Operational Systems

Theorem 4. [Harrison, Ruzzo, Ullman] There is an algorithm that will determine whether a given mono-operational protection system with an initial state $s_0$ is safe with respect to a generic right $r$.

Proof:

- Let there be $k$ commands, $c_1, \ldots, c_k$ that represent the shortest sequence of commands leaking right $r$ from initial state $s_0$. We can be sure certain classes of commands don’t appear in $c_1, \ldots, c_k$ since:
  - **Commands don’t test for the absence of rights, so delete and destroy commands don’t affect the ability of a right to leak, and can be omitted.**
  - **Only the first create matters, since we can rewrite constructs testing the rights of $a[s_1, o_1]$ and $a[s_2, o_2]$ as testing the rights of $a[s_1, o_1] \cup a[s_2, o_2]$. Thus we can merge the rights and only create the first subject (since subject can be objects). Thus there are $|S_0| + 1$ subjects and $|O_0| + 1$ objects.**
  - **Enter rights commands need to have their target adjusted to the single new subject created as described above.**

- Let there be $|R| = n$ (i.e. there are $n$ distinct rights). Then there can be at most $n$ enter commands per ACM element, which means: $k \leq n(|S_0| + 1)(|O_0| + 1) + 1$. 

CSI 424/524, William A. Maniatty, Dept. of Computer Science, University at Albany
Results for Multi-Operation Systems

Unfortunately, the general case is harder.

Theorem 5. [Harrison, Ruzzo, Ullman] It is undecidable whether a given state of a state protection system is safe for a given right.

Outline of the proof:

- Enumerating all possible states is computable (but expensive), however, this will not for all protection systems.
- We want to show that an undecidable problem (the Halting Problem) can be solved if we can solve the safety question.
- To do this, we reduce the halting problem to the safety question.
Some Background: Automata Theory

Please recall the Halting Problem and Turing Machine definitions

- The Halting Problem asks can we know if an arbitrary program on a Turing machine will terminate?
- Definition 6. [Turing Machine] A Turing machine $T$ is composed of:
  - An infinite tape divided into an infinite number of cells, where each cell contains a symbol. The alphabet contains a special blank symbol not in the input for use on the tape.
  - A head which can read and write symbols from/to the tape and move one cell to the left ($L$) or right ($R$).
  - A state register that records the current state of $T$.
  - A transition function (or action table), $\delta$ which given the current state and input tells the machine the next state, how to move the head and what symbol to write.

More formally $T$ is a tuple, $T = (s, Q, K, M, \delta)$, where:

- $K$ is a finite set of states
- $s$ is the start state, $s \in K$.
- $Q$ is the set of accepting (final) states, $Q \subseteq K$.
- $M$ is the alphabet of symbols the machine processes, including the blank symbol which is not in the input alphabet.
- $\delta$ is the transition function, $\delta : K \times M \rightarrow K \times M \times \{L, R\}$.
Proof of Undecidability 1 of 5

Reductio ad Absurdum (Proof by contradiction)

- Construct a method for expressing an arbitrary Turing machine as a Protection State system, with the Turing machine, $T$, entering a final state (i.e. halting) corresponding to leaking of a generic right.
  - Let $n$ represent the rightmost cell scanned by $T$, i.e. cells $1, 2, \ldots, n$ have been scanned by Turing machine $T$.
  - We can represent each cell as a Subject in the resulting protection model.
  - A generic right own is defined such that $s_i$ owns $s_{i+1}$ for $1 \leq i < k$ (since there are $k$ subjects).
  - If cell $i$ has symbol $A$ then subject $s_i$ has generic right $A$ over itself.
  - Subject $s_k$ has end rights over itself.
  - To indicate the heads position and state register, if the head is cell $i$ and the current state is $p$ then subject $s_i$ has rights $p$ over itself.

![Diagram of Turing machine and protection model]

CSI 424/524, William A. Maniatty, Dept. of Computer Science, University at Albany 7
Proof of Undecidability 2 of 5

• How to handle transition functions, consider $\delta(p, A) = (q, B, L)$ when the head is not in the leftmost cell.

command $c_{p,A}(s_i, s_{i-1})$
  
  if $(own \in a[s_{i-1}, s_i] \text{ and } A \in a[s_i, s_i])$
    
    delete $p$ from $a[s_i, s_i]$;
    delete $A$ from $a[s_i, s_i]$;
    enter $B$ from $a[s_i, s_i]$;
    enter $q$ from $a[s_{i-1}, s_{i-1}]$;
  

A similar treatment works for the leftmost cell, but we substitute $s_1, s_1$ for the parameters (i.e. treat $i - 1 = i$ for this special case).
Proof of Undecidability 3 of 5

