Distributed Operating Systems

Security - Foundations, Covert Channels, Noninterference

Marcus Völp / Hermann Härtig

2010

TECHNISCHE UNIVERSITÄT DRESDEN

Purpose of this Lecture

- Assurance
  - Can you trust the system you intend to use
    - to protect your private / valuable data?
    - to grant only those programs access to your data that you trust?
    - to grant your programs access to data when they need it?
- Formal methods
  - as a precise description of system behavior
  - as a tool to reason about security properties

What makes you believe that your system is secure

- Trust in the developer / company
  - I've built it so I know whats wrong!
  - I trust the guys at <add your favorite company here> (at least I can sue them)!
- Quality Assurance Processes
  - ISO 9000
    - There is a QA team that runs tests on the SW of the development team; QA- and SW teams are disjoint
- Security Evaluation
  - Common Criteria
  - DO 178b (Airplanes)
  - GISA (BSI) IT Security Evaluation Critera

(old '89 proposal for CC)

What makes you believe that your system is secure

- because the system is described in a way that is
  - precise,
  - sufficiently small to be captured in its entirety and
  - easy to understand
    - Abstract Mathematical Model
- because all security claims of the system follow from this description
    - Mathematical Proofs
- because the description and the actual system correspond
    - Refinement Proofs

Security Evaluation

- Common Criteria (EAL 7)
  - Formal top level specification
  - Informal (through tests) correspondence of source code to abstract specification

- GISA IT Security Evaluation Criteria (Q7)
  (a proposal for CC-EAL 7 - 1st version from '89)
    - “The machine language of the processor used shall to a great extent be formally defined.”
    - “The consistency between the lowest specification level and the source code shall be formally verified.”
    - “The source code will be examined for the existence of covert channels, applying formal methods. It will be checked that all covert channels detected which cannot be eliminated are documented. [...]”

Overview

- Introduction
- Security Policies
- Policy Enforcement
- Decidability of Leakage
- Take Grant Protection Model
- Covert Channels
- Compiler-Based Information Flow Control

Security Policies

- Example:
  - Only the owner of a file and root can have write privileges to this file.
- Security Policy
  - Defines what is allowed / secure and what is not allowed / unsecure
- Secure System
  - System that enforces a security policy

Notation

- iff = if and only if
- Definition :=

- Sets: S, O, R, L
- Elements: s, o, r, l

- States: $\sigma \in \Sigma$
- Subject: $s \in S$ ●
- Object: $o \in O$ ○
- Entity: $e \in E$ with $E = S \cup O$ ⊘

- Right: $r \in R$
- Access rights:
  - $S \times O \rightarrow \wp(R)$
  - $R(s,o)$

- State Transition (command c):

$$\sigma \xrightarrow{c} \sigma'$$

with result state: σ'

- $$\sigma \xrightarrow{u.c} \sigma'$$

if u is the current user in σ that invokes c

- Secrecy / Integrity Levels: $l \in L$
- Dominates relation:

$$l_1 \leq l_2$$

Information flow: from l_1 to l_2

$$l_1 \longrightarrow l_2$$

no IF: $l_1 \sim/\sim> l_2$

Security Policies – A first abstract system Model

- Example:
  - No user except the owner of a file and root can have write privileges to this file.
- A first abstract **system** model:

  (Abstracts from real-life system; keeps necessary information to reason about the above example)

  - State: $\sigma \in \Sigma$
    - **Users:** set of all possible users
      Files: set of all possible files:
    - $\Sigma = \{(U_{life}, F_{life}, \text{owner}, \text{rights}, u_{current})\}$
      - $U_{life} \subseteq \text{Users}$, $F_{life} \subseteq \text{Files}$, $u_{current} \in U_{life}$,
        owner: $F_{life} \text{->} U_{life}$, rights: $U_{life} \times F_{life} \rightarrow \wp(R)$
    - σ = ({root, myself, hermann}, {foo.txt, bar.txt}, root,
      {(foo.txt, myself), (bar.txt, hermann)}, {(root, foo.txt, {rw})})

Security Policies – A first abstract system Model

- A first abstract **system** model:
  - State transitions:
    $c \in C$; $C :=$ {read(file), write(file), create(user), delete(file),chmod(u,f,R),...}
    - σ = ({root, myself, hermann}, {foo.txt, bar.txt}, root, {(foo.txt, myself), (bar.txt, hermann)}, {(root, foo.txt, {rw})})
    - $\sigma \xrightarrow{c} \sigma'$
    - Example:

$$\sigma \xrightarrow{\text{read(bar.txt)}} \sigma' \text{ with } \sigma' := \sigma$$

$$\sigma \xrightarrow{\text{delete(bar.txt)}} \sigma' \text{ with}$$

σ' := ({root, myself, hermann}, {foo.txt, ~~bar.txt~~}, root, {(foo.txt, myself), ~~(bar.txt, hermann)~~}, {(root, foo.txt, {rw})})

if $u_{current}$ = root v owner(bar.txt, $u_{current}$)

$\sigma' := \sigma$ otherwise

Security Policies – A first abstract system Model

- A first abstract **system** model:
  - Initial State: σ_0
  - Reachable States: $\Sigma_{0,C}$
    - Set $\Sigma_{0,C}$ of states that are reachable from σ_0 through a sequence of transitions c in C
    - $\sigma_0 \xrightarrow{\quad\quad} {}^{*} \sigma$ iff $\sigma \in \Sigma_{0,C}$
    - Example: (if we require that the creator of a file becomes its owner)
      σ' := ({root, myself}, {foo.txt, bar.txt, orphan.txt}, root, {(foo.txt, myself), (bar.txt, hermann)}, {})
      - σ' is a state (i.e., $\sigma' \in \Sigma$), however σ' is not reachable
  - System := (Σ, C, σ_0)

Security Policies – A first abstract system Model

- Example Policy:
  - No user except the owner of a file and root can have write privileges to this file.
- Does the system (Σ, C, σ_0) enforce the example policy P?

  - P(σ) := $\forall$ u,f. w $\in$ rights(u,f) => owner(f,u) v u = root

$$\sigma \xrightarrow{\text{myself.chmod(u, foo.txt, \{w\})}} \sigma'$$

  - without further constraints: u = hermann => $\neg$ P(σ')

    => the system is insecure

  - but, the system is secure if we replace chmod with chmod':

    chmod'(u,f,R)(σ) := if (u = root v owner(file, u)) chmod(u,f,R)(σ) else σ

Security Policies - Definition

- Definition (Bishop – Computer Security Art and Science):
  - A *security policy P* is a statement that partitions the states (Σ) of a system into a set of authorized (or secure) states ($\Sigma_{sec} = \{\sigma \mid P(\sigma)\}$) and a set of unauthorized (or nonsecure) states.
  - A *secure system* is a system that starts in an authorized state and that cannot enter an unauthorized state.

all reachable states must be secure: $\Sigma_{0,C} \subseteq \Sigma_{sec}$

Introduction: Confidentiality, Integrity, Availability

- Confidentiality:
  - Prevent unauthorized disclosure of information
  - *Definition:*

    *Information I is confidential with respect to set of entities X if no member of X **can obtain information** about I.*

    - *Example: My EC-Card Pin is XXXX*

Introduction: Confidentiality, Integrity, Availability

- Integrity:
  - Correctness of data and information
  - *Definition 1:*

    Information is current, correct and complete.
    - *prevent damage*
  - *Definition 2: (fundamentally different to Def 1)*

    *Either information is current, correct, and complete (Def 1.), or it is possible to **detect** that these properties do not hold.*
    - *detect damage*
- *Example: balance of my bank account*
- Recoverability:
  - *Definition:*

    Information that has been damaged can be recovered eventually.

Introduction: Confidentiality, Integrity, Availability

- Availability:
  - Accessibility of information and services
  - *__Definition 1:__*

    Resource I is available with respect to X if all members of X can access I.
  - *In practice, availability has also quantitative aspects:*
    - *real-time systems:*
      - *I is available within t clock ticks*
      - *I is available t clock ticks after a certain event*
    - *fault-tolerant systems:*
      - *In 1 - 10^{-6} % of all cases I is available to X*

Security Policies

- Classification
  - Concern:

Confidentiality	e.g., Bell La Padula (Document Mgmt)
Integrity	e.g., Biba, (Inventory System)
Availability	
Hybrid	e.g., Chinese Wall, (Clinical Information System)

  - Types of Access Controls
    - discretionary (identity based)
      - A user can configure the access control mechanism to allow or deny access to an object (it owns).
    - mandatory (rule based)
      - A system-wide mechanism controls access to objects based on a set of rules; individual users cannot alter these rules.

Security Policies

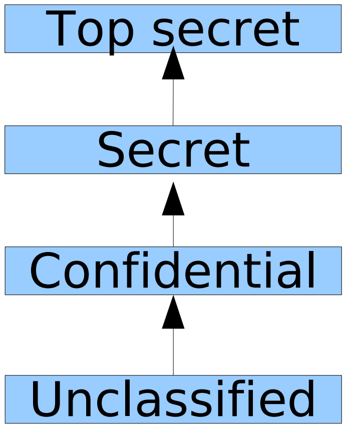
- Types of Access Controls
  - discretionary (identity based)
    - Example:
    - A user is allowed to create new entities; it becomes the owner of these entities.
    - A user can change the access rights and the ownership of the files it owns.
  - mandatory (rule based)
    - Example:

      Only system administrators are allowed to create new users.

=> A user attempt to create a new user will fail although users can create new entities.

Bell-LaPadula Model '73 (simple version)

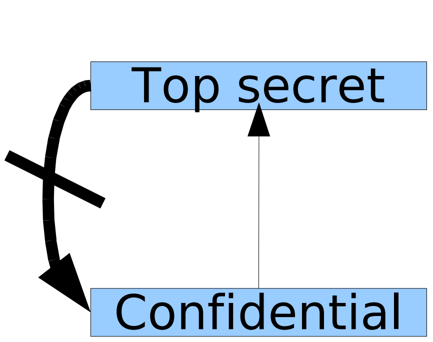
- Confidentiality Policy
- Totally ordered (by ≤) set of secrecy levels (L)
  - Higher secrecy level
    => more sensitive information
    => greater need to keep it confidential
  - Each subject has a **security clearance**
    (dom(s) ∈ L)
  - Each object has a **security classification**
    (dom(o) ∈ L)

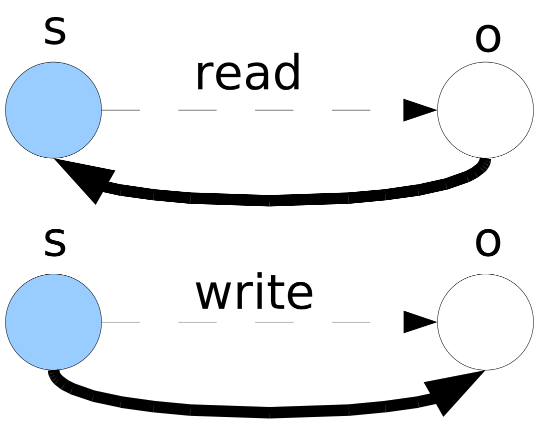


- **Bell-LaPadula and the following security policies can be described as: (L, dom, ≤)**

Bell-LaPadula Model (simple version)

- Security Policy: (L, dom, ≤)

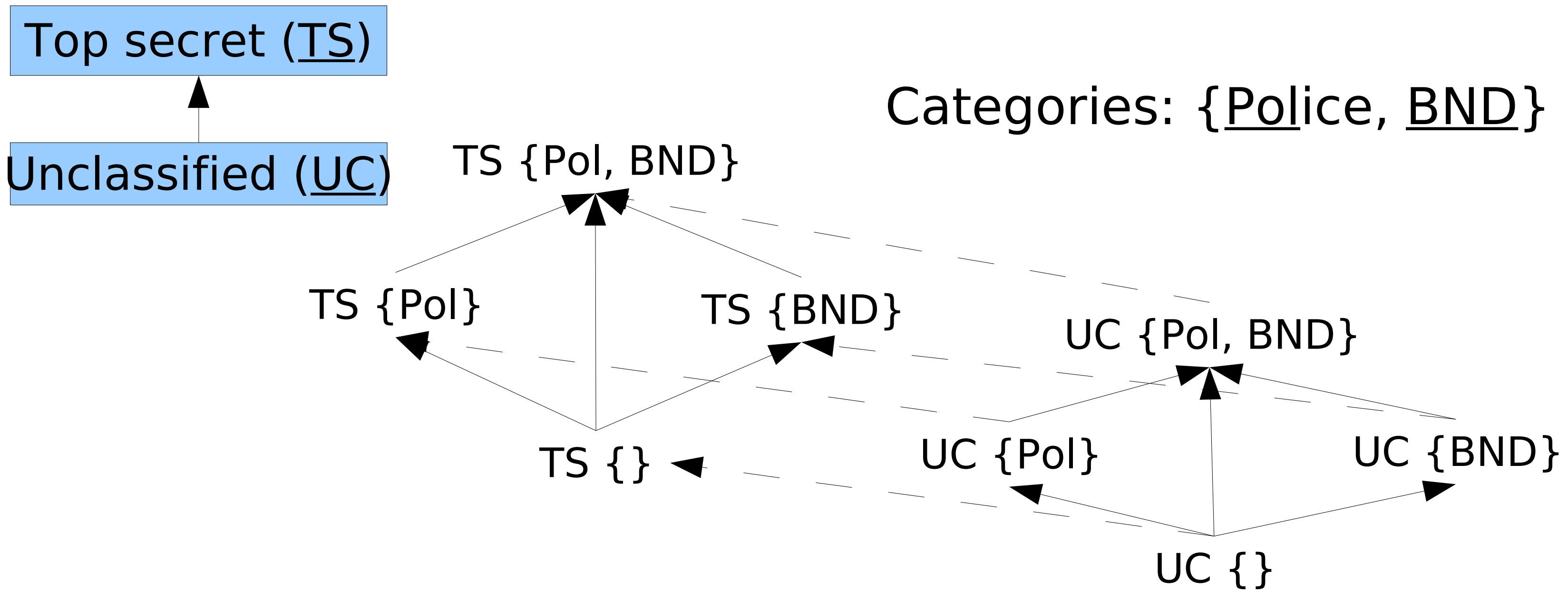




  - Simple Security Condition
    - a subject s can only read lower or equally classified objects o
    - s can read o iff dom(o) ≤ dom(s)
  - *-Property
    - a subject s can only write higher or equally classified objects o
    - S can write o iff dom(s) ≤ dom(o)

