Defining and Enforcing Referential Security

Jed Liu Andrew C. Myers

Cornell University
Department of Computer Science

3rd Conference on Principles of Security and Trust
7 April 2014

Referential security

- Distributed systems span multiple trust domains
- Natural to have cross-domain references
  - e.g., hyperlinks (web), foreign keys (DBs), CORBA, RMI, JPA+JTA, Fabric

Referential security

- Distributed systems span multiple trust domains
- Natural to have cross-domain references
  - e.g., hyperlinks (web), foreign keys (DBs), CORBA, RMI, JPA+JTA, Fabric
- **Problem:** references introduce dependencies
  - Can create security & reliability vulnerabilities
    - New class of **referential security** vulnerabilities
- First step towards programming model for writing code without these vulnerabilities

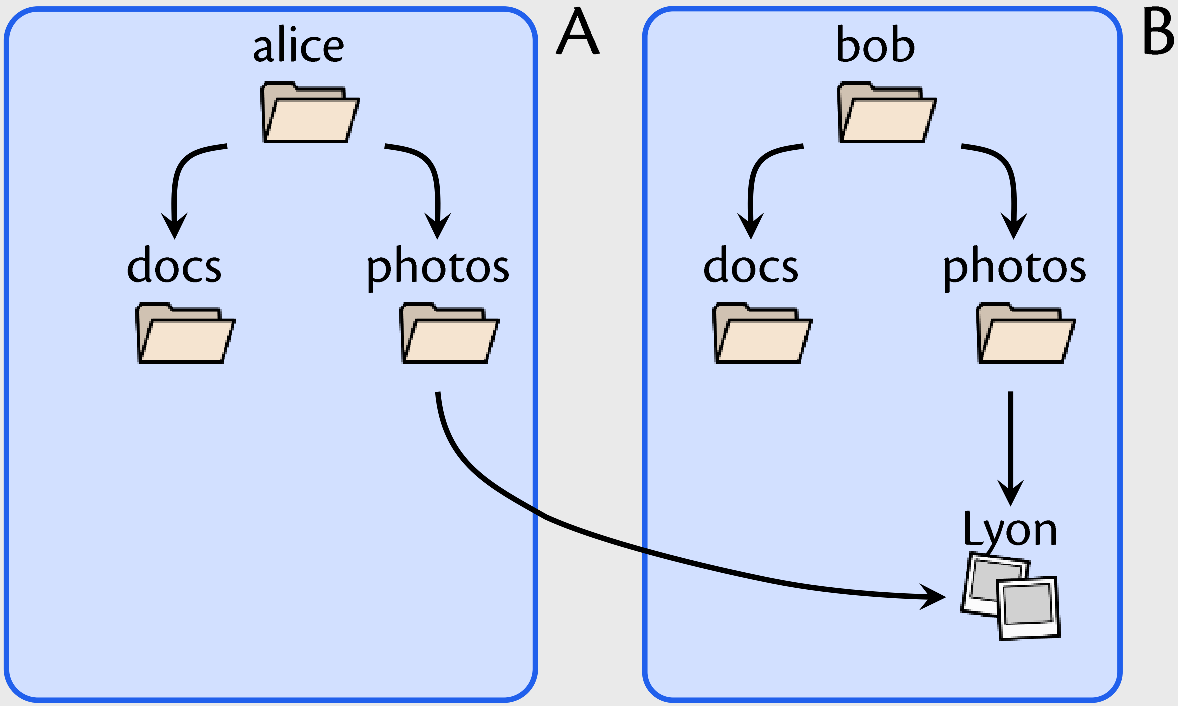
Referential security

- Distributed systems span multiple trust domains
- Natural to have cross-domain references
  - e.g., hyperlinks (web), JPA+JTA (distributed DBs)
- **Problem:** references introduce dependencies
  - Can create security & reliability vulnerabilities