- How to handle transition functions, consider $\delta(p, A) = (q, B, R)$ when the head is not in the rightmost cell.

```plaintext
command $c_{p,A}(s_i, s_{i+1})$

if $(own \in a[s_i, s_{i+1}]$ and $A \in a[s_i, s_i])$

    delete $p$ from $a[s_i, s_i]$;
    delete $A$ from $a[s_i, s_i]$;
    enter $B$ from $a[s_i, s_i]$;
    enter $q$ from $a[s_{i+1}, s_{i+1}]$;

}
Proof of Undecidability 4 of 5

• How to handle transition functions, consider $\delta(p, A) = (q, B, R)$ when the head is in the rightmost cell.

    command $c_{\text{rightmost}}_{p,A}(s_i, s_{i+1})$
    if $(\text{end} \in a[s_i, s_i] \text{ and } p \in a[s_i, s_i] \text{ and } A \in a[s_i, s_i])$
    delete $\text{end}$ from $a[s_i, s_i]$;
    create new subject $s_{i+1}$;
    enter own into $a[s_i, s_{i+1}]$;
    enter $\text{end}$ into $a[s_{i+1}, s_{i+1}]$;
    delete $p$ from $a[s_i, s_i]$;
    delete $A$ from $a[s_i, s_i]$;
    enter $B$ from $a[s_i, s_i]$;
    enter $q$ from $a[s_{i+1}, s_{i+1}]$;
    

Proof of Undecidability 5 of 5

It can be shown that the resulting ACM exactly simulates the corresponding Turing machine.

- Only one right in the corresponding ACM corresponds to a state.
- In each configuration of the protection system, there is only one applicable command (as per case analysis).

Suppose the Turing machine enters state \( q_j \)

- Then, the protection system has leaked right \( q_j \)
- Either the protection system is safe for the right \( q_f \in Q \) (a halting state) or it is not.
- But whether \( T \) will enter \( q_f \) corresponds to the undecidable halting problem.
Theorem 7. [Denning] The set of unsafe systems is recursively enumerable.

Disallowing the create command makes the system tractable.

- Definition 8. [P-Space (Computational Complexity)] P-space refers to a computational complexity class of decision problems that can be solved by a Turing machine using polynomial storage (space).

- Theorem 9. [Harrison Ruzzo Ullman] For protection systems without the create primitive, the question of safety is complete in P-Space.

- Definition 10. [Monotonic Protection Systems] Protection systems without the delete and destroy commands are called monotonic (as they only grow in size).

- Theorem 11. [Harrison Ruzzo] It is undecidable if a given configuration of a given monotonic protection system is safe for a given generic right.
Consider what happens if we restrict the complexity of predicates in the conditional statements in monotonic protection systems.

- Theorem 12. [Harrison Ruzzo] The safety question for biconditional (two conditions per command) monotonic protection systems is undecidable.
- Simplifying helps to get a positive result
  
  Theorem 13. [Harrison Ruzzo] The safety question for monoconditional (one condition per command) monotonic protection systems is decidable.

- A stronger result is:
  
  Theorem 14. [Harrison Ruzzo] The safety question for monoconditional (one condition per command) protection systems with create, enter, delete primitives, but without destroy primitives is decidable.
Now What?

Using a straight ACM protection system model, can we determine if a system is secure

- There are several negative results
- And a few weak positive results.
- It is unlikely that a real system is sufficiently constrained to allow for direct analysis.

So what do we do?

- Say I guess it’s secure and go home?
  - What about liability?
- Or perhaps we can try other (potentially less general) models.
The Take-Grant Protection Model State

The Take-Grant Protection model supports determining the safety of a system with specific rules.

Represents a protection system’s state as a labeled Directed Graph (digraph) $G = (V, E)$.

- Vertices are either subjects, objects, or either subjects or objects.
- Edges are labeled according to the set of rights that the source vertex has to the destination vertex. We denote an edge as an ordered pair $(sourcevertex, destinationvertex)$.

- Rights are elements of a predefined set $R$.

  ▶ There are 2 distinguished rights, $t$ for take and $g$ for grant.
The Take-Grant Protection Model Transition Rules

Changing the protection state of the system corresponds to changing the graph according to four graph rewriting rules (described later).

A single transition is shown by ⊢ between the graphs (the graphs may be drawn).

A finite sequence of transitions deriving $G$ from $G_0$ is denoted $G_0 \vdash^* G$.

Definition 15. [Witness] A witness refers to a sequence of rewriting rules $G_0 \vdash^* G$. The rules may be listed and their application may be drawn.
There are 4 graph rewriting rules, called the de jure rules, in the Take-Grant Protection Model, where the initial graph \( G_0 = (V_0, E_0) \) is transformed into the final graph \( G_1 = (V_1, E_1) \).

- **Take Rule**: Let \( x, y, z \in V_0 \), and \( x \) be a subject and let edge \((x, y) \in E_0\) be labeled \( \gamma \subseteq R \), \( t \in \gamma \) and edge \((y, z) \in E_0\) be labeled \( \beta \) and \( \alpha \subseteq \beta \). The take rule creates a new graph \( G_1(V_1, E_1) \) by adding an edge \((x, z)\) labeled \( \alpha \) to \( G_0 \), so \( V_0 = V_1 \) and \( E_1 = E_0 \cup \{(x, z)\} \).

\[ \text{This rule is written ""} x \text{ takes } \alpha \text{ (to } y \text{) from } z \text{"".} \]
Graph Rewriting Rules 2 of 4

- Grant Rule: Let $x, y, z \in V_0$, and $z$ be a subject and let edge $(x, z) \in E_0$ be labeled $\gamma \in R$, $g \in \gamma$ and edge $(y, z)$ be labeled $\beta$ and $\alpha \subseteq \beta$. The grant rule creates a new graph $G_1(V_1, E_1)$ by adding an edge $(x, y)$ labeled $\alpha$ to $G_0$, so $V_0 = V_1$ and $E_1 = E_0 \cup \{(x, y)\}$.

  ▶ This rule is written “$x$ grants $\alpha$ (to $y$) from $z$”.

Grant Rule

$G_0$

$\Rightarrow$

$G_1$
Create Rule: Let $x \in V_0$, and $x$ be a subject $\alpha \subseteq R$. The create rule creates a new graph $G_1(V_1, E_1)$ by adding a vertex, $y, y \in V_1$ and edge $(x, y)$ labeled $\alpha$ to $G_0$, so $V_0 = V_1 \cup \{y\}$ and $E_1 = E_0 \cup \{(x, y)\}$.

This rule is written “$x$ creates ($\alpha$ to) new vertex $y$”.

Create Rule
Graph Rewriting Rules 4 of 4

- Remove Rule: Let \( x, y \in V_0 \), and \( x \) be a subject and let edge \((x, y) \in E_0\) be labeled \( \beta \in R \) and \( \alpha \subseteq \beta \). The remove rule creates a new graph \( G_1(V_1, E_1) \) by deleting the rights \( \alpha \) from the edge \((x, y)\), so that the label becomes \( \beta - \alpha \). If \( \alpha = \beta \) then \( \beta - \alpha = \emptyset \) and the edge \((x, y)\) is removed, so \( V_0 = V_1 \) and either \( E_1 = E_0 \) (although the edge label for \((x, y)\) changes) or \( E_1 = E_0 - \{(x, y)\} \).

\[ \text{This rule is written “}(x \text{ removes } (\alpha \text{ to} ) \ y\text{”}. \]

\[ \begin{array}{c}
\text{\( G_0 \)} & \xrightarrow{\beta} & \text{\( y \)} \\
\text{\( x \)} & & \\
\end{array} \longrightarrow \quad \\
\begin{array}{c}
\text{\( G_1 \)} \\
\text{\( x \)} & \xrightarrow{\beta - \alpha} & \text{\( y \)} \\
\end{array} \]

\[ \begin{array}{c}
\text{\( G_0 \)} & \xrightarrow{\beta} & \text{\( y \)} \\
\text{\( x \)} & & \\
\end{array} \longrightarrow \quad \\
\begin{array}{c}
\text{\( G_1 \)} \\
\text{\( x \)} & \xrightarrow{\beta - \alpha} & \text{\( y \)} \\
\end{array} \]

\[ \begin{array}{c}
\text{\( G_0 \)} & \xrightarrow{\beta} & \text{\( y \)} \\
\text{\( x \)} & & \\
\end{array} \longrightarrow \quad \\
\begin{array}{c}
\text{\( G_1 \)} \\
\text{\( x \)} & \xrightarrow{\beta - \alpha} & \text{\( y \)} \\
\end{array} \]

\( \text{Remove Rule} \)
Checking Security Using TG-Models

The TG-model supports analysis of a particular protection system.

- E.G. suppose we want to answer a question like: “Can a competitor access my files?”.
- The analyst must know the rules governing the underlying protection system.
- The model must correctly capture these rules to accurately describe the underlying system.

A fundamental aspect of security is determining if one entity in the system can acquire rights held by another entity.

- In the TG system, the entities are either subjects or objects (or for purposes of theoretical analysis, labeled as unknown type).
- The transfer of privileges is called “sharing”.
- We want to know when sharing can occur.
- Recall: Rights are drawn as labels on edges between entities (vertices).
- Informally, sharing of right $\alpha$ between vertices $x$ and $y$ occurs when an application of the de jure rules allows us to draw an edge from $x$ to $y$ with label $\alpha$ in the TG graph.
A TG system example (In class exercise)

Consider a system with 2 processes, \( p \) and \( q \) that have \( r, w \) access to local memory locations \( u \) and \( v \) respectively.

- What would \( G_0 \) look like?
- Can we express this using the Take Grant model?
- What steps using the de jure rules would we take to create this system if \( G_0 = (V_0, E_0) \) and \( V_0 = \emptyset \) and \( E_0 = \emptyset \).
A TG system example 2 (Solution)

- The system would look like:

```
\begin{align*}
  p & \quad r,w \quad u \\
  \quad & \quad & \\
  q & \quad r,w \quad v
\end{align*}
```

- There is no valid sequence of steps to generate this system from the empty graph $G_0 = (\emptyset, \emptyset)$ since the create rule requires a preexisting subject, and no other rule can add subjects or objects to the system.

- Suppose instead of an initial system, there was a trusted object $s$ (e.g., an operating system) and no other objects in the initial system, so $G_0 = (V_0, E_0)$ where $V_0 = \{s\}$ and $E_0 = \emptyset$. Suppose that we wanted to create a system where $s$ had $g$ (grant) rights over subjects (processes) $p$ and $q$ which has $r, w$ rights over local memory objects $u$ and $v$ respectively.
A TG system example 2 (Solution)

- The system would look like:

- A valid sequence of steps to create the final system from $G_0$ would be:
  - $s$ creates $g$ to new subject vertex $p$
  - $s$ creates $g$ to new subject vertex $q$
  - $p$ creates $r, w$ to new object vertex $u$
  - $q$ creates $r, w$ to new object vertex $v$

- Suppose we want to create a shared memory object $b$ through which both $p$ and $q$ communicate (i.e. have $r, w$ rights). What would the resulting system look like and can we generate it from $G_n$ using the de jure rules?
A TG system example 3 (Solution)

- The system would look like:

- A valid sequence of steps to create this system from $G_n$ would be:
  - $s$ creates $r, w$ to new object vertex $b$
  - $s$ grants $r, w$ to $b$ to vertex $p$
  - $s$ grants $r, w$ to $b$ to vertex $q$
  - $s$ removes $(r, w$ to) $b$

- Suppose we want to limit communication through $b$ so that $p$ sends to $q$, what would the system look like and what de jure rules would need to be applied (try this at home).
Symmetry Rules in TG Models

Bishop’s Lemmas 3.1 and 3.2 show symmetry for take/grant using repeated application of the *de jure* rules.
Proof of Bishop's Symmetry Lemma 3.1

Lemma 3.1 (What we want to prove)

- \( x \) creates \( tg \) to new vertex \( v \)
- \( z \) takes \( g \) to \( v \) from \( x \)
- \( z \) grants \( \alpha \) to \( y \) to \( v \)
- \( x \) takes \( \alpha \) to \( y \) from \( v \)
- \( t \), \( \alpha \), and \( g \) are used in the diagram.
Sharing of Rights in Take-Grant Protection Models

Some definitions

• Definition 16. [CanShare Predicate] The predicate CanShare(α, x, y, G₀) is true for a set of rights α and vertices, x and y if
  1. there is a sequence of protection graphs G₁, . . . , Gₙ, such that G₀ ⊢* Gₙ using only the de jure rules and
  2. in G₁ there is an edge (x, y) labeled α.

• Definition 17. [tg-path] A tg-path is a nonempty sequence v₀, . . . , vₙ of distinct vertices such that for all i, 0 ≤ i < n, there is an edge (vᵢ, vᵢ₊₁) with a label containing either t or g.

• Definition 18. [tg-connected] Vertices are tg-connected if there is a tg-path between them.
Sharing of Rights in Take-Grant Protection Models

Some definitions

- **Definition 19. [CanShare Predicate]** The predicate $\text{CanShare}(\alpha, x, y, G_0)$ is true for a set of rights $\alpha$ and vertices, $x$ and $y$ if
  1. there is a sequence of protection graphs $G_1, \ldots, G_n$, such that $G_0 \vdash^* G_n$ using only the de jure rules and
  2. in $G_1$ there is an edge $(x, y)$ labeled $\alpha$.

- **Definition 20. [tg-path]** A $tg$-path is a nonempty sequence $v_0, \ldots, v_n$ of distinct vertices such that for all $i, 0 \leq i < n$, there is an edge $(v_i, v_{i+1})$ with a label containing either $t$ or $g$.

- **Definition 21. [tg-connected]** Vertices are $tg$-connected if there is a $tg$-path between them.

- **Definition 22. [Island]** An island is a maximal $tg$-connected subject only subgraph.
  - It can be shown that any right possessed by any vertex in an island can be shared with any other vertex in the island.
  - Can we transfer rights between islands? If so, how?
Sharing of Rights in Take-Grant Protection Models

Some Notation and definitions

- Notation: $\overleftarrow{g}, \overrightarrow{g}, \overleftarrow{t}, \overrightarrow{t}$ use arrows to indicate directions of connection between vertices of edges with $t$ or $g$ in the labels.

  - We can construct words composed of symbols from $\{\overleftarrow{g}, \overrightarrow{g}, \overleftarrow{t}, \overrightarrow{t}\}$ to describe the sequence of labels and edge directions on a $tg$-path.
  - If a $tg$-path has length 0 it is called a null path and is considered having the reserved label $\nu$.
  - $\overleftarrow{g}^*$ (Kleene Star) means zero or more repetitions of $\overleftarrow{g}$, i.e. $\nu, \overleftarrow{g}, \overleftarrow{g} \overleftarrow{g}, \ldots$, and likewise with all symbols except $\nu$.

- Definition 23. [bridge] A bridge is a $tg$-path with subject endpoints $v_0, v_n$ and has the associated word in $\{\overrightarrow{t}, \overleftarrow{t}, \overrightarrow{t}\overleftarrow{g}t, \overrightarrow{t} \overleftarrow{g}t\}$.

  - Note that since both endpoints are subjects the bridge can be used to transfer rights between them.
  - Rights can be shared between subjects in the same island, can we share rights between islands?
Sharing of Rights in Take-Grant Protection Models

- Definition 24. [SubjectCanShare (Lipton and Snyder)] The predicate SubjectCanShare(α, x, y, G₀) is true iff ∃(x, y) ∈ G₀ labeled α or if both the following hold true at the same time:
  1. ∃s ∈ G₀ where s is a subject with edge (s, y) having label α and
  2. There are islands I₁, . . . , Iₙ, such that x ∈ I₁, s ∈ Iₙ and ∀j, 1 ≤ j < n there is a bridge from Iₗ to Iₗ₊₁.

- Subjects can act, but objects cannot, so we need rules and notation for handling sharing of rights in cases may involve objects.

  - Definition 25. [Initially Spans] A vertex 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 \{t → *g\} ∪ \{ν\}.

    - I.e. x initially spans to y if x can grant some of its rights to y.

  - Definition 26. [Terminally Spans] A vertex 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 \{t\} ∪ \{ν\}.

    - I.e. x terminally spans to y if x can take any right y possesses.

These rights imply take and grant are not symmetric if an object is an end point.
The CanShare Theorem

Sharing can now be generalized to handle objects as endpoints, with the following necessary and sufficient condition for rights to be transferred from vertex \( y \) to another vertex \( x \).

\[\text{Theorem 27. [CanShare (Jones, Lipton and Snyder)]} \quad \text{The predicate } \text{CanShare}(\alpha, x, y, G_0) \text{ is true iff } \exists (x, y) \in G_0 \text{ labeled } \alpha \text{ or if the following all hold true at the same time:}
\]

1. \( \exists s \in G_0 \text{ with edge } (s, y) \text{ having label } \alpha \),
2. There is a subject vertex \( x' \) such that either \( x = x' \) or \( x' \) initially spans to \( x \).