Bell-LaPadula Model (MLS)

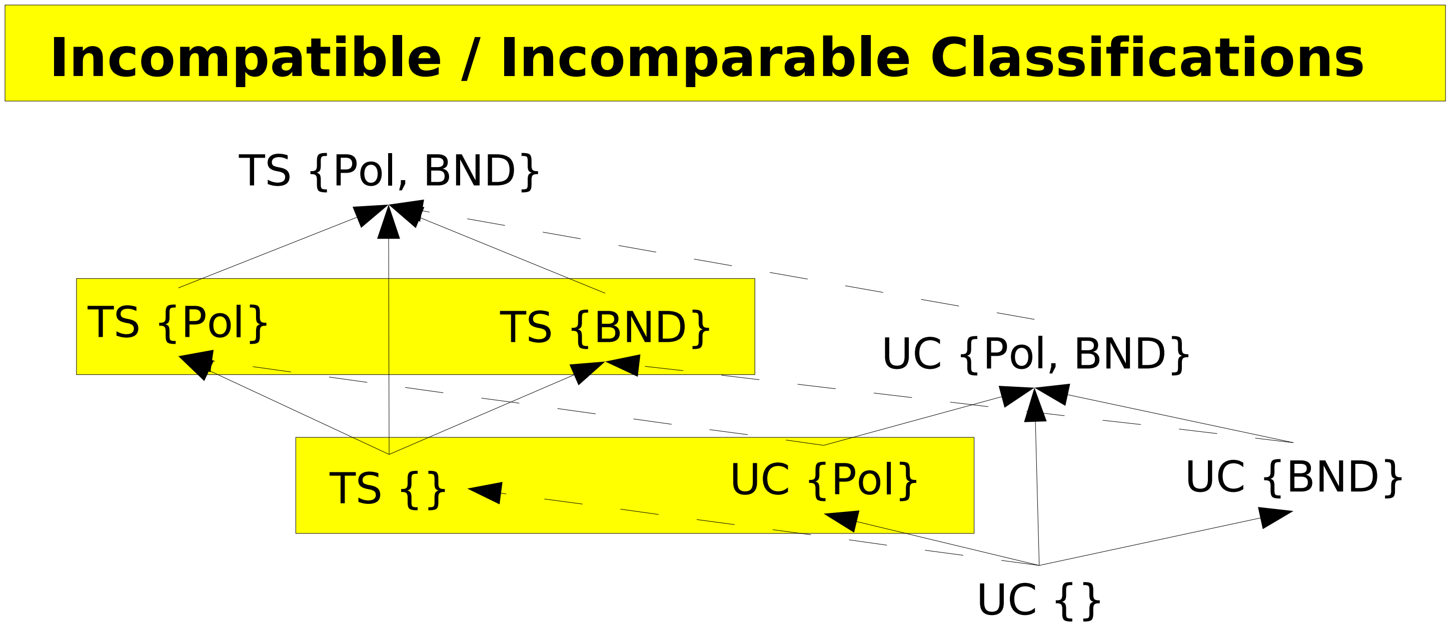
- Security clearance comprised of hierarchical level and set of nonhierarchical categories
- Partial order (≤); (L, ≤) form a lattice



- German law (Bundesverfassungsschutzgesetz §17 - §26):
  In general, no information exchange between BND and Police.

Bell-LaPadula Model (MLS)

- Security clearance comprised of hierarchical level <u>and</u> set of nonhierarchical categories
- Partial order (≤); (L, ≤) form a lattice



Biba '77: Integrity Policies

(to prevent damage on integer data (Def. 1))

- Strict Integrity Policy (Biba Model)
  - Set of hierarchical integrity levels L
  - Integrity policy as triple (L, dom, ≤)

  - s can read o iff dom(s) ≤ dom(o)
  - s can write o iff dom(o) ≤ dom(s)

  - Strict Integrity Policy is dual to MLS

  - It **prevents** subjects from reading less integer objects
  - Alternative: allow subjects to read less integer data but prevent the consequences such a read may have on other objects => Low Water Mark.

Biba: Integrity Policies

- Low Water Mark
  - s can write to o if and only if dom(o) ≤ dom(s)
  - If s reads o then **dom’(s) = min(dom(s), dom(o))**
  - Problem: label creep
    - decrease of subjects integrity level and thus the integrity level of the subject's results.
    - (dual for confidentiality policies:
      increase object's confidentiality level)

D.Denning '76: Lattice Model (+ R. Sandhu '93)

- Most security policies can be expressed by the triple (L, dom, ≤) where (L, ≤) is a lattice.
- Confidentiality and integrity are dual properties; they can be combined into a single lattice, which describes the flow of information between the classified objects and subjects.

Confidentiality: $l_{conf} \leq h_{conf}$

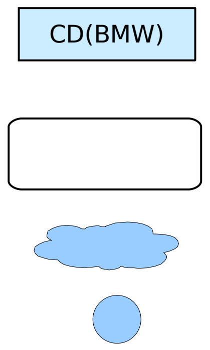
Integrity: $h_{int} \leq l_{int}$

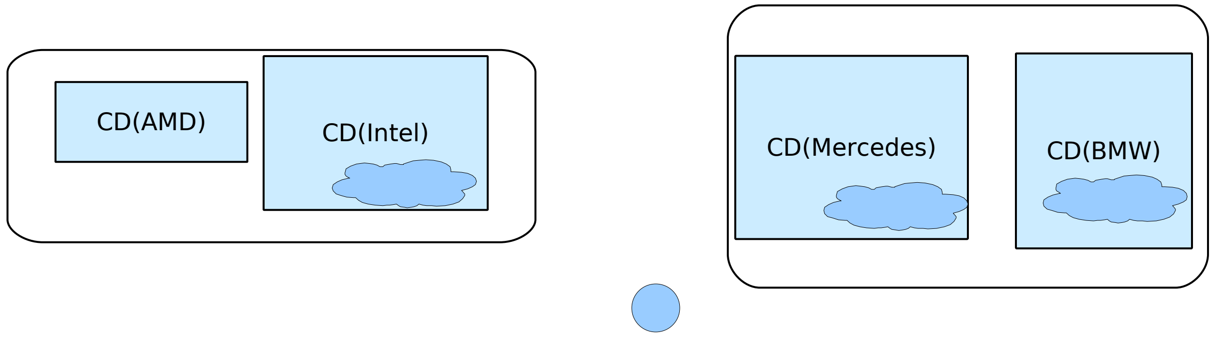
$$\begin{tikzcd}
 & h_{conf}, l_{int} & \\
l_{conf}, l_{int} \arrow[ur] & & h_{conf}, h_{int} \arrow[ul] \\
 & l_{conf}, h_{int} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Chinese Wall (Brewer '89)

- Conflict of Interest
  - E.g., British law for stock exchange
    - Trader must not represents two competitors. Otherwise, the trader could help one to gain an advantage at the expense of the other.

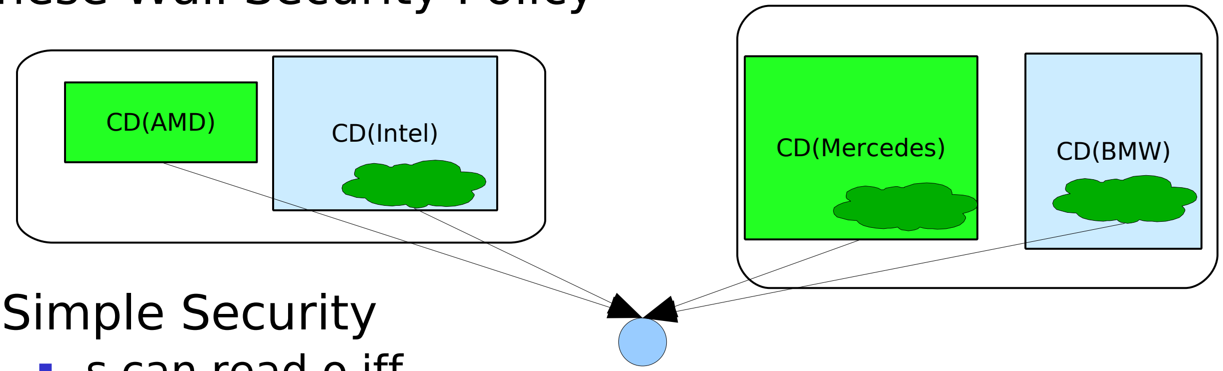
  - Company Dataset (CD):
    set of objects (files) related to a single company
  - Conflict of Interest Class (COI):
    datasets of companies in competition
  - Sanitized Objects: objects cleared to the public
  - Subjects: s *(the traders, not the companies)*





Chinese Wall

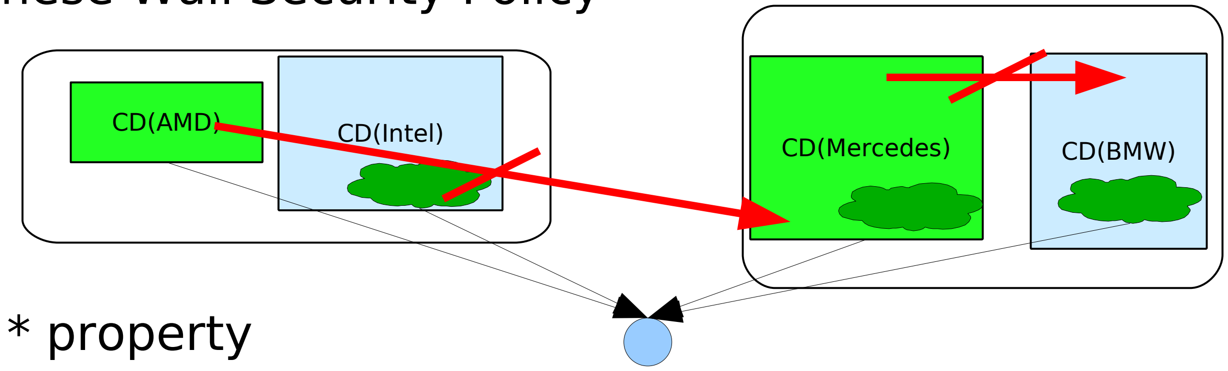
- Chinese Wall Security Policy



  - Simple Security
    - s can read o iff
      - s has already access to an object of this company:
        ∃ o’ accessed by s with CD(o’) = CD(o),
    - <u>or</u>
      - no object o' that s has read is in conflict to o:
        ∀ o’ read by s => COI(o’) ≠ COI(o)
    - <u>or</u>
      - o is sanitized

Chinese Wall

- Chinese Wall Security Policy

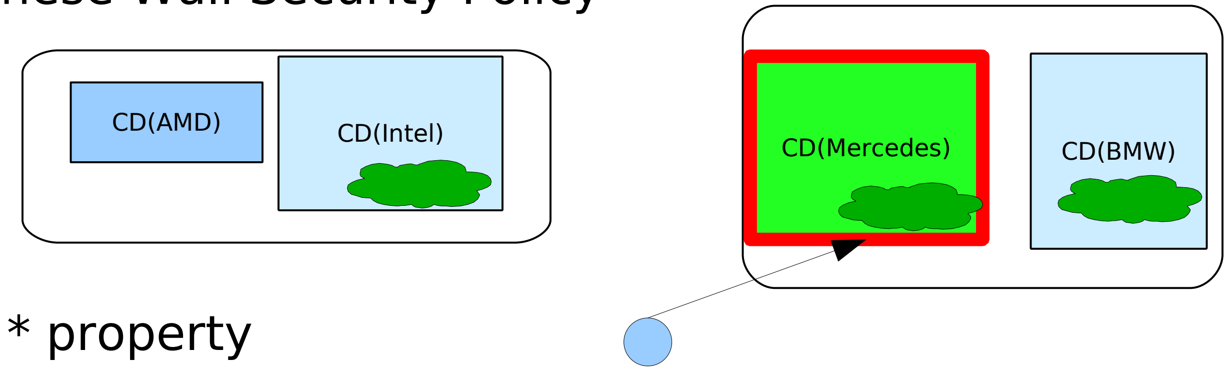


- * property
  - s can write o iff
    - s can read o,
  - <u>and</u>
    - If s can read an **unsanitized** object o', then o' must belong to the same company as o:
      ∀ o’. s can read o' => CD(o’) = CD(o)

- That is, s must not leak data to another company unless this release is explicitly allowed (by sanitizing the data).

Chinese Wall

- Chinese Wall Security Policy

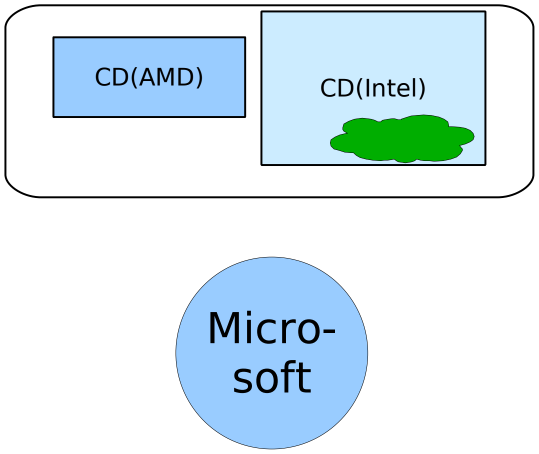


  - * property
    - s can write o iff
      - s can read o,
    - <u>and</u>
      - If s can read an **unsanitized** object o', then o' must belong to the same company as o:
        ∀ o’. s can read o' => CD(o’) = CD(o)

  - That is, s must not leak data to another company unless this release is explicitly allowed (by sanitizing the data).