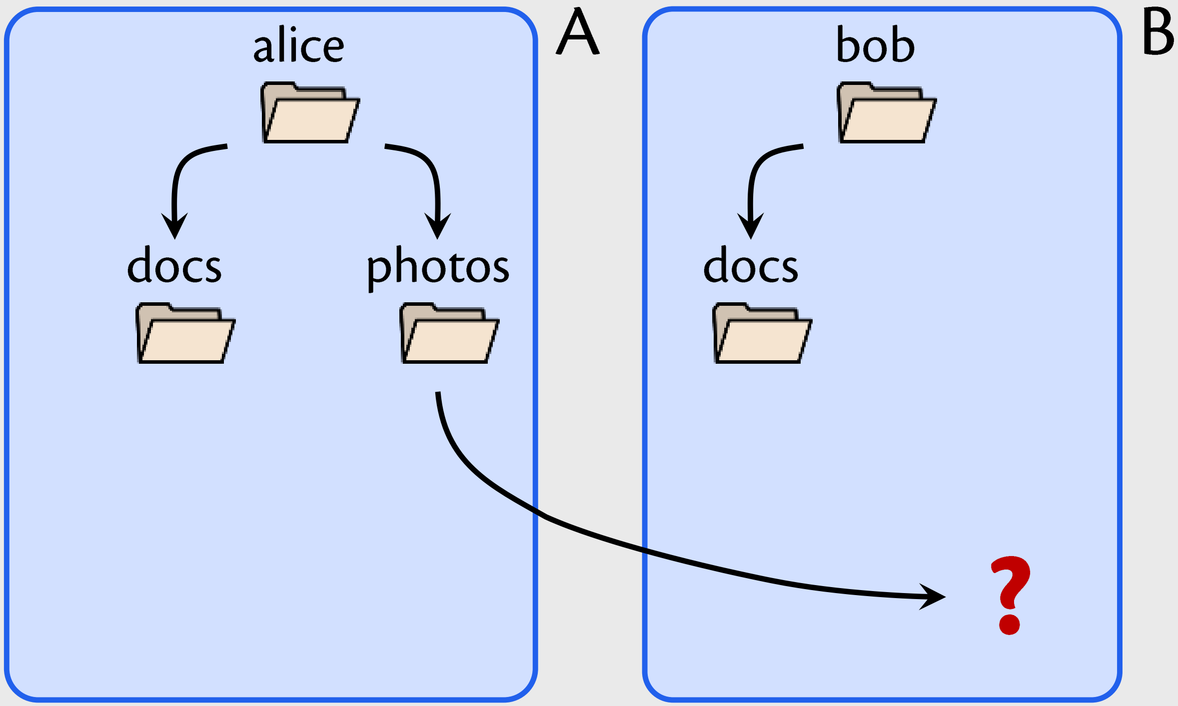
Contributions

- Formalized three **referential security** goals
- Static analysis (type system) to enforce them
- Soundness proof