3. There is a subject vertex \( s' \) such that either \( s = s' \) or \( s' \) terminally spans to \( s \).
4. There are islands \( I_1, \ldots, I_n \), such that \( x' \in I_1, s' \in I_n \) and \( \forall j, 1 \leq j < n \) there is a bridge from \( I_j \) to \( I_{j+1} \).

\[\text{Proof Outline:}
\]

- Because, as per property 2, \( s' \) terminally spans to \( s \), \( s' \) can acquire \( \alpha \) rights to \( y \).
- All subjects in \( I_n \) can acquire \( \alpha \) rights to \( y \) (follows from definition of Island).
- The presence of bridges between the islands permits island \( I_{j-1} \) to get these rights from island \( I_j \) for \( 1 < j \leq n \) (allowing for induction and trascitivity).
- Since, as per property 4, \( x' \) initially spans to \( x \) and can pass those rights to \( x \).
Corollary 28. [To Theorem 27 (Jones, Lipton and Snyder)] There is an algorithm of complexity $O(|V_0| + |E_0|)$ that tests the CanShare predicate, where $V_0$ is the set of vertices and $E_0$ is the set of edges in $G_0$. 
Some Properties of TG Models

Recall that in many real systems, the system initially has a single trusted entity (the operating system) that is modeled as a single subject that creates all other subjects. In these systems the resulting TG graph has the following properties.

- **Theorem 29.** [(Snyder)] Let $G_0$ be a protection graph containing exactly one subject vertex, no edges and $R$ be a set of rights. Then $G_0 \vdash^* G$ iff $G$ is a finite directed acyclic graph containing subjects and objects only, with edges labeled with nonempty subsets of $R$ and one subject with no incoming edges.

This allows us to bound the amount of work required to construct $G$ from $G_0$.

- **Corollary 30.** [To Theorem 29 (Snyder)] A $k$-component graph with $n$-edge can be constructed in $t$ rule applications, where $2(k - 1) + n \leq t \leq 2(k - 1) + 3n$. 
Defining Theft in the Take Grant Model

The proof of Theorem 27 about CanShare shows that cooperation is needed by all subjects in the witness.

Stealing in the Take Grant model refers to when some node $x$ acquires a right, say $\alpha$ to $y$ without any node granting $\alpha$ to $y$. We can define a predicate describing vulnerability to theft:

- Definition 31. [CanSteal] Let $G_0$ be a protection graph containing distinct vertices $x$ and $y$ and let $R$ is a set of rights with $\alpha \subseteq R$. The predicate $\text{CanSteal}(\alpha, x, y, G_0)$ holds true if there is no edge $(x, y)$ labeled $\alpha$ in $G_0$ and the following hold true simultaneously:
  1. $\exists (x, y) \in G_n$ with $(x, y)$ labeled $\alpha$.
  2. There is a sequence of rule applications $\rho_1, \ldots, \rho_n$ such that $G_{i-1} \vdash G_i$ using $\rho_i$.
  3. For all vertices $v, w \in G_{i-1}$, if there is an edge $(v, y) \in G_0$ labeled $\alpha$, then $\rho_i$ is not of the form "$v$ grants $\alpha$ to $y$ to $w$".
An example of theft

Show $s$ can steal $\alpha$ from $w$

$u$ grants $(t \to v)$ to $s$

$s$ takes $(t \to u)$ from $v$

$s$ takes $(\alpha \to w)$ from $u$
Conditions Allowing Theft in the Take Grant Model

- **Theorem 32. [CanSteal, Necessary and Sufficient Conditions (Snyder)]** The predicate $\text{CanSteal}(\alpha, x, y, G_0)$ holds true iff the following hold true simultaneously:

  1. $\not\exists (x, y) \in G_0$ with $(x, y)$ labeled $\alpha$.
  2. There is a subject vertex $x' \in G_0$ such that either $x' = x$ or $x'$ initially spans to $x$.
  3. There is a vertex $s \in G_0$, with an edge labeled $\alpha$ to $y$ in $G_0$ and for which $\text{CanShare}(\alpha, x, y, G_0)$ holds.

- **Proof:**
  - $\Rightarrow$ --- Assume the conditions hold, proof is done via construction of the path.
  - $\Leftarrow$ --- Assume $\text{CanSteal}(\alpha, x, y, G_0)$ holds
    - There is no $\alpha$ labeled edge (as per condition ?? in Definition 31 of CanSteal).
    - In the final graph, $G_n$ there will be an edge $(x, y) \in G_n$ labeled $\alpha$ as per condition ?? of Theorem 27 since $\text{CanShare}(\alpha, x, y, G_0)$ holds.
    - As per the CanShare theorem ?? condition 2, $s$ must exist.
    - $\text{CanShare}(\alpha, x', s, G_0)$ - $s$ cannot grant $\alpha$ to $y$ (by definition of CanSteal). It can be shown that applying the take rule accomplishes the sharing.
Conspiracy in the Take Grant Model

Conspiracy --- A conspiracy relies on the set of cooperating subjects in the theft witness.

- Want to find the minimal set of subjects needed to allow \( \text{CanShare}(a, x, y, G_0) \).

Definition 33. [Access set] Access set \( A(y) \) with focus \( y \) is 
\[ A(y) = \{ y \} \cup \text{set of vertices which } y \text{ initially spans to } \cup \text{ the set of vertices to which } y \text{ terminally spans}. \]
Note that \( y \) must be a subject

Definition 34. [Deletion set] The deletion set \( \delta(y, y') \) is all \( z \in A(y) \cap A(y') \) such that

- \( y \) initially spans to \( z \) and \( y' \) terminally spans to \( z \) and
- \( y \) terminally spans to \( z \) and \( y' \) initially spans to \( z \) and
- \( z = y \) and
- \( z = y' \).

Create a conspiracy graph that shows how rights can flow.

- if \( \delta(y, y') \neq \emptyset \) add an edge from \( y \) to \( y' \).
Quick Recap

The HRU (Harrison Ruzzo Ullman) Model is general, but is undecidable.

The Take Grant Model can be applied to specific systems, and those applications can be analyzed.

So why is the HRU model undecidable but the TG model decidable?
The Schematic Protection Model (SPM) is type based.

- Rights in SPM are partitioned into the set of control rights (RC) that allow updates to the rights in the protection system or inert rights (RI).
  - E.g. create, take, grant and remove are control rights, while read rights is inert.
  - SPM ignores the effect of applying inert rights, but not control rights.
  - Rights are manipulated in SPM via two relationships, the Link Predicate and the Filter Function.

- The copy flag, denoted \( c \), is an attribute that can be applied to a right, say \( r \) with copy flag is denoted \( r : c \), and allows the owner to transfer the associated ticket to another domain.

- Protection Types are labels describing how control rights affect that entity.
  - The set of protection types is denoted \( T \), which is partitioned into the set of subject types \( TS \) and the set of object types \( TO \).

- Tickets are descriptions of rights. An entity has a set of tickets describing its rights, called a domain.
  - Notation: Ticket \( X/r \) allows the holder \( r \) rights to entity \( X \), and \( \text{dom}(X) \) denotes the domain of \( X \).
Link Predicates in SPM

Links describe relationships between two subjects depending on only the tickets possessed by the subjects.