Chinese Wall

- Chinese Wall Security Policy



- NDAs: a real-life example for OS developers
  - MS needs early access to hardware to adjust Windows
  - Intel and AMD need to protect their IP from respective competitor

  - Chinese Wall in Practice:
    - 1 Group of MS Developers with Intel
    - 1 Group of MS Developers with AMD
    - NO information exchange between these groups

Overview

- Introduction
- Security Policies
- Policy Enforcement
- Decidability of Leakage
- Take Grant Protection Model
- Covert Channels
- Compiler-Based Information Flow Control

Policy Enforcement Mechanisms:

- Access Control Matrix (ACM):
  - Subjects S, Objects O, Entities E = S u O, Rights R
  - Matrix: S x E x R
    - any operation c from s on o (or s') checks the respective cell R(s,o) of the ACM for sufficient rights for this operation c.

	o1	o2	s1	s2
s1	rd,wr	rd	rd,wr	rd
s2	rd,wr	-	wr	rd,wr

  - Operations: C
    - read entity, write entity
    - create subject, create object
    - destroy subject, destroy object
    - **enter right r into cell R(s,o)**, **delete right r from cell R(s,o)**

Policy Enforcement Mechanisms:

- Access Control List:
  - Each entity has a list of tuples: Subjects S x Rights R
  - e.g., foo.txt: (MV, {rd,wr}), (root, {rd})

  - Abbreviations:
    - Owner, Groups: Unix, AIX (e.g., [user;group;all])
    - Wildcards: foo.txt: (sysadmin_*, {rd,wr})

  - Conflicts:
    - two opposing rights in ACL: u – r; g + r
      - order of occurrence in ACL: u – r; g + r => access
        (e.g., Cisco Router) g + r; u – r => denied
      - deny rule has precedence over allow rule (e.g., AIX)

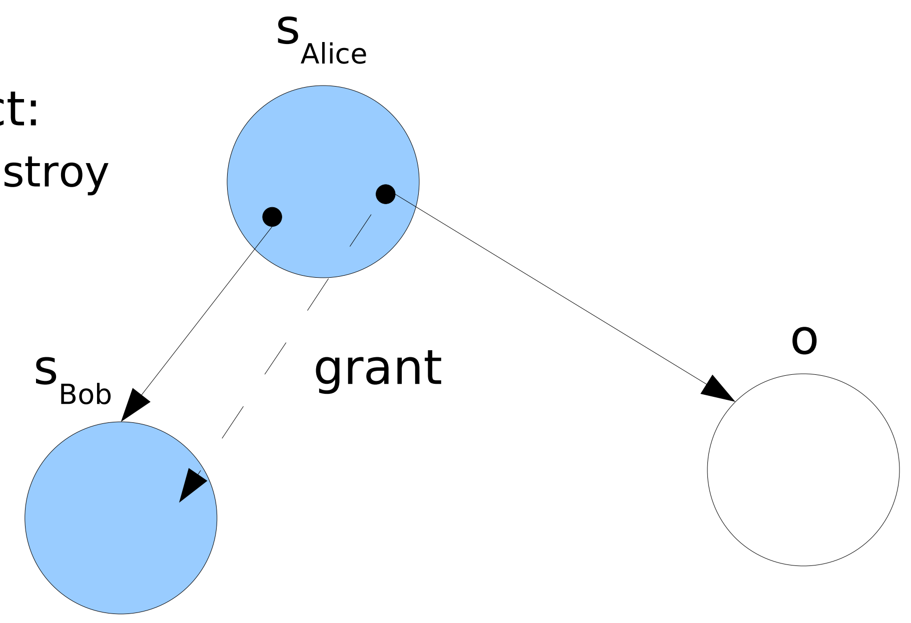
Policy Enforcement Mechanisms:

- Problem: Who is allowed to modify the ACM / ACLs?
  - Ownership: foo.txt: (MV, {rd,wr,own}), (HH, {rd})
  - Principle of Attenuation:
    (in German: Abschwächung, Verminderung)
    - A subject s must not give away rights it does not possess!
      - In principle, cannot be enforced with above ACM operations: any subject i can invoke **enter r into R(s,o)**
      - Solution: replace **enter r into R(s,o)** with:
        - **i.grant r into R(s,o)** :=
          if r $\in$ R(i,o) then enter r into R(s,o)

      (Notation: s.c = the command c invoked by subject s)

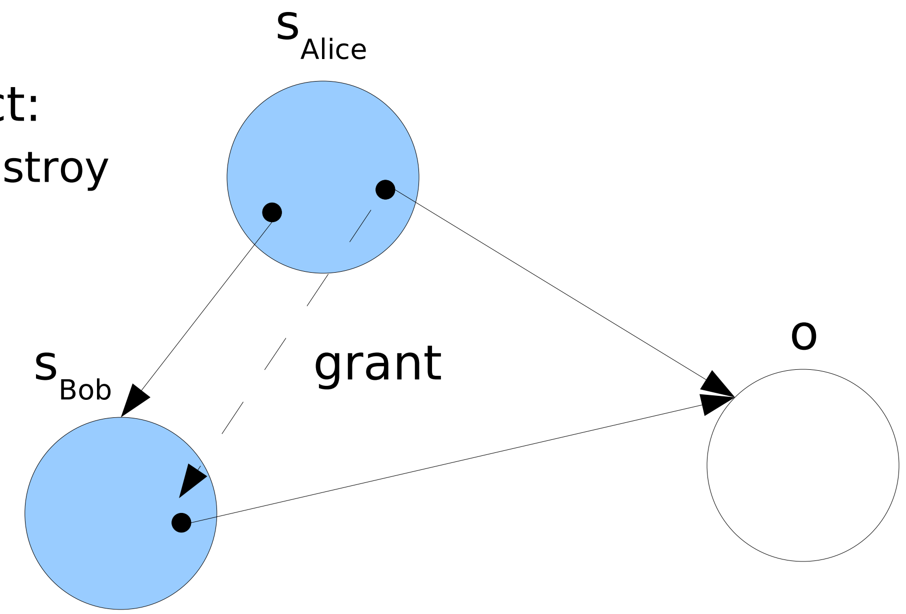
Policy Enforcement Mechanisms:

- Capabilities:
  - Capability = unforgeable token (e,R)
    - with e ∈ Entity, R ⊆ Rights
  - Possession of a Capability is necessary and sufficient to access the referenced entity.
  - Operations
    - on the referenced object:
      - read, write, create, destroy
    - on the capability itself:
      - take, grant
      - diminish, remove



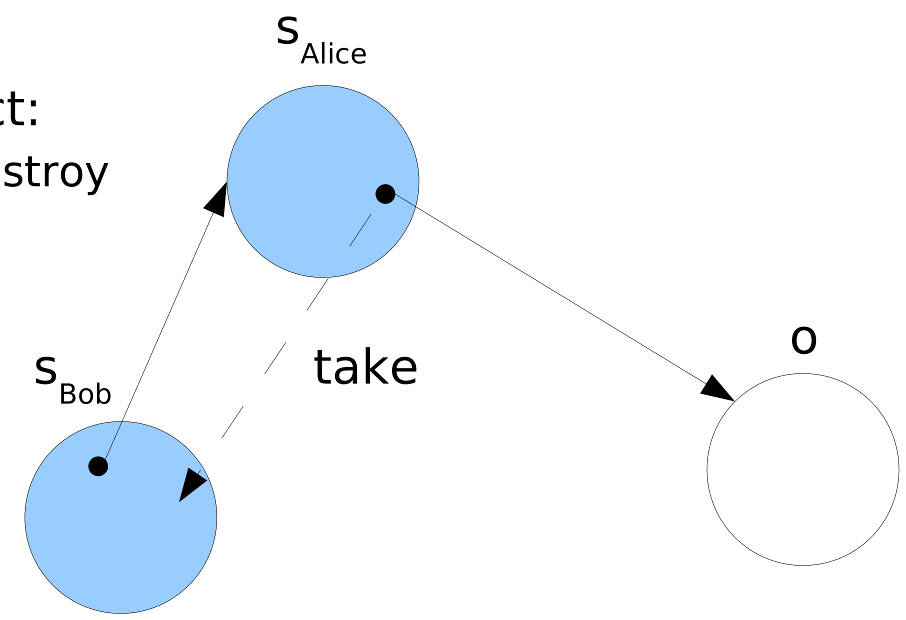
Policy Enforcement Mechanisms:

- Capabilities:
  - Capability = unforgeable token (e,R)
    - with $e \in$ Entity, $R \subseteq$ Rights
  - Possession of a Capability is necessary and sufficient to access the referenced entity.
  - Operations
    - on the referenced object:
      - read, write, create, destroy
    - on the capability itself:
      - take, grant
      - diminish, remove



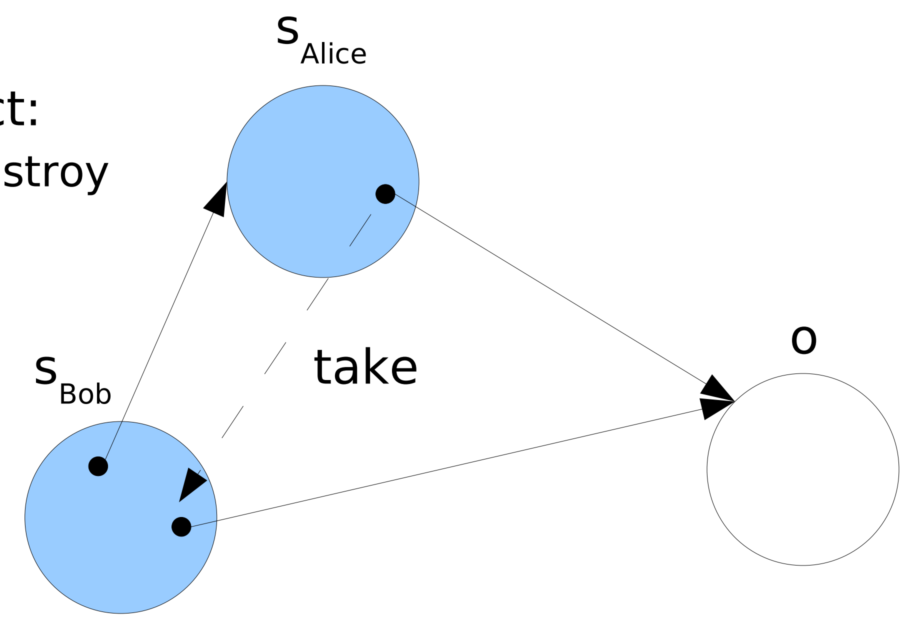
Policy Enforcement Mechanisms:

- Capabilities:
  - Capability = unforgeable token (e,R)
    - with $e \in$ Entity, $R \subseteq$ Rights
  - Possession of a Capability is necessary and sufficient to access the referenced entity.
  - Operations
    - on the referenced object:
      - read, write, create, destroy
    - on the capability itself:
      - take, grant
      - diminish, remove



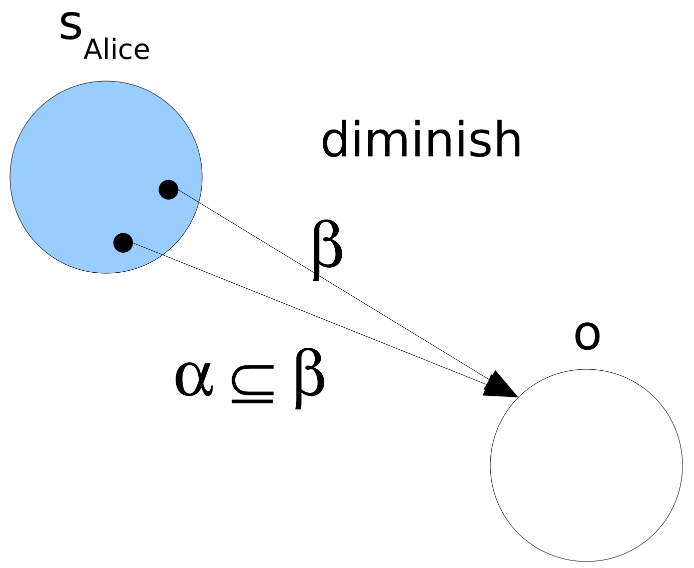
Policy Enforcement Mechanisms:

- Capabilities:
  - Capability = unforgeable token (e,R)
    - with e ∈ Entity, R ⊆ Rights
  - Possession of a Capability is necessary and sufficient to access the referenced entity.
  - Operations
    - on the referenced object:
      - read, write, create, destroy
    - on the capability itself:
      - take, grant
      - diminish, remove



Policy Enforcement Mechanisms:

- Capabilities:
  - Capability = unforgeable token (e,R)
    - with e $\in$ Entity, R $\subseteq$ Rights
  - Possession of a Capability is necessary and sufficient to access the referenced entity.
  - Operations
    - on the referenced object:
      - read, write, create, destroy
    - on the capability itself:
      - take, grant
      - diminish, remove



Policy Enforcement Mechanisms:

- Capabilities:
  - Implementation:
    - Software: OS protected segment / memory page
    - Hardware: Cambridge CAP / TLB
    - Cryptography: Amoeba
  - Problems:
    - How to control the propagation of capabilities?
    - How to revoke capabilities?

Capability Propagation

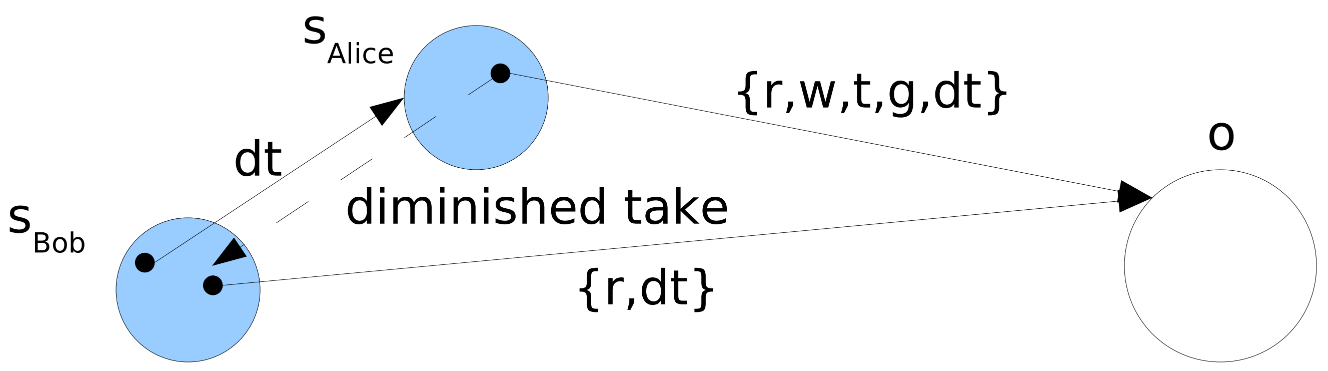
- Controlling Propagation
  - Dual to controlling modification of ACM / ACL

  - Permissions on channel capability:
    - take-permission (t), grant-permission (g)
  - Copy permission on the to be transferred capability

  - Right-diminishing channels: (an extension of TG)