Directory example



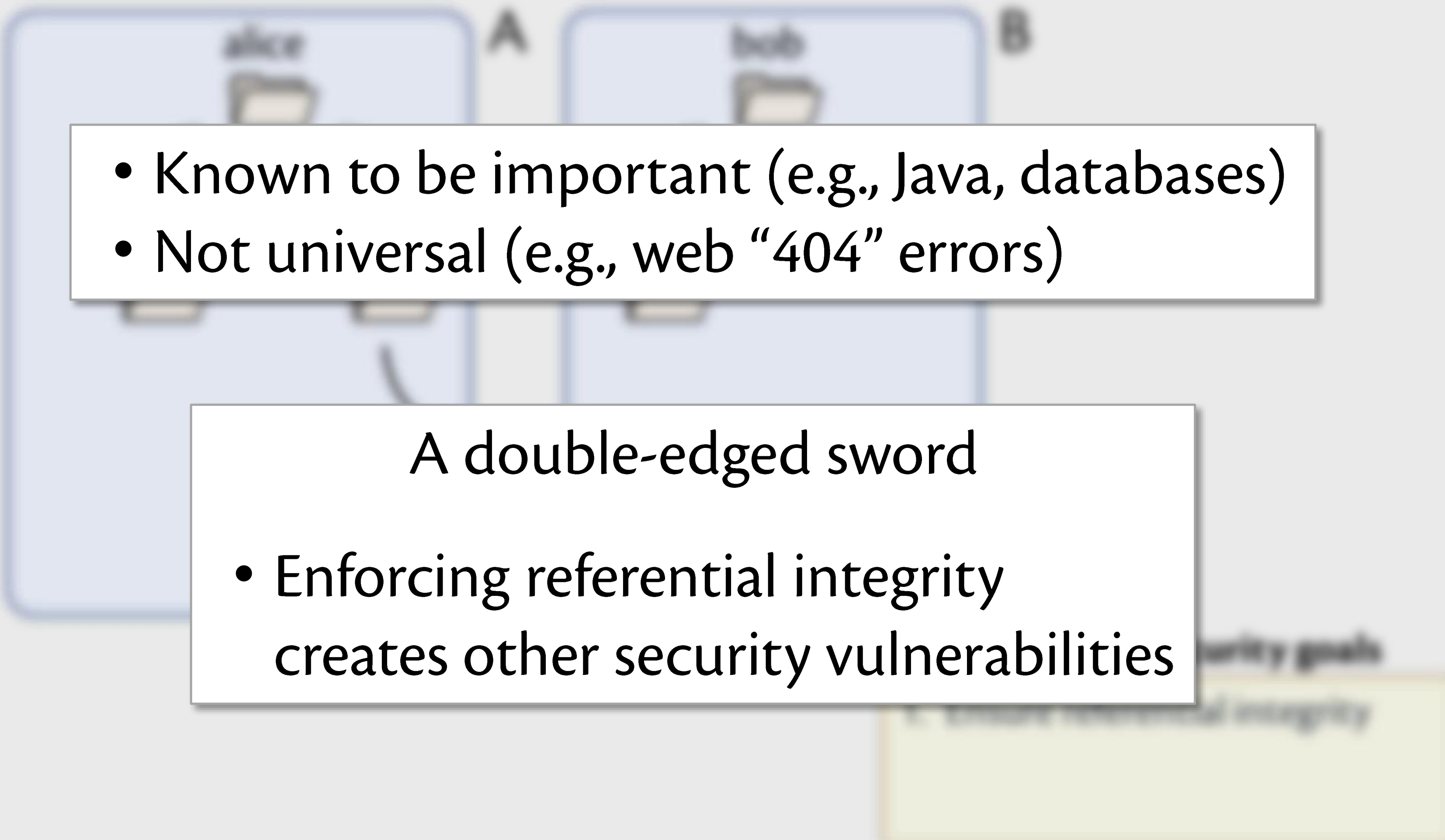
Referential integrity



Referential security goals

1. Ensure referential integrity

Referential integrity

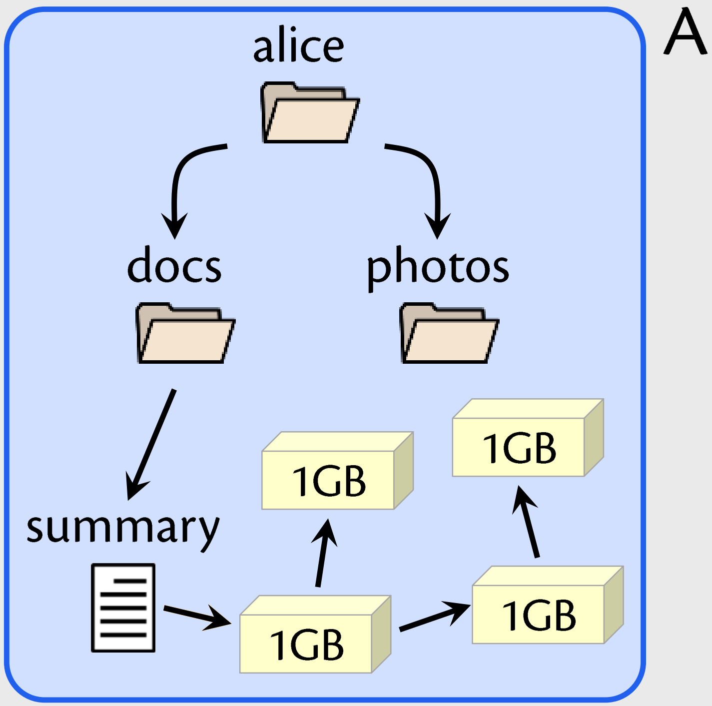


- Known to be important (e.g., Java, databases)
- Not universal (e.g., web “404” errors)

A double-edged sword

- Enforcing referential integrity creates other security vulnerabilities

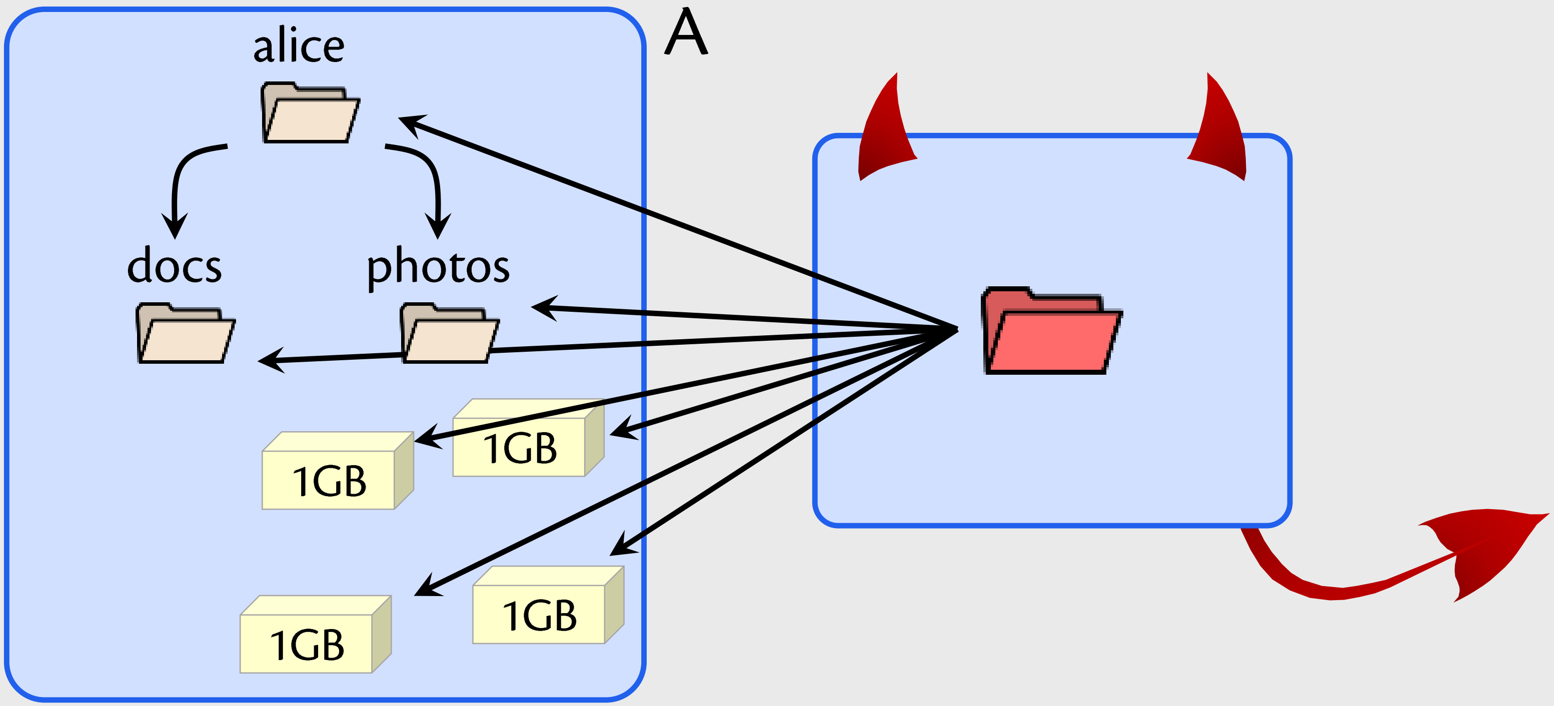
Accidental persistence



Referential security goals

1. Ensure referential integrity
2. Prevent accidental persistence

Storage attacks



Referential security goals

1. Ensure referential integrity
2. Prevent accidental persistence
3. Prevent storage attacks

A framework for referential security

- Static analysis for enforcing referential security:

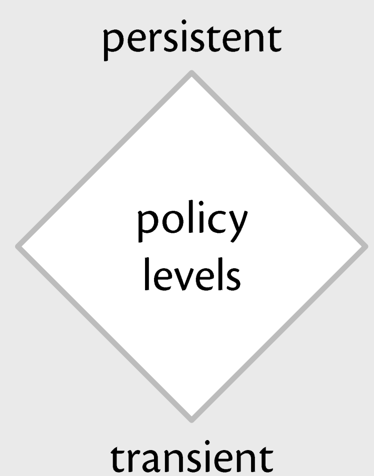
- Presented as type system of $\lambda_{persist}$ language
- $\lambda_{persist}$ extends $\lambda^{\rightarrow}$ with:
  - objects (mutable records)
  - references (immutable references to records)

1. Ensure referential integrity
2. Prevent accidental persistence
3. Prevent storage attacks

Preventing accidental persistence

- Persist by policy, not by reachability
- Ea
  pe

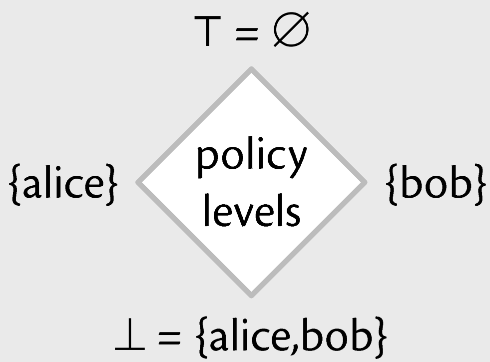
1. Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks



Preventing accidental persistence

- Persist by policy, not by reachability
- Each object has a **persistence policy** *p*

Node-set interpretation:
Who can delete object?



1. Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

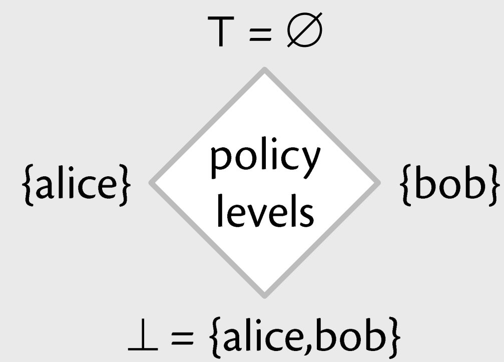
Ensuring referential integrity

- Type system ensures all persistence failures are handled

  try e_1 catch p: e_2

  – Factors out failure-handling code

Who can delete object?



✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

Ensuring referential integrity

- Type system ensures all persistence failures are handled

  `try e₁ catch p: e₂`

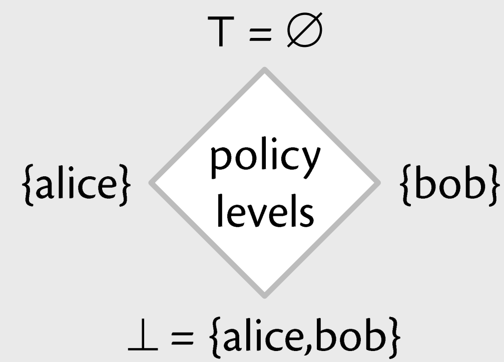
  - Factors out failure-handling code

- Typing judgement:

  $$\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$$

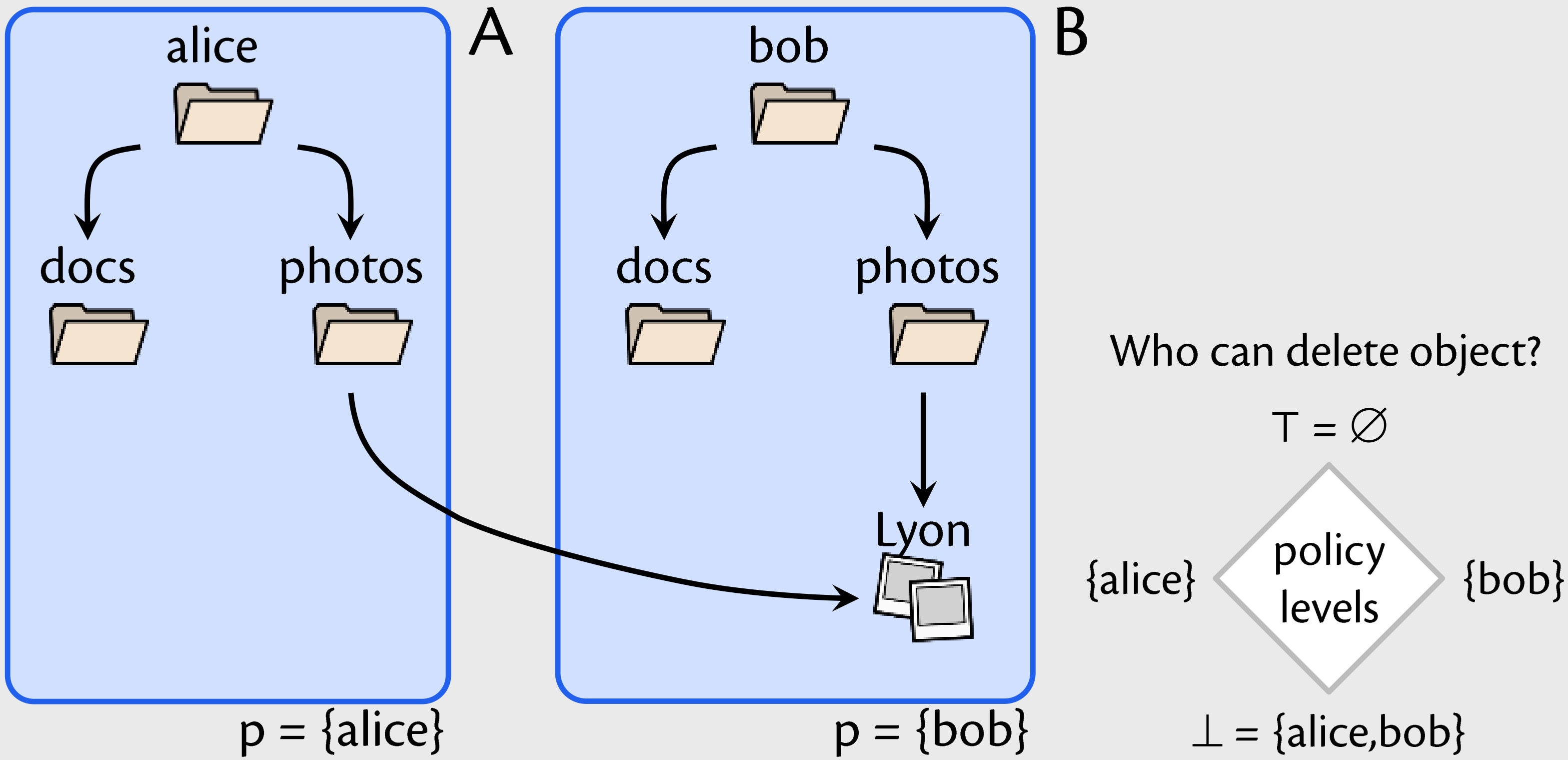
  - $\mathcal{H}$ = failures handled by context
  - $\mathcal{X}$ = possible failures produced by *e*
  - Invariant: $\mathcal{X} \subseteq \mathcal{H}$