- **Definition 35. [Link Predicate \( \text{link}_i(X, Y) \)]** Let \( X, Y \) be subjects and \( z \in RC \) be an arbitrary control right. Recall that \( \text{dom}(X) \) denotes the tickets held by \( X \). Then the Link Predicate \( \text{link}_i(X, Y) \) is a conjunction or disjunction (but not a negation) of the following terms:
  1. \( X/z \in \text{dom}(X) \)
  2. \( X/z \in \text{dom}(Y) \)
  3. \( Y/z \in \text{dom}(X) \)
  4. \( Y/z \in \text{dom}(Y) \)
  5. \( \text{true} \)

- **Definition 36. [Scheme]** A scheme is a finite set of link predicates \( \{\text{link}_i \mid 1 \leq i \leq n\} \). If \( n = 1 \) we omit the subscript \( i \).

- **Some Examples**
  - \( \text{link}(X, Y) = X/b \in \text{dom}(X) \) connects \( X \) to every other entity \( Y \) provided \( X \) has \( b \) rights over itself.
  - \( \text{link}(X, Y) = \text{true} \) -- the universal predicate does not depend on tickets held by \( X \) or \( Y \).
Filter Functions in SPM

A filter function, \( f_i \), imposes conditions on when a transfer of tickets can occur, with \( f_i \) associated with \( \text{link}_i \).

- \( f_i : TS \times TS \rightarrow t^{T \times R} \) has
  - Domain is pairs of subjects appearing in a link predicate
  - Range is the set of copyable tickets

- Thus a ticket \( X/r : c \) can be copied from \( \text{dom}(Y) \) to \( \text{dom}(Z) \) iff there is a \( i \) for which the following hold:
  1. \( X/r : c \in \text{dom}(Y) \)
  2. \( \text{link}_i(Y, Z) \)
  3. \( \tau(X)/r : c \in f_i(\tau(Y), \tau(Z)) \)
An SPM Example

Consider an Owner-Based policy, i.e. a subject $U$ can authorize another subject $V$ to access $F$ iff $U$ owns $F$.

- Suppose users are the subjects and files are the objects, then $TS = \{\text{users}\}$ and $TO = \{\text{files}\}$.
- Typical types of access might be Read, Write, Append and eXecute.
- Ownership can be modeled using the copy attribute.
  - The set of inert rights are $RI = \emptyset$.
  - The set of control rights are $RC = \{r : c, w : c, a : c, x : c\}$.
- Since the owner can authorize any subject to access an object, $\text{link}(U, V) = \text{true}$.
- The filter function supports authorization for all rights, so
  
  $$f(\text{user}, \text{user}) = \{\text{file}/w, \text{file}/r, \text{file}/a\text{file}/x\}$$
Another SPM Example

Consider the take-grant model, suppose that subjects may have read or write access to objects

- The set of subject types is $TS = \{\text{subject}\}$.
- The set of object types is $TO = \{\text{object}\}$.
- The set of inert rights is $RI = \{r, w\}$.
- The set of control rights is $RC = \{t, g\}$.
- Transferring rights requires $t$ or $g$ access, so

  $$\text{link}(p, q) = p/t \in \text{dom}(q) \lor q/g \in \text{dom}(p)$$

- Since any right can be transferred between linked subjects, the filter function is:

  $$f(\text{subject}, \text{subject}) = \{\text{subject, subject}\} \times \{tc, gc, rc, wc\}$$
The SPM Demand Operation

The *demand function*, \(d : TS \times TS \rightarrow 2^{T \times R}\), authorizes a subject to demand a right from another entity.

- Let \(a, b \in TS\) be subject types and \(r \in R\) be a right.
- Then \(a/r : c \in d(b)\) means that every subject of type \(b\) can demand a ticket \(X/r : c\) for all \(X\) such that \(\tau(X) = a\).
- Sandhu demonstrated that a careful construction (omitted here) can avoid the need for demand functions in SPM.
  > So more recently developed models don’t employ demand functions.
The SPM Create Operation

The *create operation* adds types and their rights to SPM systems, consider two issues

- When are creates permitted (can-create)
- The create-rule describing adding types or entities and their rights.
The SPM Can-Create

The *Can-Create* \((cc)\) relation \(cc \subseteq TS \times T\) indicates that an object of type \(a\) can create an entity of type \(b\) iff \(cc(a, b)\) holds.

- Sometimes for notational convenience (as per Sandhu) \(cc : TS \rightarrow 2^T\)
  - so if we write \(cc(a) = S, a \in TS, S \subseteq T\), subjects of type \(a\) are permitted to create entities whose types are in \(S\).

- We can construct a can create relation graph \(G_C = (V_C, E_C)\) with types as vertices (so \(T = V_C\)) and edge \((a, b), a, b \in V_C\) iff \(cc(a, b)\).

- The *rule of acyclic create* states that the can create relation graph should not have cycles.
The SPM Create-Rule

Let \( a \in TS \) be a subject type and \( b \in T \) be an entity type.

The *create-rule*, \( cr(a, b) \), specifies the set of tickets generated when subject \( A \) of subject type \( a = \tau(A) \) creates entity \( B \) of type \( b = \tau(B) \).

- if \( b \in TO \) (i.e. \( B \) is an object of type \( b \)), the rule specifies that \( cr(a, b) \subseteq \{b/r : c \in RI\} \) (i.e. only inert tickets can be generated).
- otherwise \( b \in TS \) (i.e. \( B \) is a subject of type \( b \)), has two sets of rights when
  - if \( a \neq b \), i.e. \( a \) and \( b \) are different types then
    - **if \( a \neq b \)**, i.e. \( a \) and \( b \) are different types then
      - \( cr_p(a, b) \) --- “parent rights” inserted into \( a \),
        \( \text{dom}(A) \) gets \( B/r : c \) iff \( b/r : c \in cr_p(a, b) \).
      - \( cr_c(a, b) \) --- “child rights” inserted into \( b \),
        \( \text{dom}(B) \) gets \( A/r : c \) iff \( a/r : c \in cr_c(a, b) \).
  - **Otherwise \( a = b \)** so parent and child are the same type (i.e. \( cr(a, a) \))
    - Tickets belonging to the creator (parent) are labeled \( \text{self}/r : c \)
    - Tickets belonging to the created (child) are labeled \( a/r : c \).
Attenuating Creates and Safety Result

Definition 37. [attenuating create rule] A create rule, $cr(a, b)$, is **attenuating** if:

- $cr_c(a, b) \subseteq cr_p(a, b)$ and
- $a/r : c \in cr_p(a, b) \Rightarrow self/r : c \in cr_p(a, b)$

A **scheme is attenuating** if for all types $a$ such that $cc(a, a)$ then $cr(a, a)$ is attenuating.

A Safety Result

- If the scheme is acyclic and attenuating, security is decidable.

Safety results analyzed using an approach like Max Flow problem.

- Each system has a maximal state (from theorems in book)
- The attenuating property of a create rule implies that the parent cannot give the child any rights it does not already possess
  - ▶ This is analogous to conservation of flow.
  - ▶ This permits derivation of the maximal state.
Expressive Power of SPM relative to HRU 1 of 2

Both HRU and SPM capable of expressing any take/grant model

- HRU/Access Control matrix approach can express take/grant model
- SPM can express take/grant models since
  - Subject/Object protection types are in SPM
  - SPM’s tickets are rights labels on edges
  - take/grant are control rights
What about relative the expressiveness of SPM and HRU?

- HRU - Can Support ‘‘Multi-parent’’ create, e.g.