Capability Propagation

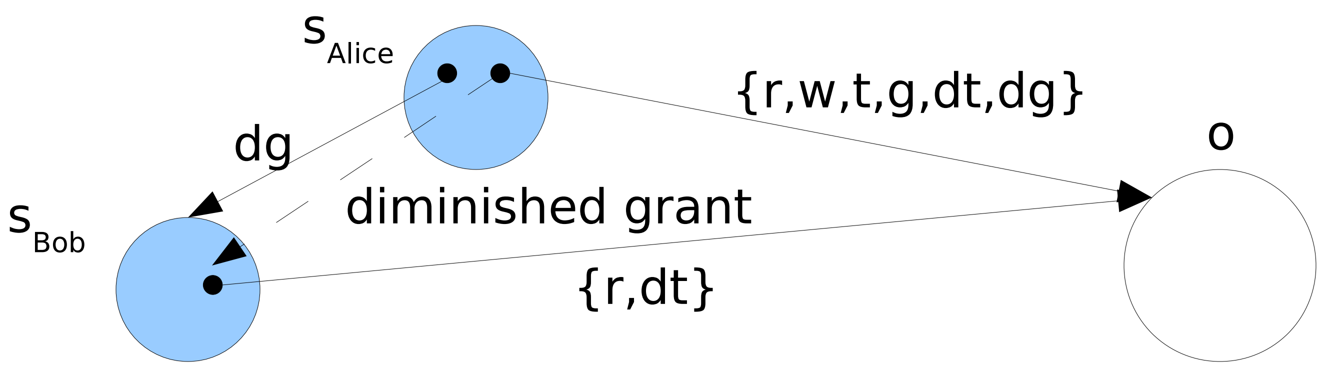
- Controlling Propagation
  - Right-diminishing channels: (an extension of TG)
    - s may take from s' but the caps taken are diminished
      - diminished-take perm. (dt) on channel
      - diminished take (s,c) := diminish(take(s,c), {w,t,g,dg})
      - Can be used to ensure that
        s can only ever **receive** information from s'



Capability Propagation

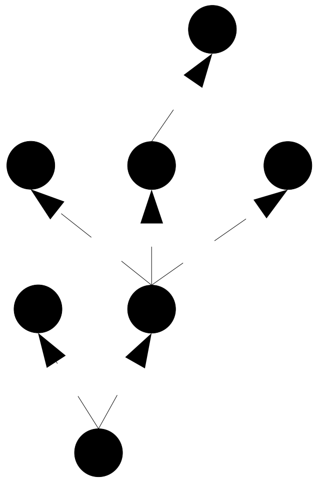
- Controlling Propagation
  - Right-diminishing channels: (an extension of TG)
    - s may grant to s' but the caps granted are diminished
      - Diminished-grant perm. (dg) on channel
      - Diminished grant (s,c) := diminish(grant(s,c), {w,t,g,dg})
      - Can be used to ensure that

        s can only ever **send** information to s'

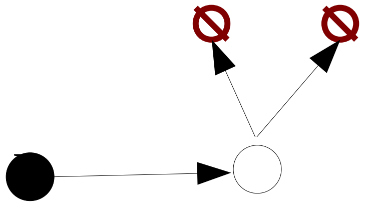


Capability Revocation

- Find and invalidate all direct and indirect copies

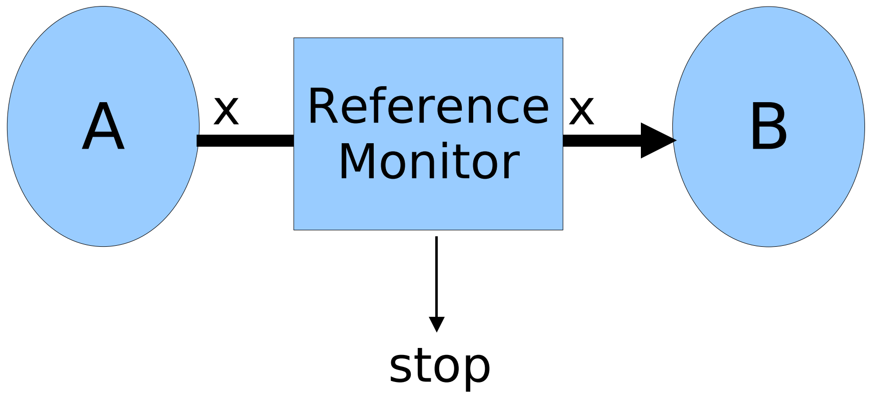


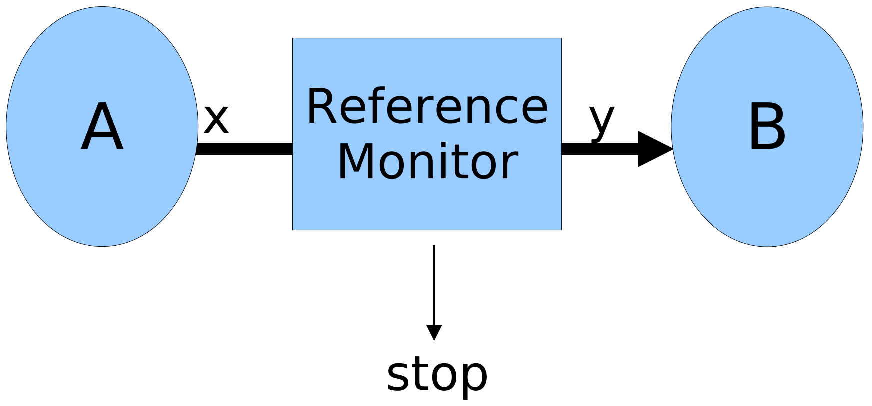
- Indirection Object:
    - Stores capabilities
    - Allows stored caps to be used but not to be taken out
    - Revoke by destruction of indirection object



Policy Enforcement Mechanisms

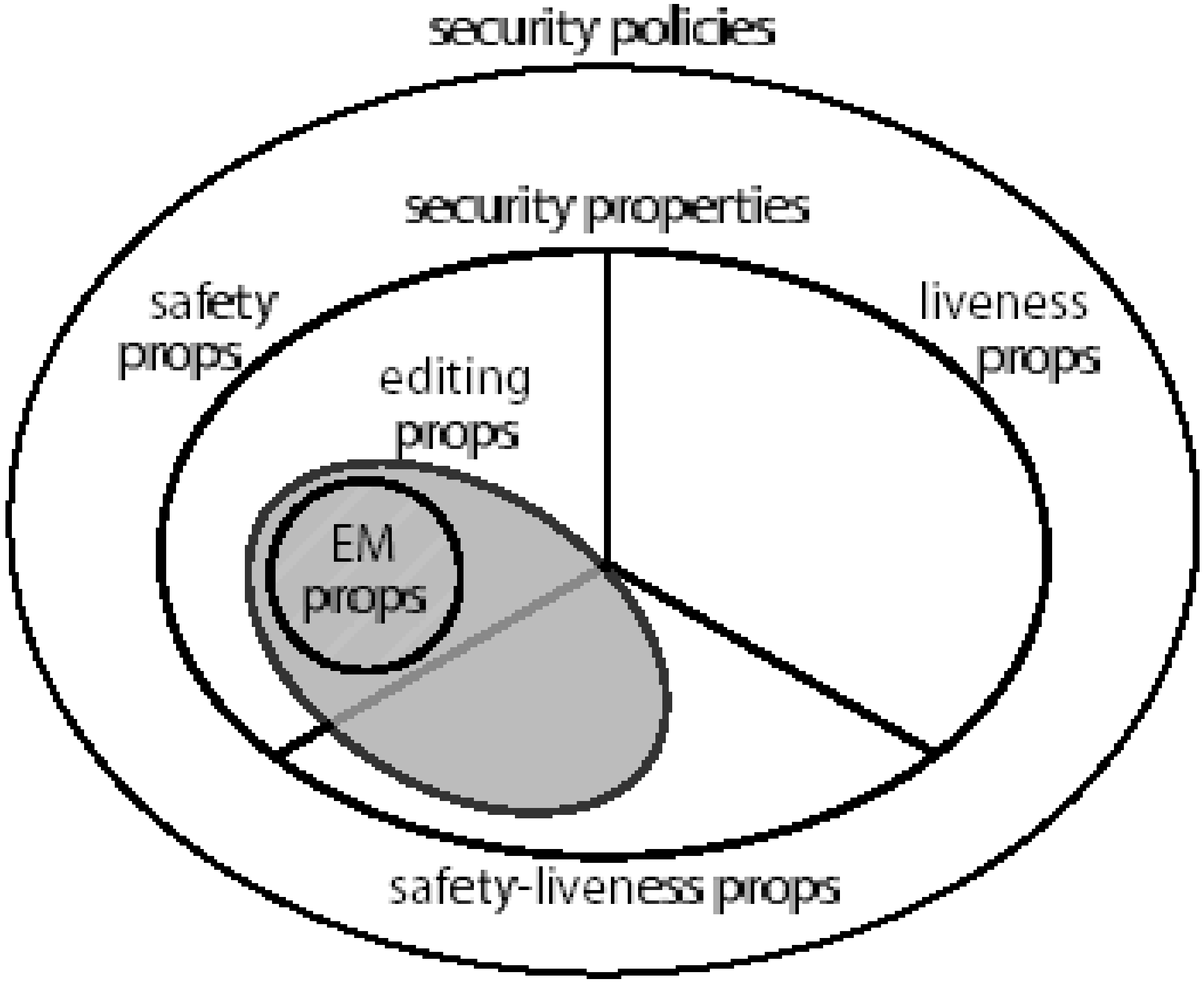
- Reference Monitor:
  EM: suppress, pass Edit



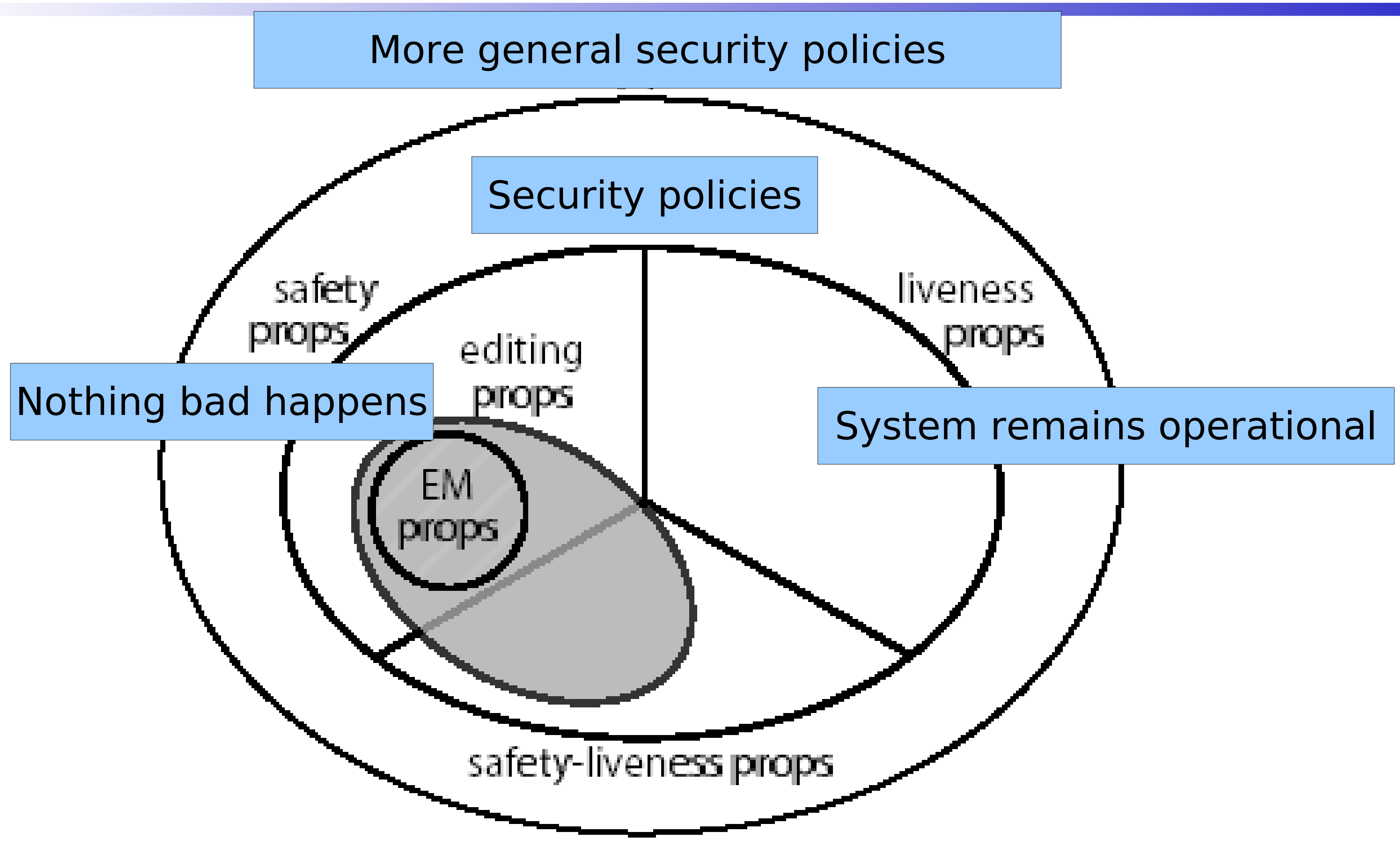


- Schneider [98] / Bauer [02]:
  Which security policies are enforceable by reference monitors that are modeled as:
  - EM automata
  - Edit automata
- **!!! results are based on a different system model !!!**