Who can delete object?



✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

Directory example

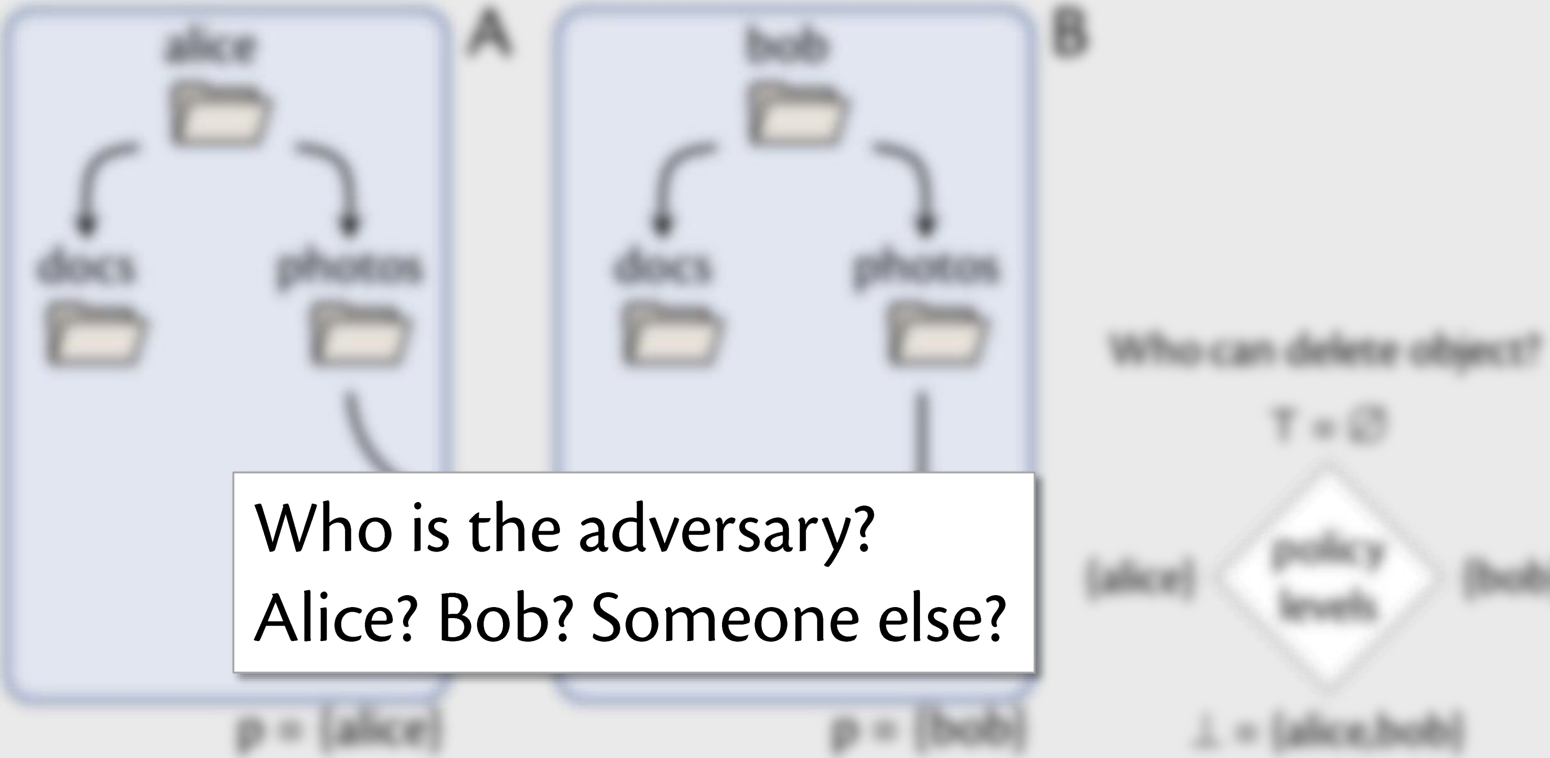


- Programs must be ready to handle failure:

```
try Lyon.show () catch bob: ...
```

✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

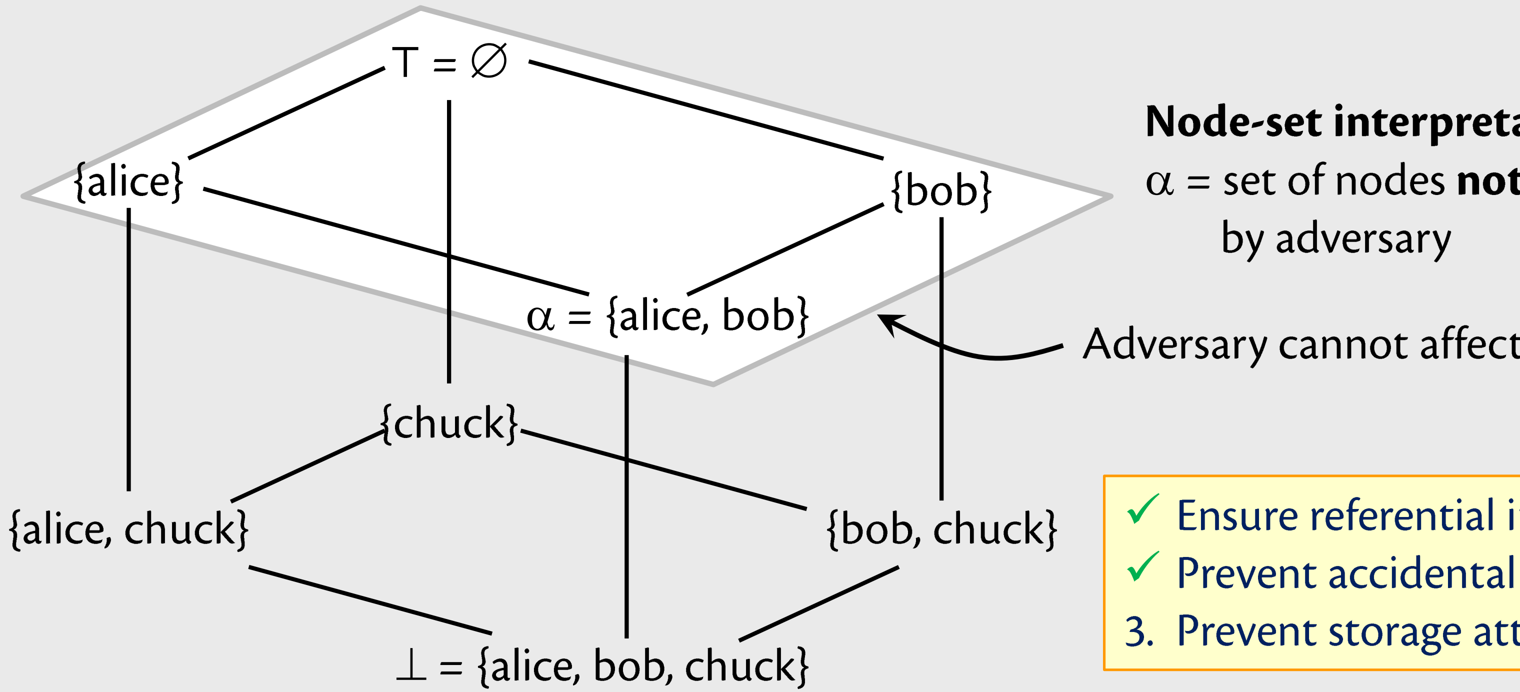
Directory example



Who is the adversary?
Alice? Bob? Someone else?

Modelling the adversary

- Assume adversary controls some nodes in system
- Adversary modelled as a point α on lattice
  - Cannot affect objects having policies at or above α

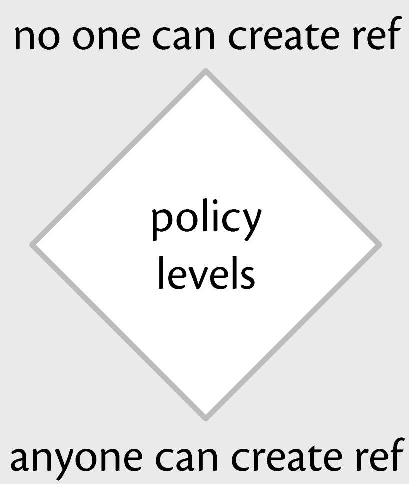


Node-set interpretation:
α = set of nodes **not** controlled by adversary

✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

Preventing storage attacks

- Each object has a **creation authority policy** *a*
  - **Authority policy** for short
  - Restricts ability to create new refs
  - Taken from same lattice as persistence policies

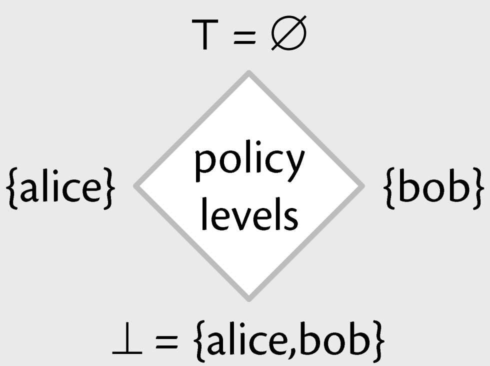


✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

Preventing storage attacks

- Each object has a **creation authority policy** *a*
  - **Authority policy** for short
  - Restricts ability to create new refs
  - Taken from same lattice as persistence policies

Node-set interpretation:
Who can create reference?