  ```
  // creates object \textit{o} and gives subjects \textit{s}_1 \text{ and } \textit{s}_2
  // gives \textit{r} rights over \textit{o}
  // precondition \textit{s}_0 \text{ and } \textit{s}_1 \text{ must have the ‘‘parent right’’}, \textit{p}
  command MultiCreate(\textit{s}_0, \textit{s}_1, \textit{o}){
    \text{if } (p \in a[\textit{s}_0, \textit{o}] \text{ and } p \in a[\textit{s}_1, \textit{o}]){
      create \textit{o};
      enter \textit{r} into a[\textit{s}_0, \textit{o}];
      enter \textit{r} into a[\textit{s}_1, \textit{o}];
    }
  }
  ```

- SPM - Lacks Multi-parent create and has no revocation of rights/removal of entities

**Multi-Parent create solves problems of mutual suspicion.**

- Parents jointly create a \textit{proxy} and each gives proxy the rights needed to do the job.
Extended SPM

Extended SPM (ESPM) adds multiparent create to SPM

Consider how SPM create works works

- CanCreate relation
  - Recall, in traditional SPM $cc \subseteq TS \times T$
  - ESPM generalizes this so $cc \subseteq TS \times TS \times \cdots \times TS \times T$.
  - For notational convenience we may write
    \[
    cc : TS \times TS \times \cdots \times TS \rightarrow T
    \]

- Create Rules:
  - Each parent $X_i, 1 \leq i \leq n$ of child $Y$ in an $n$-parent create has a rule:
    \[
    cr_{P_i}(\tau(X_1), \ldots, \tau(X_n), \tau(Y)) = Y/R_{1,i} \cup X/R_{2,i}
    \]
  - The child $Y$ has a rule of the form:
    \[
    cr_{C}(\tau(X_1), \ldots, \tau(X_n), \tau(Y)) = Y/R_{3} \cup X/R_{4,1} \cup \cdots \cup X/R_{4,n}
    \]
An ESPM Example

Suppose Anne and Bill want to cooperate to do something, but they don’t trust each other.

- e.g. Gamblers want to ensure the bet will be paid.
- e.g. Stock market trading transactions need to be enforced.

Jointly create a proxy endowed with only the needed rights.

- e.g. Designate a bookkeeper/casino/gambling house to escrow wagers.
- e.g. Establish stock exchanges like NASDAQ, NYSE, etc.

Extended SPM (ESPM) is SPM with multi-parent create extensions

- Let $a$ be the type of Anne and Bill i.e. $\tau(Anne) = \tau(Bill) = a$.
- Let $p$ be the type of the proxy
- $cc(a, a) = \{p\}$.
- $cr_{Anne}(a, a, p) = cr_{Bill}(a, a, p) = \emptyset$
  - $\triangleright$ i.e. don’t give Anne or Bill rights over the proxy
- $cr_{proxy}(a, a, p) = \{Anne/x, Bill/x\}$
  - $\triangleright$ Where $x$ is the set of rights needed for the proxy to do its job.
So we considered 2-parent creates

- What if we have more than 2-parents
- Can we emulate an $n$-parent create with a 2-parent create?
  - It seems we can (if we can do $n+1$-parent create) induction will allow us to handle arbitrary $n$

Consider a 3-parent create, what operations must be supported?

- Let parents be subjects $P_1, P_2, P_3$, child be $C$
- CanCreate $cc(\tau(P_1), \tau(P_2), \tau(P_3)) = Z \in T$.
- Create Rule has both parent and child side semantics
  - $cr_{P_1}(\tau(P_1), \tau(P_2), \tau(P_3)) = C/R_{1,1} \cup P_1/R_{2,1}$
  - $cr_{P_2}(\tau(P_1), \tau(P_2), \tau(P_3)) = C/R_{1,2} \cup P_2/R_{2,2}$
  - $cr_{P_3}(\tau(P_1), \tau(P_2), \tau(P_3)) = C/R_{1,3} \cup P_3/R_{2,3}$
  - $cr_{C}(\tau(P_1), \tau(P_2), \tau(P_3)) = C/R_{3} \cup P_1/R_{4,1} \cup P_2/R_{4,2} \cup P_3/R_{4,3}$
ESPM Emulating 3-parent creates

Suppose we only have 2-parent creates but want to emulate a 3 parent-create

- Introduce new entities:
  - **Agents of Parents:** $A_1, A_2, A_3$ where $a_i = \tau(A_i)$
  - **Agent of Child:** $S$, $s = \tau(S)$

- Introduce new right $t$ indicating parentage.
  - So $X/t \in \text{dom}(Y)$ means $Y$ has $X$ as a parent.

- Introduce new rules:
  - **CanCreate:**
    - $cc(\tau(P_1)) = a_1$
    - $cc(\tau(P_2, a_1)) = a_2$
    - $cc(\tau(P_2, a_2)) = a_3$
    - $cc(\tau(a_3)) = S$ (agent of all parents creates child’s agent)
    - $cc(\tau(s)) = C$ (Child’s agent creates child/proxy)
  - **CreateRule:** Need a distinct rule for each parent
    - $cr_{P_{\text{first}}}(\tau(P_1), \tau(P_2), \tau(C))$ refers to rights given to $P_1$.
    - $cr_{P_{\text{second}}}(\tau(P_1), \tau(P_2), \tau(C))$ refers to rights given to $P_2$.
    - $cr_{C}(\tau(P_1), \tau(P_2), \tau(C))$ refers to rights given to $C$
In ESPM the copy right notation omits semicolon

- so $X/r_c$ in ESPM corresponds to $X/r : c$ in SPM

A correct emulation only gives rights to parents *after* successfully creating the child (aborted creates should not escalate privilege).

- Handled in the link predicates
ESPM Emulating 3-parent creates, Create Rule

Create rules for emulating a 3-parent create using 2-parent creates:

- $cr_{P_{first}}(p_3, a_2, a_3) = \emptyset$
- $cr_{P_{second}}(p_3, a_2, a_3) = \emptyset$
- $cr_C(p_3, a_2, a_3) = p_3/Rtc \cup a_2/tc$
  \hspace{1cm} \triangleright \textbf{Child agent } a_3 \text{’s parents set to } p_3 \text{ and } a_2, \text{ child given rights } R \text{ over } p_3$
- $cr_P(a_3, s) = \emptyset$
- $cr_C(a_3, s) = a_3/tc$
  \hspace{1cm} \triangleright \textbf{s has } a_3 \text{ as parent.}$
- $cr_P(s, c) = C/Rtc$
- $cr_C(s, c) = c/R_3 t$
  \hspace{1cm} \triangleright \textbf{Child agent gets full rights over child, child gets } R_3 \text{ over agent.}$
ESPM Emulating 3-parent creates, Link Predicates

Recall Link predicate indicates links over which rights can flow

Idea: No tickets to parents until child is created.

- Enforced by requiring each agent to have its own parent rights.
- link₁(A₁, A₂) = A₁/t ∈ dom(A₂) ∧ A₂/t ∈ dom(A₂)
- link₁(A₂, A₃) = A₂/t ∈ dom(A₃) ∧ A₃/t ∈ dom(A₃)
- link₂(S, A₃) = A₃/t ∈ dom(S) ∧ C/t ∈ dom(C)
- link₃(A₁, C) = C/t ∈ dom(A₁)
- link₃(A₂, C) = C/t ∈ dom(A₂)
- link₃(A₃, C) = C/t ∈ dom(A₃)
- link₄(A₁, P₁) = P₁/t ∈ dom(A₁) ∧ A₁/t ∈ dom(A₁)
- link₄(A₂, P₂) = P₂/t ∈ dom(A₂) ∧ A₁/t ∈ dom(A₂)
- link₄(A₃, P₃) = P₃/t ∈ dom(A₃) ∧ A₁/t ∈ dom(A₃)
Recall Filter functions specify which rights are copied from one entity to another.

- \( f_1(a_2, a_1) = a_1/t \cup c/Rtc \)
- \( f_1(a_3, a_2) = a_2/t \cup c/Rtc \)
- \( f_2(s, a_3) = a_3/t \cup c/Rtc \)
- \( f_3(a_1, c) = p_1/R_{4,1} \)
- \( f_3(a_2, c) = p_2/R_{4,2} \)
- \( f_3(a_3, c) = p_3/R_{4,3} \)
- \( f_4(a_1, p_1) = c/R_{1,1} \cup p_1/R_{2,1} \)
- \( f_4(a_2, p_2) = c/R_{1,2} \cup p_1/R_{2,2} \)
- \( f_4(a_3, p_3) = c/R_{1,3} \cup p_1/R_{2,3} \)
ESPMT Emulating 3-parent creates, Order of Creates

Initially, create agents $A_1, A_2, A_3, S$ and child $C$, what tickets are granted by the create rule?

- Parents $P_1, P_2, P_3$ have no relevant tickets
- $A_1$ has $P_1/Rtc$ (rights $R$ over $P_1$ and $P_1$ is the parent)
- $A_2$ has $P_2/Rtc \cup A_1/tc$
- $A_3$ has $P_3/Rtc \cup A_2/tc$
- $S$ has $S/Rtc \cup C/tc$
- $c$ has $c/R_3t$
Now apply link rules and filter functions

- Propagate Rights to the child via the agents
- Only \( \text{link}_2(S, A_3) \) holds so apply \( f_2(s, a_3) \)
  - \( A_3 \) has \( P_3/Rtc \cup A_2/t \cup A_3/t \cup C/Rtc \)
- This makes \( \text{link}_1(A_3, A_2) \) true so apply \( f_1(a_3, a_2) \)
  - \( A_2 \) has \( P_2/Rtc \cup A_1/t \cup A_2/t \cup C/Rtc \)
- Now \( \text{link}_1(A_2, A_1) \) is true so apply \( f_1(a_2, a_1) \)
  - \( A_2 \) has \( P_2/Rtc \cup A_1/t \cup A_2/t \cup C/Rtc \)
- Now all \( \text{link}_3 \) rules true so apply \( f_3 \)
  - \( C \) has \( C/R_3 \cup P_1/R_{4,1} \cup P_2/R_{4,2} \cup P_3/R_{4,3} \)
- Propagate rights to the the Parents