(More) Enforceable Security Policies

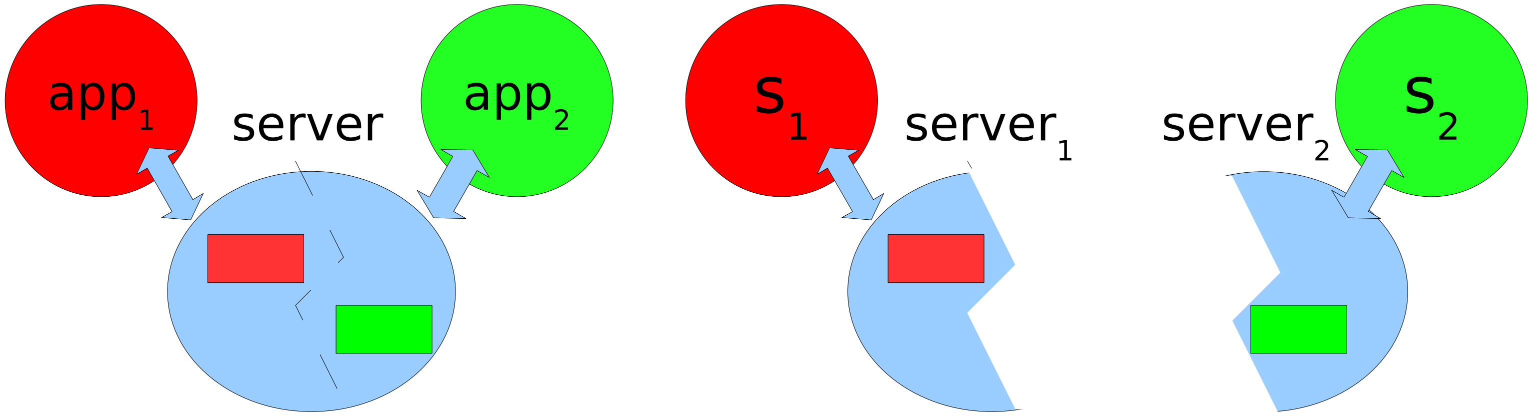


(More) Enforceable Security Policies



Policy Enforcement Mechanisms

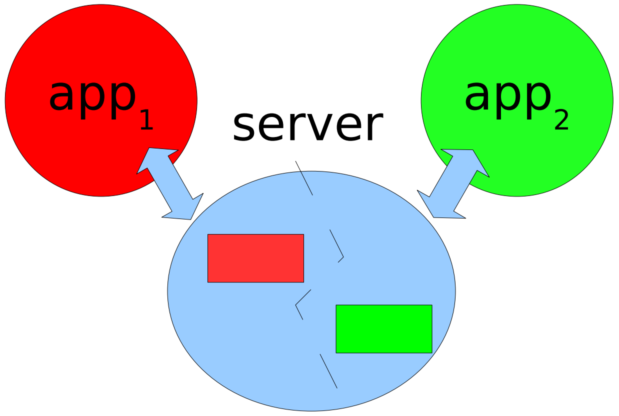
- Compile-time analyzes to enforce security policies
  - Problem:



    - OS-based (“peripheral”) policy enforcement mechanisms cannot control process-internal information flows.
  - Solutions:
    1) Reinstantiate server for differently classified clients
       not possible / feasible for all servers
       (device drivers, buffer cache, OS kernel)

Policy Enforcement Mechanisms

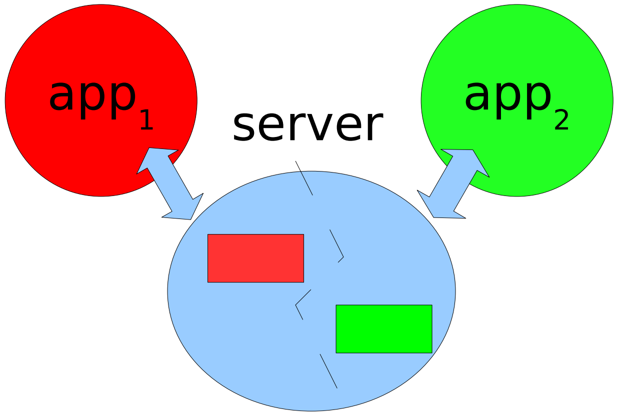
- Compile-time analyzes to enforce security policies
  - Problem:



    - OS-based (“peripheral”) policy enforcement mechanisms cannot control process-internal information flows.
  - Solutions:
    2) Trust server to enforce security policy
    (without enforcement mechanism)

Policy Enforcement Mechanisms

- Compile-time analyzes to enforce security policies
  - Problem:

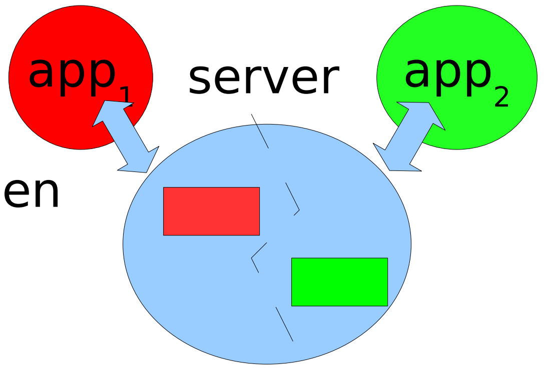


    - OS-based (“peripheral”) policy enforcement mechanisms cannot control process-internal information flows.
  - Solutions:
    3) Check policy enforcement with static (compile-time) analysis of server program
    Run only successfully checked servers on differently classified confidential data

Policy Enforcement Mechanisms

- Example of server internal information flow:
  - Server State:

```
int h;      // in red part of server state
            // possibly contains secret data
int l;      // eventually becomes visible to green
            // e.g., located in shared memory
```



  - Server Function:

```
void foo(int c) {
   if (c < 5)
        l = h;    // possible information leak from red to green
}
```

  - Check program at compile time for the occurrence of expressions such as l = h
  - Note: static analysis cannot decide whether certain input will ever occur in reality – here: server is secure if c >= 5

Overview

- Introduction
- Security Policies
- Policy Enforcement
- Decidability of Leakage
- Take Grant Protection Model
- Covert Channels
- Compiler-Based Information Flow Control

Decidability of Leakage

- Given
  - a security policy P
  - an enforcement mechanism (e.g., the ACM)
  - initial state σ_0

- Can we decide before the system runs (i.e., by considering only the initial state σ_0) whether it will reach a state in which P does not hold?

<u>If P is a security policy based on access rights</u>

- Can we decide before the system runs whether the system can reach a state in which a subject s has r rights over an object o (i.e., r is leaked to R(s,o))?
- **Theorem:**
  - It is undecidable for generic ACM-enforced systems whether they will reach a state in which a subject s has a generic right r over an object o!

Decidability of Leakage: ACM

- Definition:
  - Leakage: r is entered in R(s,o)
    - Does not take into account whether the security policy P authorizes r $\in$ R(s,o).
  - Decidability of Leackage:
    - Is there an algorithm that is able to decide before the system runs whether the system will leak a generic right r on an object o to a subject s
  - Theorem:
    - Leakage is undecidable for ACMs.
    - Proof: by reduction to the halting problem of a turing machine

Decidability of Leakage: ACM

- Theorem:
  - It is undecidable whether a system, which evolves from an initial state s_0, will leak a generic right r on o to s.

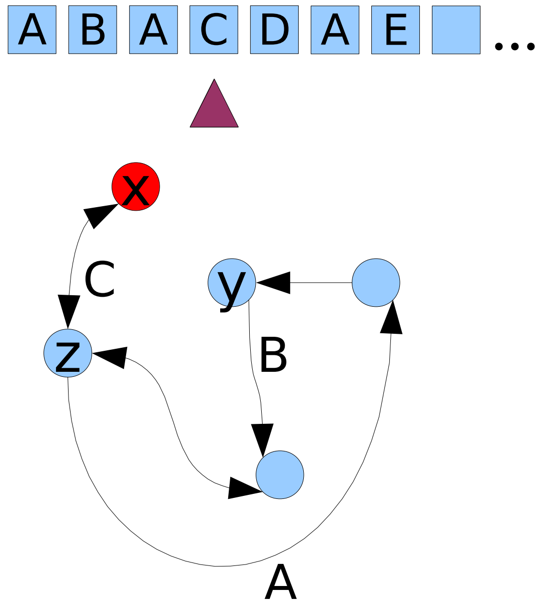
  - Proof by contradiction:
    Reduction to halting problem of Turing machine.
    - Simulate a Turing Machine with the help of an ACM
    - Relate the state of the ACM in which r is leaked to R(s,o)
      to the state of the TM in which a corresponding program halts

      => - because the specific ACM implements the TM such that
      the ACM leaks whenever the TM program halts
      - if leakage is decidable so would be the TM halting problem

- Leakage is decidable (in linear time) for the
  Take-Grant Protection Model

Turing Machine

- http://wiki...
  - Turing Machine
    - infinite tape
    - tape symbols M : A,B,C,...
    - state automaton K: x,y,z,...
    - head



    - TM transition: δ
      - read symbol from tape (at position of head)
      - perform an automaton transition dependent on this symbol
      - write a new symbol to the tape
      - move head one step to the left or to the right

δ: K x M -> K x M x {L,R}

Halting Problem

- http://wiki...
  Halting Problem:
  Given a TM and a Program P, find a program of the TM that decides whether P will terminate (halt).
- (TM ≅ universal TM ≅ <u>while</u>)
- Proof by contradiction: assume such a program exists

```
does_P_terminate_on_E (P, E) :=
   if P(E) terminates {}
       return true
   return false
```

```
test(P) :=
while(does_P_terminate_on_E(P, P))
```

  - if **does_P_terminate(test, test)** returns true => **test(test)** must terminate (if condition)
  - but then the condition of the while loop is true ↯
    => **test(test)** does not terminate

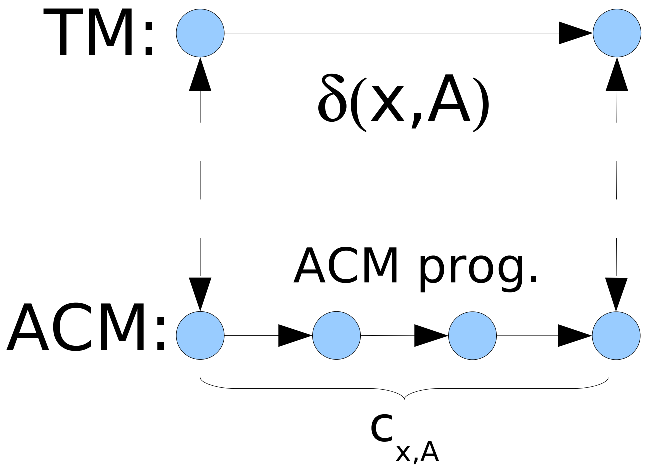
=> there can be no test such as **P(E) terminates** for all P, E

Proof: Leakage is undecidable with ACM

1) Formally define ACM and the ACM operations.

2) Construct a specific ACM, which simulates a generic TM.
   a) Construct a mapping between states of a generic TM and states of a specific ACM
   b) Simulate TM transitions with ACM programs such that each program yields a valid state that corresponds to a state of the TM

4) Correlate the state in which the ACM leaks r into R(s,o) to the state in which the TM halts given P(E)



Access Control Matrix

	o1	o2	s1	s2
s1	rd,wr	rd	rd,wr	rd
s2	rd,wr	-	wr	rd,wr

- ACM operations: C
  - create subject s
  - create object o
  - destroy subject s
  - destroy object o
  - enter right r into R(s,o)
  - delete right r from R(s,o)

Access Control Matrix

- **create subject s**

  Pre: $s \notin S$,

  Post: $S' = S \cup \{s\}$, // new subject
  $E' = E \cup \{s\}$, // subject also object
  $\forall x \in E': R'(s, x) = \emptyset$, // new subject has no rights
  $\forall y \in S': R'(y, s) = \emptyset$, // no rights on new subject
  $\forall x \in E, y \in S:$ // no change of old ACM cells
  $R'(x, y) = R(x, y)$

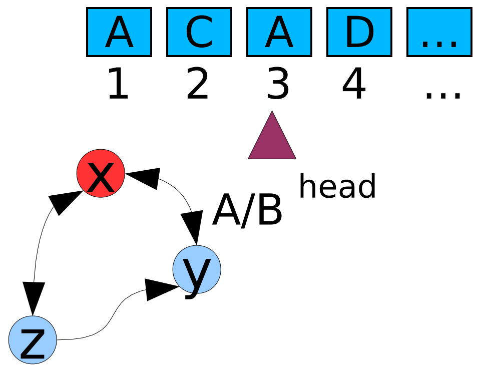
- **enter r into R(s,o)**

  Pre: $s \in S$, $o \in E$

  Post: $S' = S, E' = E$, // only R(s,o) changes
  $\forall x \in E', y \in S'$:
  $(s,o) \neq (x, y) \Rightarrow R'(x,y) = R(x, y)$
  $R'(s, o) = R(s, o) \cup \{r\}$ // add r to R(s,o)

Leakage is undecidable with ACM

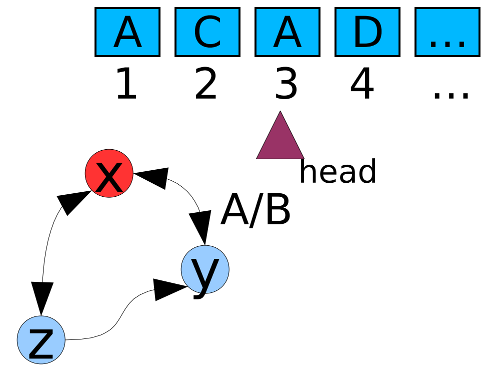
- Proof Sketch:



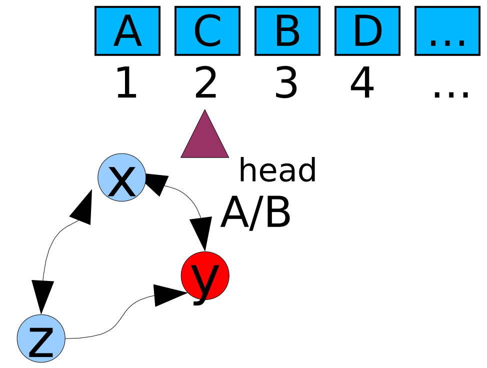
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A,x	
s_4			▲ head	D

Leakage is undecidable with ACM

- Proof Sketch:



δ: (*x*, *A*) -> (*y*, *B*, L)



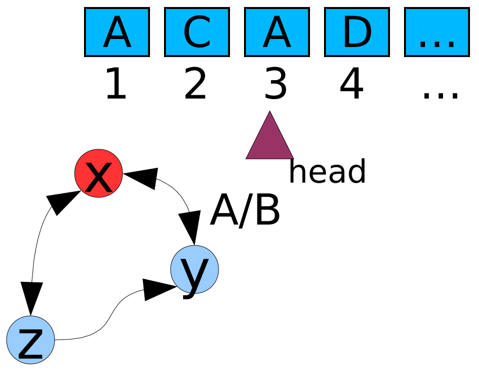
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A,x	
s_4				D

$c_{x,A}$

	s_1	s_2	s_3	s_4
s_1	A			
s_2		C,y		
s_3			B	
s_4				D

Leakage is undecidable with ACM

- Proof Sketch:



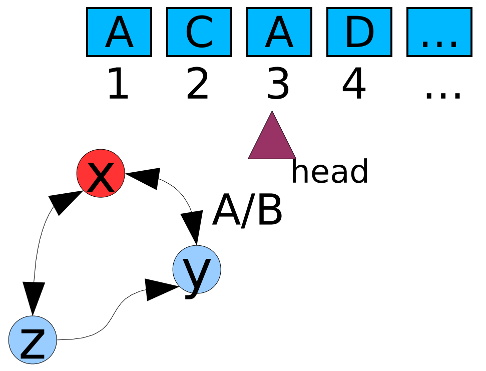
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A,x	
s_4				D

  - δ: $(x, A) \rightarrow (y, B, L)$

    $c_{x, A}(s_{head}, s_{left})$:=
      if $x \in R(s_{head}, s_{head})$ and
         $A \in R(s_{head}, s_{head})$ then
          ...