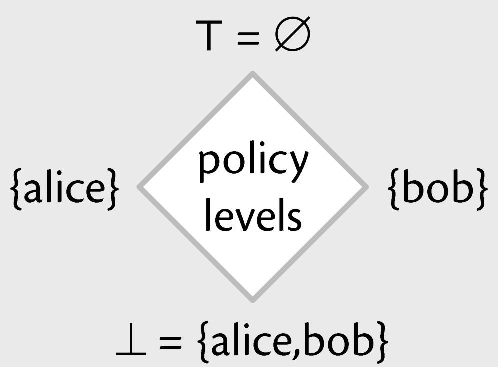


✓ Ensure referential integrity
✓ Prevent accidental persistence
3. Prevent storage attacks

Preventing storage attacks

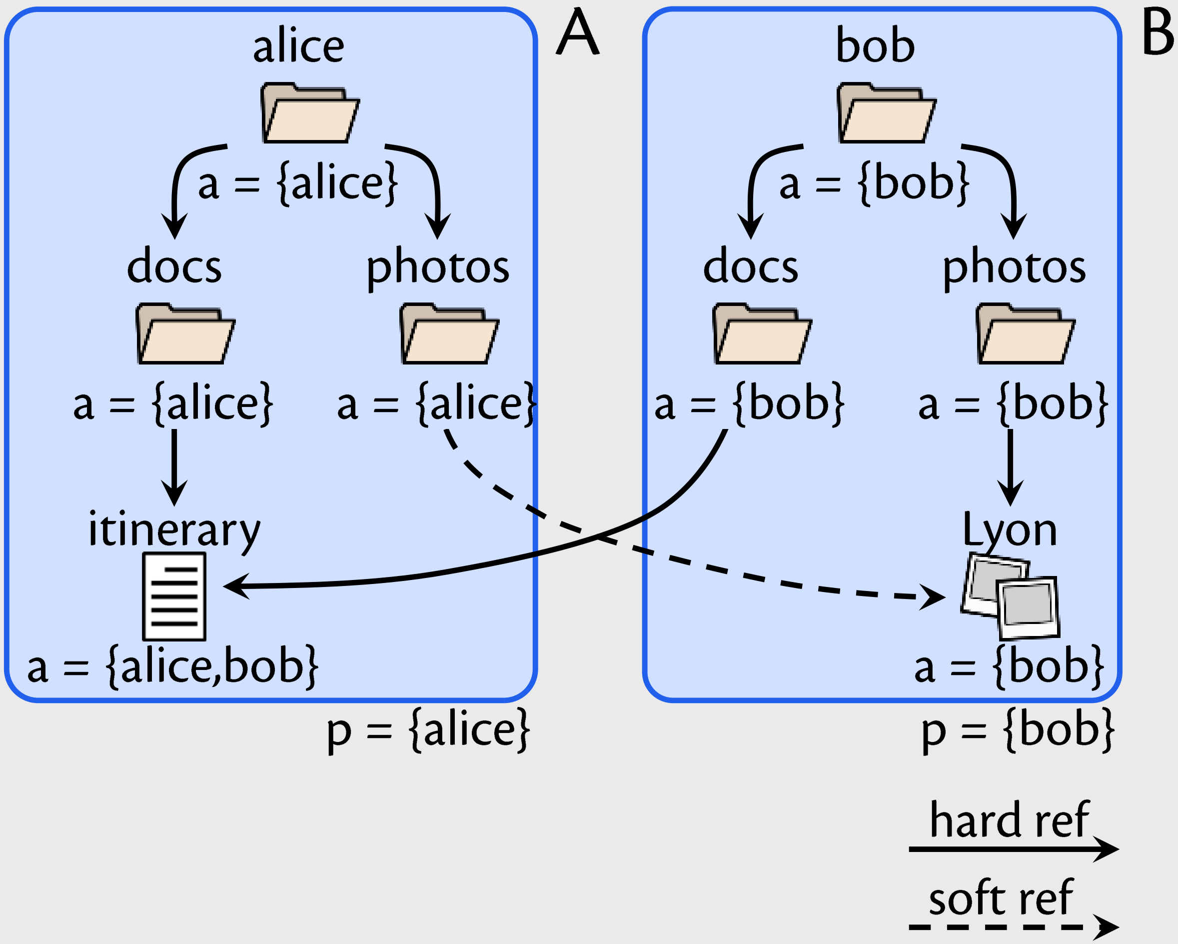
- Each object has a **creation authority policy** *a*
  - **Authority policy** for short
  - Restricts ability to create new refs
  - Taken from same lattice as persistence policies
- What if you don't have authority?
  - **(Hard) References** have referential integrity, require authority
  - **Soft references** do not

Host-set interpretation:
Who can create reference?

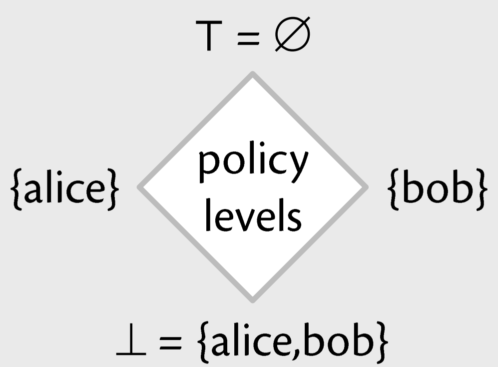


✓ Ensure referential integrity
✓ Prevent accidental persistence
✓ Prevent storage attacks

Example



Who can create reference?