- Now all \( \text{link}_4 \) rules true, so apply \( f_4 \)
  - \( P_1 \) has \( C/R_{1,1} \cup P_1/R_{2,1} \)
  - \( P_2 \) has \( C/R_{1,2} \cup P_1/R_{2,2} \)
  - \( P_3 \) has \( C/R_{1,3} \cup P_1/R_{2,3} \)
Some correctness properties:

- The emulation gives the appropriate rights to $P_1$, $P_2$, $P_3$ and $C$
- If create of $C$ fails, then $\text{link}_2$ will be false, preventing inappropriate rights transfers to $P_1$, $P_2$ or $P_3$.

Given a 2-parent create, an $n$-parent create can be emulated with a fixed (finite) number of extra types, rights, link rules and filter functions.

- Proof follows from the previous construction.
- Construction can be generalized to emulate a $n$-parent create from an $(n - 1)$-parent create.
- Apply induction to get down to base case 2-parent create.
- Note: The two systems may have different initial states.
ESPM Simulation Using Graphs

Amman, Sandhu and Lipton represent an ACM using a digraph

- A vertex is an entity with a static type assigned at creation.
- Each edge corresponds to a right, edges have a static type assigned at creation.
- Allowed operations are:
  - Initial State Operations - Create the graph in a particular state
  - Node Creation Operations - Add new vertices and edges with those vertices as targets.
  - Edge adding Operations - Add new edges between existing vertices.

- A scheme is a finite state machine that defines a set of node types, edge types, initial state operations, node creation operations and edge adding operations.
- A model is a set of schemes
- Scheme $A$ and scheme $B$ correspond iff the graph containing the state in scheme $A$ is identical to the subgraph in $B$ obtained by removing all nodes and edges in $B$ that don’t have types found in $A$. 
Consider simulating 2-parent emulation of 3-parent create

- First, recall 3-parent create (without 2-parent emulation)
  - Initial state \( P_1, P_2, P_3 \), no edges
  - Create vertex \( C \) of type \( c \) with edges of type \( e \) (drawn as solid lines).

\[ \begin{align*}
P_1 \quad &\quad P_2 \quad &\quad P_3 \\
\end{align*} \]

Initial state (before 3−parent create)

\[ \begin{align*}
P_1 &\quad -\quad P_2 &\quad -\quad P_3 &\quad \text{Add vertex} \ C &\quad \text{and edges of type} \ e \\
&\quad \text{from Parents to child} \\
\end{align*} \]
ESPM Simulation Using Graphs - Child Creation

- Create Agent of child $S$
- Create Agent Child $C$
**ESPM Simulation Using Graphs - Add Edges**

- Use transitive closure of paths to create edges from $P_1, P_2, P_3$ to child.

  - $P_1 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow S \rightarrow C$ to add edge of type $e$ from $P_1$ to $C$
  - $P_2 \rightarrow A_2 \rightarrow A_3 \rightarrow S \rightarrow C$ to add edge of type $e$ from $P_1$ to $C$
  - $P_3 \rightarrow A_3 \rightarrow S \rightarrow C$ to add edge of type $e$ from $P_1$ to $C$
Simulation Fidelity (Equivalence under Simulation)

Consider our example - 2-parent create simulation of 3-parent create

- \( P_1, P_2, P_3, C \) vertices similar (with same type) and edges of type \( e \) similar.
- In 3-parent case doesn’t have the new vertex type \( a \) for \( A_1, A_2, A_3, S \) and new edges of new type \( e' \) found in the 2-parent emulation.

\[ \Gamma \] By definition, 2-parent emulation corresponds to 3-parent create.

Scheme \( A \) simulates schem \( B \) iff both

- For every reachable state \( b \) in scheme \( B \) there is a corresponding state \( a \) in \( A \), and
- For every state \( a \) reachable by \( A \), either the corresponding state \( b \) is reachable by \( B \) or there is a successor \( a' \) reachable by \( A \) that corresponds to a state reachable by \( B \).
Model Expressiveness

Given two models, $MA$ and $MB$ which is more expressive?

- If $MB$ cannot simulate every scheme in $MA$, then $MB$ is at less expressive than $MA$.
- If $MB$ can simulate every scheme in $MA$, then $MB$ is as expressive as $MA$.
  
  ▶ *Note: It is possible that $MB$ may be more expressive than $MA$.*

- If $MA$ is as expressive as $MB$ and $MB$ is as expressive as $MA$ then $MA$ and $MB$ are equivalent.
Consider scheme $A$ in model $M$

- Nodes $X_1, X_2, X_3$
- 2-parent create supported
- No edge adding operations for this particular model.
- Initial condition: $X_1, X_2, X_3$, no edges.

Consider Scheme $B$ in model $N$

- Same as $M$ but has only 1-parent create.

So which model is more expressive?
Consider model $M$, can it express all schemes in model $N$?

- Consider single parent create in $N$ where $X_1$ creates $Y$.
- Can $M$ emulate $N$’s 1-parent create using 2-parent create?
  - Yes, let $X_1 = X_2$ (make them the same node)

Now consider 1-parent create of model $N$, can $N$ emulate $M$?

- Consider 2-parent create where parents $X_1$ and $X_2$ create $Y$.
  - Here we show a specific example, but generally 2-parent creates support nodes having an even in-degree.
- After 2-parent create in $M$, in-degree of child, $Y$, is 2 (an even number).
  - Try $X_1$ creates $Y$ in $N$.
  - However, $N$ lacks edge create operations (as per the given).
  - So edge from $X_2$ to $Y$ cannot be constructed.
  - Using $X_2$ as parent gives similar symmetric result.

- So $M$ (with 2-parent create) is more expressive than $N$ (with 1-parent create).

So which model is more expressive?

- $M$ (with 2-parent create) is more expressive than $N$ with 1-parent create.
Some Theorems

Monotonic single-parent create models are less expressive than monotonic multi-parent create models.

ESPM is more expressive than SPM

- ESPM monotonic with multi-parent creates.
- SPM monotonic but has only single-parent creates.
Typed Access Matrix Model (TAM)

TAM extends ACM by introducing a set of types, $T$.

- $TS$ is the set of subject types.
- Objects have a set of types (the book gives no notation) $T - TS$.

The protection state in TAM is $(S, O, \tau, A)$

- $S$ is the set of subjects.
- $O$ is the set of objects.
- $\tau : O \rightarrow T$ is the type function, specifying the type of each object.
- $A$ is the access control matrix.

Operations in TAM like in ACM except create operations are augmented with types.
Create operations in TAM

CreateSubject\((s, ts)\).

- Create Subject \(s\) of type \(ts\).
- Precondition: \(s \not\in S\)
- PostConditions: \(S' = S \cup \{s\}, O' = O \cup \{s\}, \forall y \in O'[\tau'(y) = \tau(y)], \tau'(s) = ts, \forall y \in O'[a'[s, y] = \emptyset], (\forall x \in S')[a'[x, s] = \emptyset]\)

CreateObject\((o, to)\).

- Create Object \(o\) of type \(to\).
- Precondition: \(o \not\in O\)
- PostConditions: \(S' = S, O' = O \cup \{o\}, \forall y \in O'[\tau'(y) = \tau(y)], \tau'(o) = to, \forall x \in S'[a'[x, o] = \emptyset], (\forall x \in S')(\forall y \in O)[a'[x, y] = a[x, y]]\)
Monotonic Typed Access Matrix Model (MTAM) is TAM without delete, DestroySubject and DestroyObject primitive operations.

Consider create command $\alpha(x_1 : t_1, \ldots, x_k, t_k)$ where $x_1, \ldots, x_k \in O$ and $\tau(x_i) = t_i$ for $1 \leq i \leq k$.

- $t_i$ is a child type in $\alpha(x_1 : t_1, \ldots, x_k, t_k)$ if any create subject $x_i$ of type $t_i$ or create object $x_i$ of type $t_i$ occurs in $\alpha$.
- $t_i$ is a parent type otherwise.

In spite of this wording, it is possible for a type to be both a parent type and child type for the same operation.

An example:

```plaintext
command foo (s_1:u, s_2:u, s_3:v, s_4:w, o:O) {
  create subject s_2 of type u;
  create subject s_3 of type v;
}
```

$s_2$ create means $u$ is a child type and $s_3$ create means $v$ is a child type.

$s_1$ not created, so $u$ is also a parent type.

$w$ and $O$ are also parent types.
Consider the command \(\text{havoc}(s_1:u, s_2:u, o_1:v, o_2:v, o_3:w, o_4:w)\)

\[
\text{command havoc}(s_1:u, s_2:u, o_1:v, o_2:v, o_3:w, o_4:w)\{
    \text{create subject } s_1 \text{ of type } u;
    \text{create object } o_1 \text{ of type } v;
    \text{create object } o_3 \text{ of type } w
    \text{enter } r \text{ into } a[s_2, s_1];
    \text{enter } r \text{ into } a[s_2, o_2];
    \text{enter } r \text{ into } a[s_2, o_4];
\}
\]

- \(u, v, w\) are child types (from \(s_1, o_1, o_3\)).
- \(u, v, w\) are also parent types (from \(s_2, o_2, o_4\)).
Creation Graphs in MTAM

Creation Graph has edges from each parent type to each child type

- An MTAM is cyclic if its creation graph contains a cycle, otherwise it is acyclic.
- Consider the creation graph from our example

![Creation Graph Diagram]

CSI 424/524, William A. Maniatty, Dept. of Computer Science, University at Albany

76
MTAM Theorems (By Sandhu)

Safety is decidable for systems with acyclic MTAM schemes (creation graphs).

Safety is $NP$-hard for systems with acyclic MTAM schemes (creation graphs)

Safety is decidable in ternary MTAM in time polynomial to the size of the initial ACM.

- *Ternary* means all commands have no more than 3 parameters.
- Is equivalent in expressive power to MTAM.
Conclusions

In general Safety problem is undecidable.
Some limited scope variants of safety problem are decidable.
Types are useful for safety problem analysis.