Leakage is undecidable with ACM

- Proof Sketch:



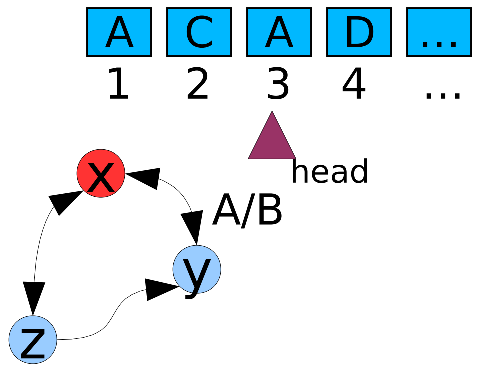
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A	
s_4				D

  - δ: $(x, A) \rightarrow (y, B, L)$

```
c_{x, A}(s_head, s_left) :=
  if  x ∈ R(s_head, s_head) and
      A ∈ R(s_head, s_head) then
    delete x from R(s_head, s_head)
      …
```

Leakage is undecidable with ACM

- Proof Sketch:



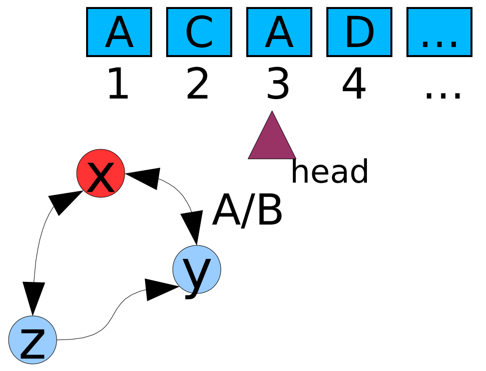
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3				
s_4				D

- δ: $(x, A) \rightarrow (y, B, L)$

```
$c_{x, A}(s_{head}, s_{left})$ :=
  if $x \in R(s_{head}, s_{head})$ and
     $A \in R(s_{head}, s_{head})$ then
    delete $x$ from $R(s_{head}, s_{head})$
    delete $A$ from $R(s_{head}, s_{head})$
      ...
```

Leakage is undecidable with ACM

- Proof Sketch:



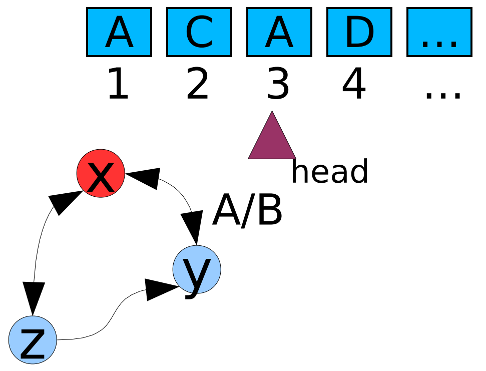
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			B	
s_4				D

- δ: (*x*, *A*) -> (*y*, *B*, L)

```
c_{x, A} (s_head, s_left) :=
  if  x ∈ R(s_head, s_head) and
      A ∈ R(s_head, s_head) then
    delete x from R(s_head, s_head)
    delete A from R(s_head, s_head)
    enter B into R(s_head, s_head)
       ...
```

Leakage is undecidable with ACM

- Proof Sketch:



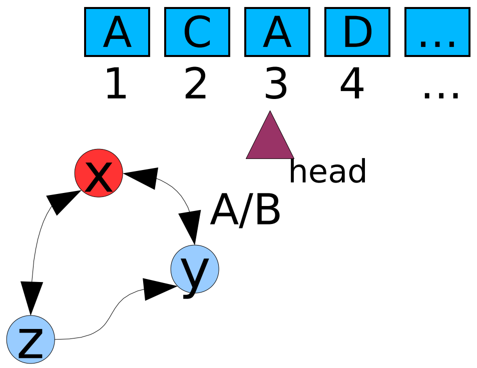
	s_1	s_2	s_3	s_4
s_1	A			
s_2		C,y		
s_3			B	
s_4				D

- δ: $(x, A) \rightarrow (y, B, L)$

```
c_{x, A} (s_head, s_left) :=
  if  x ∈ R(s_head, s_head) and
      A ∈ R(s_head, s_head) then
    delete x from R(s_head, s_head)
    delete A from R(s_head, s_head)
    enter B into R(s_head, s_head)
    enter y into R(s_left, s_left)
```

Leackage is undecidable with ACM

- Proof Sketch:

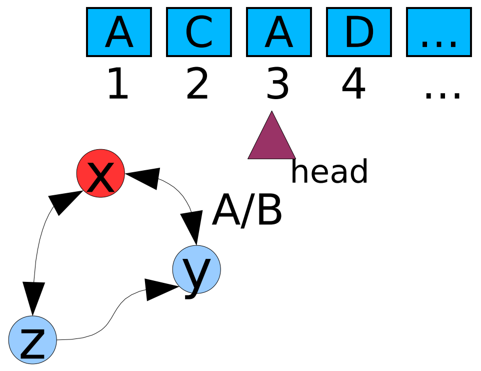


	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A	
s_4				D,x,end

- **Problem 1:**
  - δ: (*x*, *D*) -> (*y*, *B*, R) if head is in last cell (s_4,s_4)
    - distinguished right end to mark last cell
    - insert new subject s_5
    - propagate end right to s_5

Leakage is undecidable with ACM

- Proof Sketch:

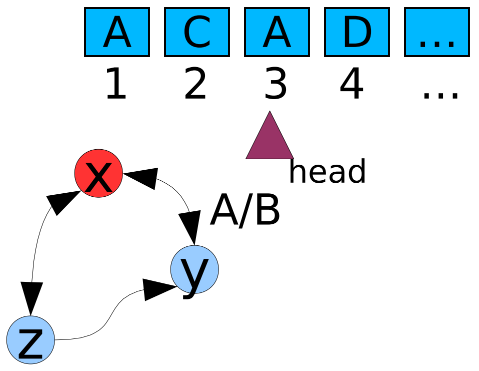


	s_1	s_2	s_3	s_4
s_1	A			
s_2		C		
s_3			A,x	
s_4				D,en d

- **Problem 2:** δ: (*x*, *A*) -> (*y*, *B*, L) $c_{x,\ A}\ (s_{head}, s_{left})$
  - Non-trivial problem:
    - Finite states + tape symbols but infinite many tape cells
      => subjects must remain parameters
      (otherwise infinite many ACM programs)
    - ACM has no way to express neighborhood (e.g., s_{left} is left of s_{head})

Leakage is undecidable with ACM

- Proof Sketch:

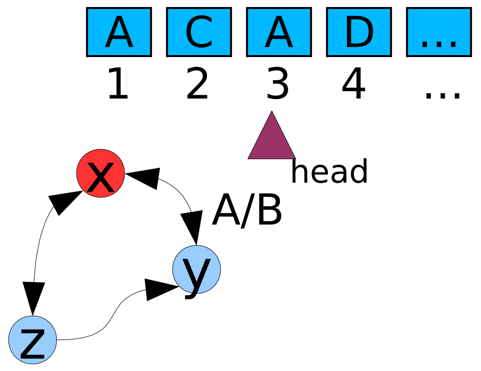


	s_1	s_2	s_3	s_4
s_1	A	own		
s_2		C	own	
s_3			A,x	own
s_4				D,end

- **Problem 2:** δ: (x, A) -> (y, B, L) $c_{x, A}$ (s_{head},s_{left})
  - Non-trivial problem:
    - Finite states + tape symbols but infinite many tape cells
      => subjects must remain parameters
      (otherwise infinite many ACM programs)
    - ACM has no way to express neighborhood (e.g., s_{left} is left of s_{head})
  - Solution: own $\in$ R(s_{head},s_{left})

Leakage is undecidable with ACM

- Proof Sketch:



	s_1	s_2	s_3	s_4
s_1	A	own		
s_2		C	own	
s_3			A,x	own
s_4				D,end

- δ: (*x*, *A*) -> (*y*, *B*, L)
  - $c_{x,A}(s_{head}, s_{left})$:=

```
if own ∈ R(s_left, s_head) and
      x ∈ R(s_head, s_head) and
    A ∈ R(s_head, s_head) then
  delete x from R(s_head, s_head)
  delete A from R(s_head, s_head)
  enter B into R(s_head, s_head)
  enter y into R(s_left, s_left)
```

=> TM (executing P(E)) halts at tape cell n in automaton state x with head tape symbol A **iff** A,x is leaked to $R(s_n, s_n)$.

=> if leakage would be decidable so is the halting problem

Overview

- Introduction
- Security Policies
- Policy Enforcement
- Decidability of Leakage
- Take Grant Protection Model
- Covert Channels
- Compiler-Based Information Flow Control

Take-Grant Protection Model

- Directed Graph
  - Vertices: ○ object, ● subject (⊘ either object or subject)
  - Edges: $\bullet \xrightarrow{r} \circ$ subject has capability with r right on object

  - Transition Rules:
    - Take

$$\begin{tikzcd} x \arrow[r, "t"] & y \arrow[r, "\beta"] & z & \vdash & x \arrow[r, "t"] \arrow[rr, bend left, "\beta"] & y \arrow[r, "\beta"] & z \end{tikzcd}$$

    - Grant

$$\begin{tikzcd} x & y \arrow[l, "g"'] \arrow[r, "\beta"] & z & \vdash & x \arrow[rr, bend left, "\beta"] & y \arrow[l, "g"'] \arrow[r, "\beta"] & z \end{tikzcd}$$

    - Create

$$\begin{tikzcd} x & \vdash & x \arrow[r, "\alpha"] & y \end{tikzcd}$$

    - Remove

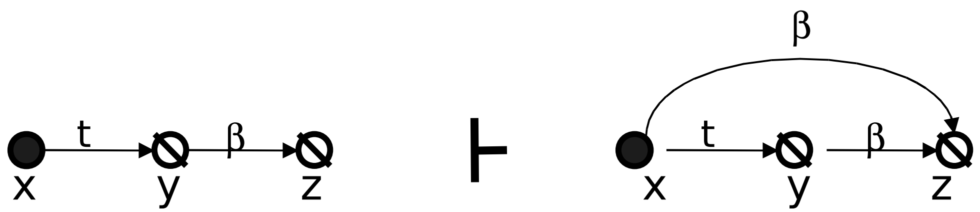
$$\begin{tikzcd} x \arrow[r, "\beta"] & y & \vdash & x \arrow[r, "\beta-\alpha"] & y \end{tikzcd}$$

    - Diminish

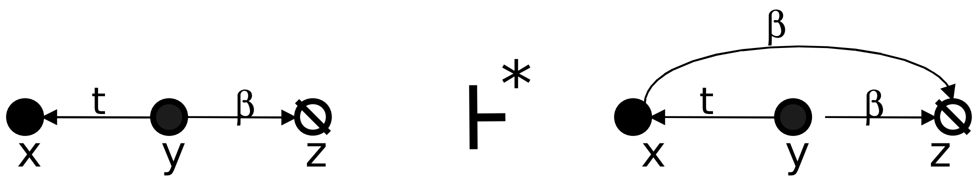
$$\begin{tikzcd} x \arrow[r, "\beta"] & y & \vdash & x \arrow[r, "\beta"] \arrow[r, bend left, "\beta-\alpha"] & y \end{tikzcd}$$

Take-Grant Protection Model

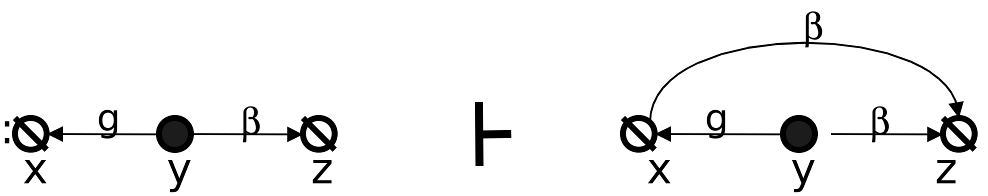
- 3 Lemmas:
  - Take Rule:



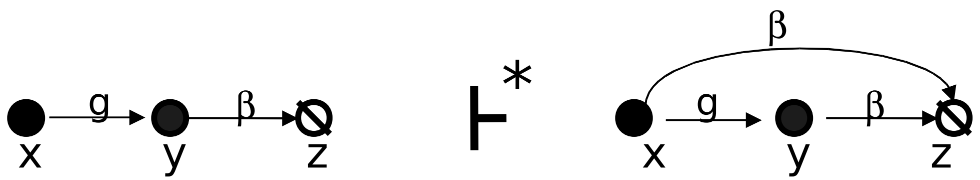
  - <u>Lemma 1:</u>



  - Grant Rule:

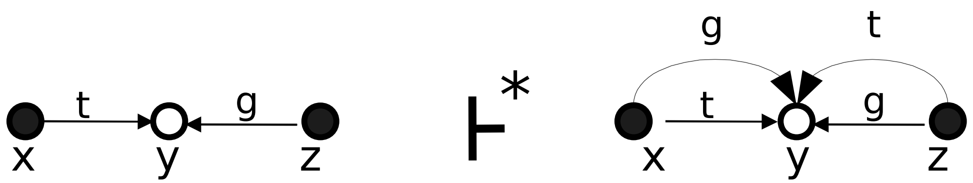


  - <u>Lemma 2:</u>



Take-Grant Protection Model

- 3 Lemmas:
  - <u>Lemma 3:</u>



Take-Grant Protection Model

- Proof of Lemma 1:

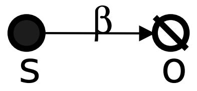
$$\begin{tikzcd} x & y \arrow[l, "t"'] \arrow[r, "\beta"] & z \end{tikzcd} \vdash^{*} \begin{tikzcd} x \arrow[rr, bend left, "\beta"] & y \arrow[l, "t"'] \arrow[r, "\beta"] & z \end{tikzcd}$$

  - Proof:
    x.create v (tg) ; y.take g ; $\underline{\text{y.grant } \beta \text{ to v}}$; x.take β from v

- See exercises for the proof of Lemmas 2, 3

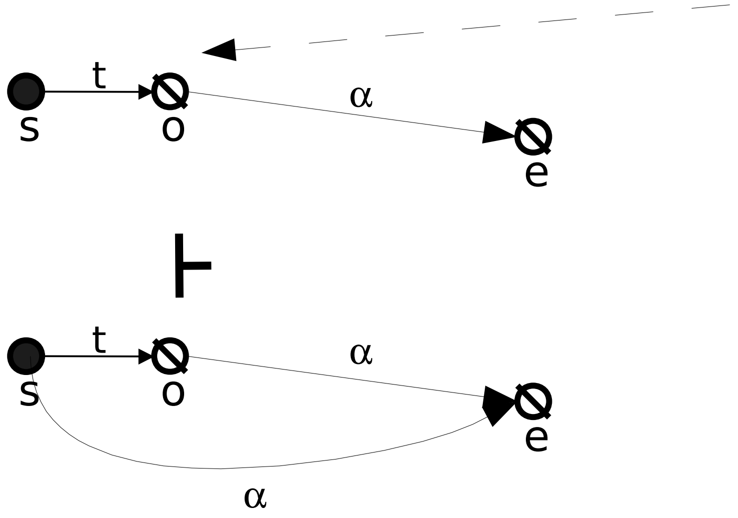
Take-Grant Protection Model

- Leakage is decidable in linear time in the Take-Grant Protection model.
  - Proof Sketch for decidability: (not: decidability in linear time)
    - construct potential-access graph (worst case rights propagation)
    - apply take + grant transition rules + the 3 lemmas until the no more rights can be added (i.e., the resulting potential-access graph no longer changes)
      - (delete, diminish, remove only reduce access rights)
      - (create establishes a new entity which cannot get no more privileges than its creator)
    - a right r on an object o can be leaked to a subject s if the potential access graph contains with r $\in \beta$



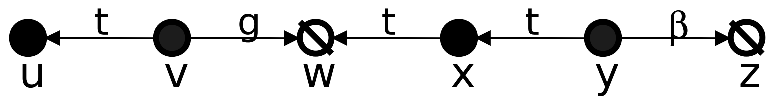
Take-Grant Protection Model

- Creating an Entity gives all rights to Creator
  - The creator s of an object o gets all permissions on o. In particular, s gets take permissions on o.
  - Assume a right r on e is leaked to o (i.e., o holds a capability (e,R) with r $\in$ R)
  - Then s can take this capability from o. => s can get all of o's rights



Take-Grant Protection Model

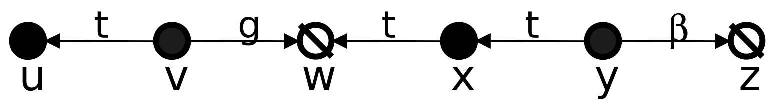
- Example: propagation of b on z to u
(towards a potential access graph)



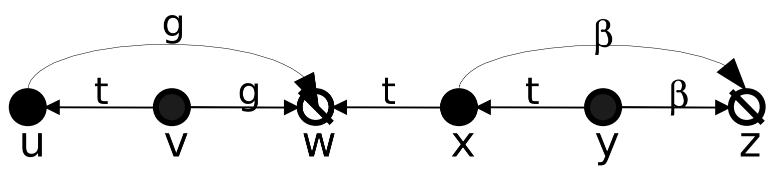
$\vdash^*$ by Lemma 1

Take-Grant Protection Model

- Example: propagation of b on z to u
  (towards a potential access graph)



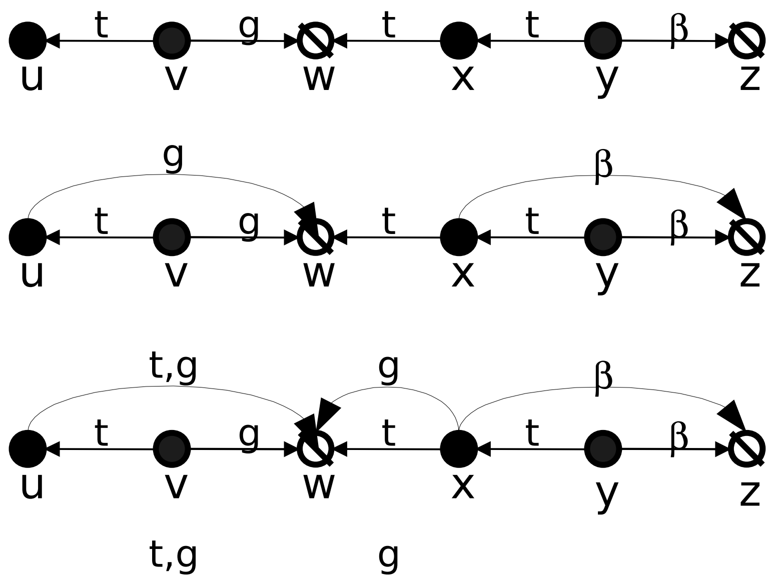
$\vdash^*$ by Lemma 1



$\vdash^*$ by Lemma 3

Take-Grant Protection Model

- Example: propagation of b on z to u
(towards a potential access graph)



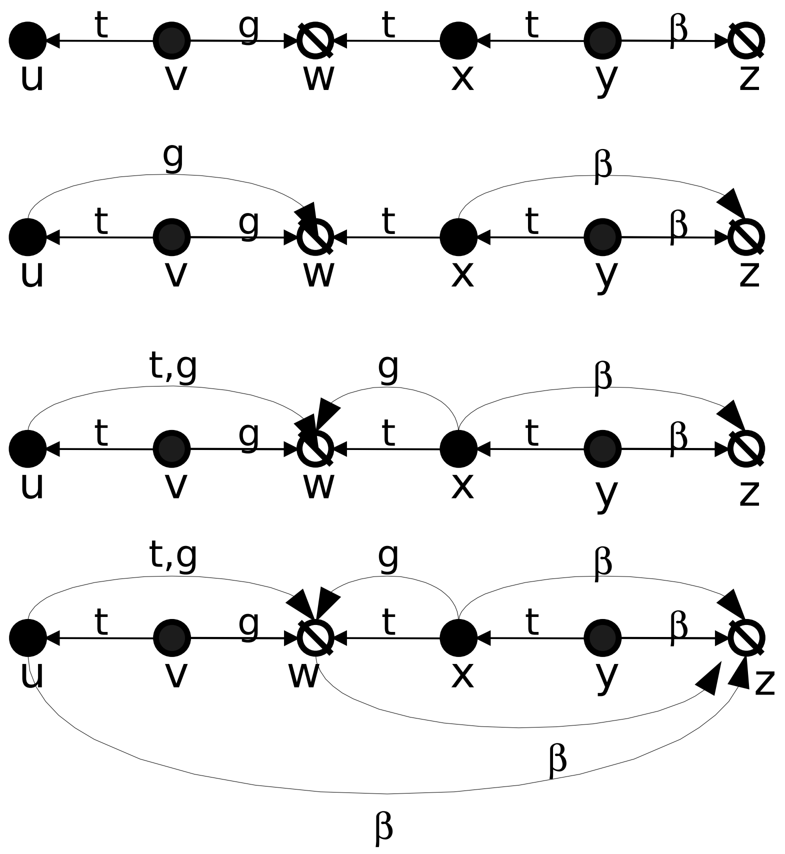
$\vdash^*$ by Lemma 1

$\vdash^*$ by Lemma 3

$\vdash^*$ x.grant β on z to h
u.take β on t from h

Take-Grant Protection Model

- Example: propagation of b on z to u
(towards a potential access graph)



$\vdash^*$ by Lemma 1

$\vdash^*$ by Lemma 3

$\vdash^*$ x.grant β on z to h
u.take β on t from h

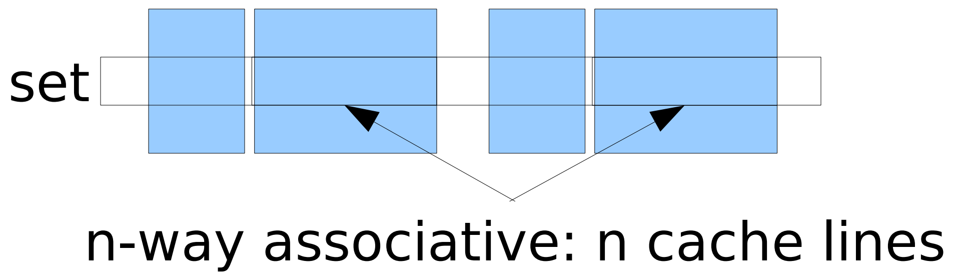
Overview

- Introduction
- Security Policies
- Policy Enforcement
- Decidability of Leakage
- Take Grant Protection Model
- Covert Channels
- Compiler-Based Information Flow Control

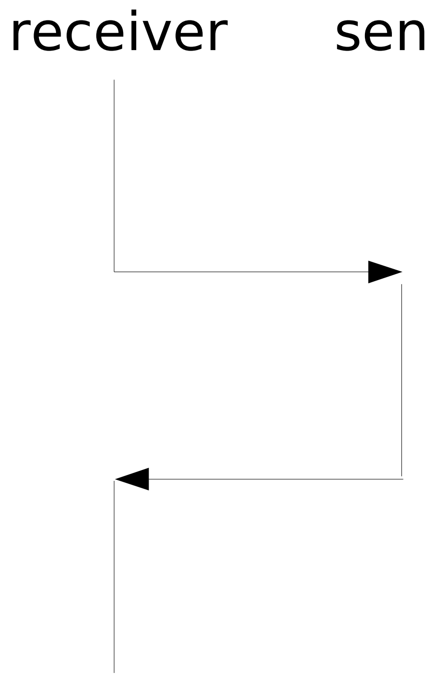
Covert Channels

- Covert Channel:
  - Lampson [73]:
    - Overt channel:
      - means of communication in the interface (e.g., read, write, error code)
    - Covert channel:
      - channel not intended for communication
  - TCSEC (Canadian predecessor of Common Criteria)
    - Covert channel:
      - Information flow in violation to the system's security policy
- Noise:
  - noiseless - only sender writes to covert channel
  - noisy - also other writers

Covert Channels: Cache



- Certain memory locations map to the same set of cache lines
- Cache replacement policy is set internal

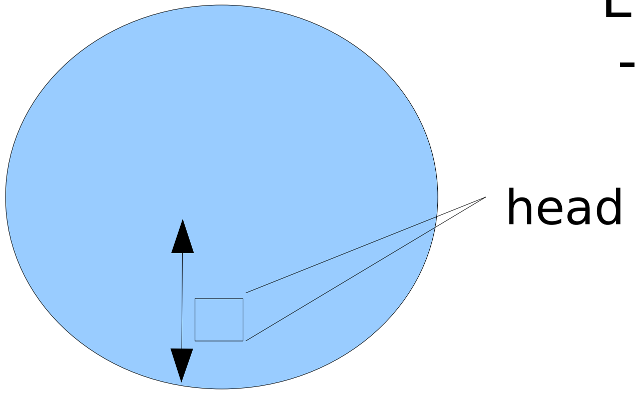


prepare cache:
  by accessing n memory locations that map to the same set

access certain cacheline of same set
  (e.g., AES – key dependent table lookups [Osvik])

probe timing of n memory locations:
  short: sender did not access CL of this set
  long: sender has evicted one (or more) of the n memory locations

Covert Channels: Disk [Wray]



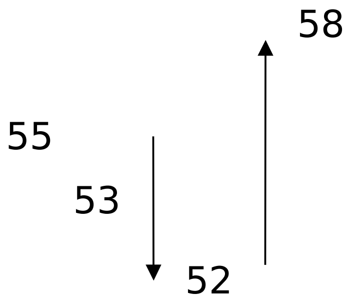
Elevator algorithm:
- cylinders in head movement direction are accessed first

<u>Prepare:</u>
read cyl. 55 ;
wait for completion

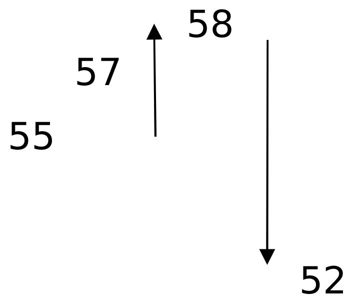
<u>Send:</u>
read cyl. 53 to send 0 or
read cyl. 57 to send 1
wait for completion

<u>Probe:</u>
read cyl. 52 and 58
observer order of completion

<u>Send 0:</u>



<u>Send 1:</u>



Covert Channels: in Programs

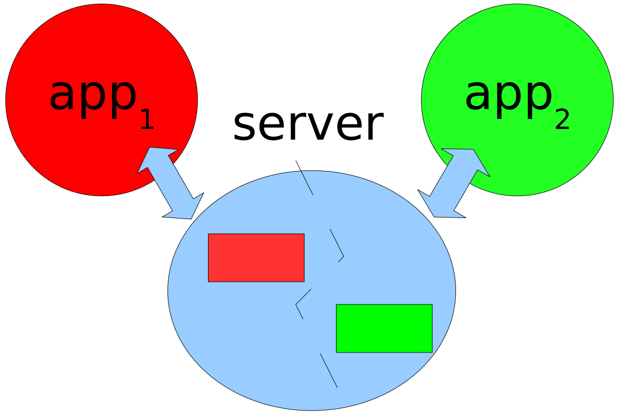
```
int l;      // eventually becomes observable by an l-classified observer
int h;      // stores a secret to which the l-classified observer is not cleared
```

<u>// explicit flow</u>

```
  l = h;
```

<u>// implicit flow</u>

```
  if (h % 2){
      l = 1;
  } else {
      l = 0;
  }
```



<u>// probabilistic</u>

```
  if (h % 2) {
     l = random (0, ..., 1);
  } else {
     l = 1;
  }
```

<u>// internal timing channel</u>

```
  if (h % 2) {
     l = 1 ; spin (10ms);
  } else {
     spin (10ms) ; l = 1;
  }
```

Covert Channels: in Programs

```
int l;      // eventually becomes observable by an l-classified observer
int h;      // stores a secret to which the l-classified observer is not cleared
```

<u>// external timing channel</u>

```
  if (h % 2) {
    // long op
    for (int i = 0; i < 10000; i++)
{}
  } else {
    // short op
  }
```

also h-dependent blocking:

```
            sleep(n ms)
```

<u>// termination</u>

```
  if (h % 2) while (true) {}
```

<u>// power, heat, …</u>

```
  if (h % 2)
      float_ops()
  else
      int_ops()
```

Noninterference

- Noninterference
  - Prevailing formalization for the complete absence of covert channels in deterministic systems (e.g., programs)
  - An l-classified observer sees the same output of a program p despite variations in secret (i.e., l'-classified) inputs (with $l \not\leq l'$).

$$s \sim_l s' \Rightarrow p(s) \sim_l p(s')$$

    - $s \sim_l s'$ stands for s, s' are indistinguishable by an l-classified observer.

Information Flow

- A new (more general?) formalism:
  - Confidentiality (Denning [67])
    - A ~/~> B =>
      B cannot deduce information on A (A's data), **A is confidential** with respect to B
  - Integrity (Denning [67])
    - A ~/~> B =>
      B's integrity is independent of information / results from A, B is integer with respect to A
  - Availability (Myers [05])
    - A ~/~> B =>
      B's availability is independent of information / results from A, B's availability cannot be affected by A
- Open Question: Is it possible to express any interesting access-control policy in terms of information flow?

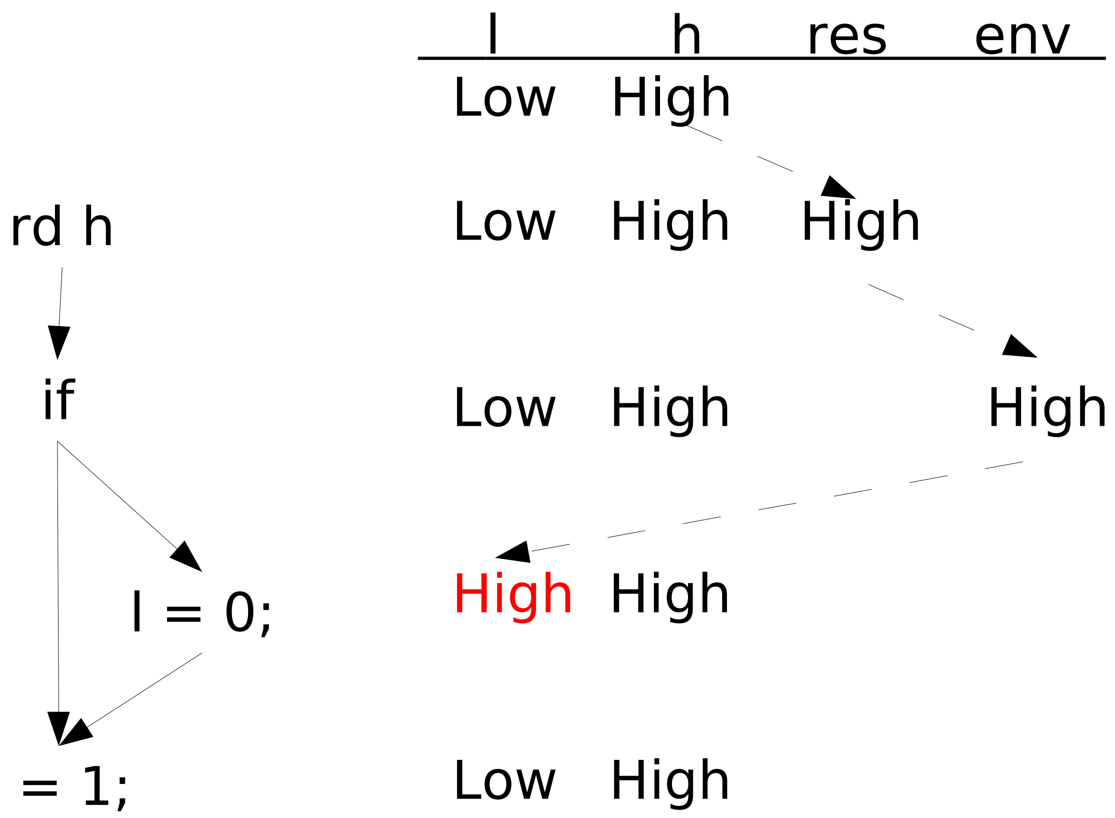
Compile-Time Information-Flow Analysis

- Flow Insensitive (Denning, Volpano)
- Flow Sensitive (Hunt, Warnier)
  - Abstract from concrete system state:
    - Start with:
      - clearance of output variables
      - classification of input variables / initially stored secrets
    - Abstract from concrete values;
      - maintain only secrecy levels of stored information

  - Abstractly interpret program
    - side-effect free expression: **f(in$_0$, ..., in$_1$) = out**
      - out can only encode secrets of in$_i$:
        dom(out) = least_upper_bound(dom(in$_i$))
    - control flow:
      - secrecy level **env** for the instruction pointer:
        wr(a, h) => dom(a) = lub(dom(h), env)

Compile-Time Information-Flow Analysis

- Example: if (h) { l = 0; } l = 1;



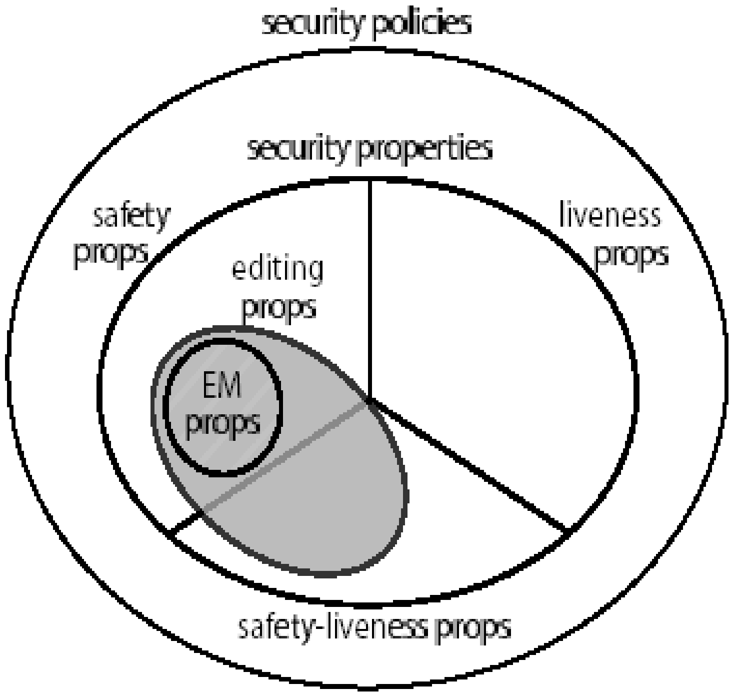
	l	h	res	env
	Low	High		
rd h	Low	High	High	
if	Low	High		High
l = 0;	High	High		
l = 1;	Low	High		

Questions

- References

B. Lampson:	A note on the confinement problem
Matt Bishop – Text Book:	Computer Security – Art and Science
P. Gallagher:	A Guide to Understanding the Covert Channel Analysis of Trusted Systems [TCSEC – CC Guide]
Proctor, Neumann:	Architectural Implications of Covert Channels
Sabelfeld, Myers:	Language-based information-flow security
Karger, Wray:	Storage Channels in Disk Arm Optimizations
Alpern, Schneider 87:	Recognizing safety and lifeness
Alves, Schneider:	Enforceable security policies
Walker, Bauer, Ligatti:	More enforcable security policies
Osvik, Shamir, Tromer:	Cache Attacks and Countermeasures: the Case of AES
Denning 67:	A Lattice Model of Secure Information Flow
Denning:	Certification of programs for secure information flow.
Hunt, Sands:	On flow-sensitive security types
Volpano, Irvine, Smith:	A sound type system for secure inform. flow analysis
Warnier:	Statically checking confidentiality via dynamic labels
Zheng, Myers:	End-to-End Availability Policies and Noninterference
Shapiro, Smith, Farber:	EROS: A Fast Capability System

Security Policies – Safety / Lifeness



System:

- Commands C := $\{c_1, c_2, \ldots, c_n\}$
- Set of action traces
  T := $\{ <c_1 c_2 c_1>, <c_3 c_1 c_6 c_4>, \ldots \}$

Security Policy:

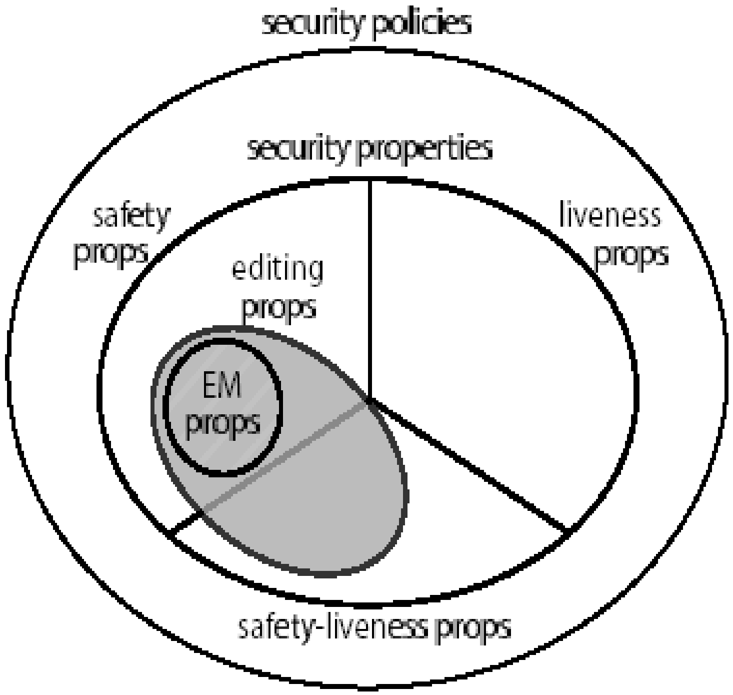
- Predicate on subsets of T

Security Property:

- Predicate on a single trace
  $P(T) := \forall\, t \in T.\ P'(t)$

- Security Property:
  - Decission whether system is secure can be made by just observing a single execution of the system
- Security Policy:
  - Can also compare multiple executions of the system

Security Policies – Safety / Lifeness



System:

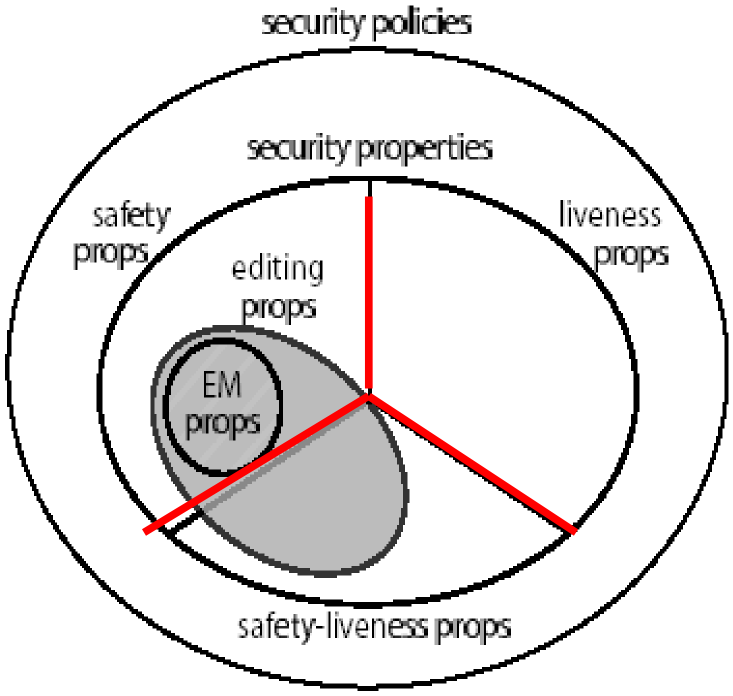
- Commands C := $\{c_1, c_2, \ldots, c_n\}$
- Set of action traces T := $\{ <c_1 c_2 c_1>, <c_3 c_1 c_6 c_4>, \ldots \}$

Example: Noninterference

- Indistinguishable despite variations in high inputs
- H $\subseteq$ C actions $c_i(h)$ on high input (h)
- $c_3 c(h)_6 c_4$ and $c_3 c(h')_6 c_4$ produce l-similar results

=> Noninterference is Security Policy but not a Security Property!

Security Policies – Safety / Lifeness



Safety property:

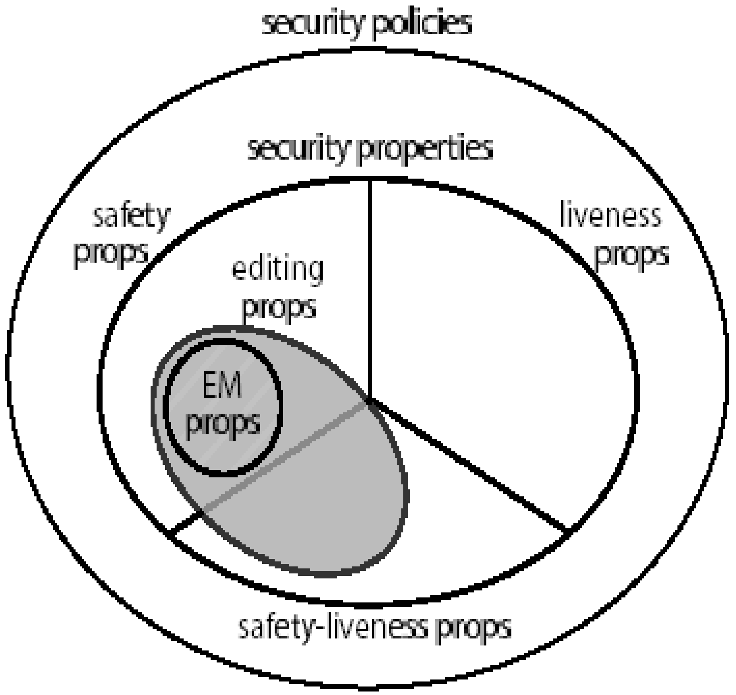
- "Rules out bad things"
- ¬P(t) states that the system is insecure because $\sigma_0 \xrightarrow{t} \sigma'$ and something "bad" is going on in σ'
- $\neg P(t) \Rightarrow \forall t'. \neg P(t\ t')$
  - A system that is insecure will remain insecure when it continues to execute.

Lifeness property:

- "A system can stay good"
  - $\forall \sigma. \exists \sigma'. \sigma \rightarrow^* \sigma' \Rightarrow P(\sigma')$

- Alpern, Schneider [87]: "Recognizing safety and lifeness"
  - Any security property can be expressed as a conjunct of safety and lifeness properties.

Security Policies – Safety / Lifeness



- **Alves, Schneider: “Enforceable Security Policies”**
  - EM automata can only enforce safety properties
- **Walker, Bauer, Ligatti: “More enforceable Sec. Policies”**
  - Edit automata can also enforce some safety+lifeness properties
  - Neither EM nor Edit automata can enforce pure lifeness properties