Integrity

- Adversary controls some nodes
  - Can modify some objects → affect program state
  - Can affect decision to create references

```
if L then x.f = o
```

- To enforce authority, type system tracks:
  - Integrity of values
  - Integrity of control flow

$$\Gamma; pc; \mathcal{H} \vdash e : \tau, \mathcal{X}$$

    - *pc* bounds authority of references created by *e*

Policies on reference types

- Reference types have policies too
  - Persistence policy p
    - Lower bound on persistence of referent
    - Ensures persistence failures are handled when using ref
  - Authority policy a^+
    - Upper bound on authority required by referent
    - Prevents storage attacks: need a^+ authority to copy ref

- Subtyping contravariant on p, covariant on a^+

λ_{persist}

Base types	$b ::= \ \text{bool} \mid \tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2 \mid R \mid \text{soft } R$
Types	$\tau ::= b_w \mid \mathbf{1}$
Values	$v, u ::= \ x \mid \text{true} \mid \text{false} \mid * \mid m^S \mid \text{soft } m^S \mid \lambda(x:\tau)[pc;\mathcal{H}].\, e \ \ (\mid \bot_p)$
Terms	$e ::= \ v \mid v_1\ v_2 \mid \text{if } v_1 \text{ then } e_2 \text{ else } e_3 \mid \{\overrightarrow{x_i = v_i}\}^S \mid v.x \mid v_1.x := v_2$
	$\mid \text{soft } e \mid e_1 \Vert e_2 \mid \text{exists } v \text{ as } x:\ e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2$
	$\mid \text{try } e_1 \text{ catch } p\text{: } e_2$

- soft *e* – creates soft ref out of hard ref
- exists *v* as *x*: e_1 else e_2
  - checks whether soft ref still valid (if yes, promotes to hard ref)
- try e_1 catch *p*: e_2 – persistence-failure handler

λ_{persist}

$$
\begin{array}{rrl}
\text{Base types} & b ::= & \text{bool} \mid \tau_1 \xrightarrow{pc,\mathcal{H}} \tau_2 \mid R \mid \text{soft } R \\
\text{Types} & \tau ::= & b_w \mid \mathbf{1} \\
\text{Values} & v, u ::= & x \mid \text{true} \mid \text{false} \mid * \mid m^S \mid \text{soft } m^S \mid \lambda(x:\tau)[pc;\mathcal{H}].\, e \;\; (\mid \perp_p) \\
\text{Terms} & e ::= & v \mid v_1\ v_2 \mid \text{if } v_1 \text{ then } e_2 \text{ else } e_3 \mid \{\overrightarrow{x_i = v_i}\}^S \mid v.x \mid v_1.x := v_2 \\
 & & \mid \text{soft } e \mid e_1 \Vert e_2 \mid \text{exists } v \text{ as } x : e_1 \text{ else } e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\
 & & \mid \text{try } e_1 \text{ catch } p\text{: } e_2
\end{array}
$$

- Operational semantics
  - Machine configuration: $<e, M>$

    partially evaluated program

    program memory
    - maps typed locations m^S to records or to $\perp$ if deleted
  - Small step: $<e_1, M_1> \rightarrow <e_2, M_2>$
    - Includes model of garbage collector

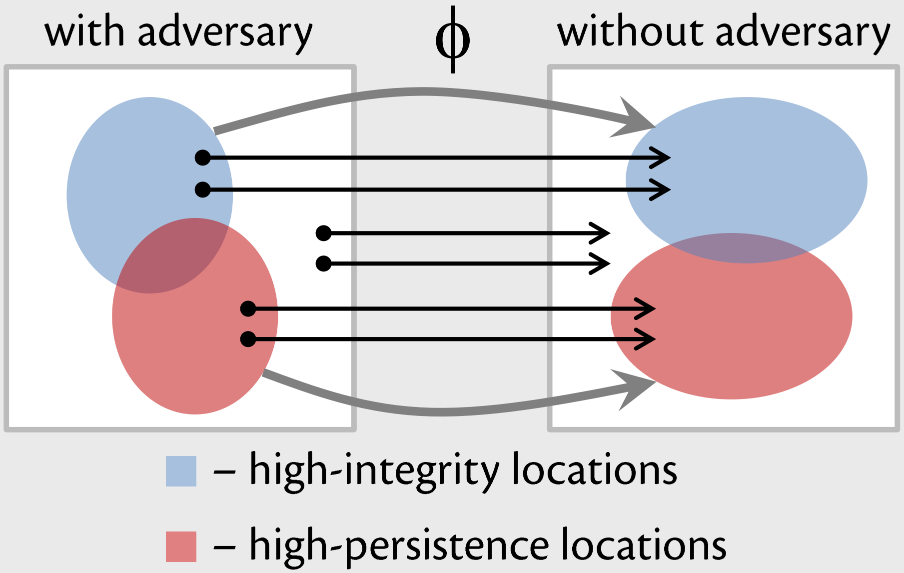
Power of the adversary

- Between program steps, adversary can arbitrarily:
  - Create new objects
    - Objects must have low integrity & low persistence
  - Assign into low-integrity fields
  - Delete low-persistence objects

- Matches assumption: adversary has total control over its nodes

Proving referential security

- Idea: execution with adversary should be "equivalent" to execution without adversary
- But memory locations may not match up
  - Relate traces using **homomorphism** ϕ on typed locations



Properties of ϕ

- Partial
- Injective
- Type-preserving
- Isomorphic when restricted to:
  - high-integrity locations
  - high-persistence locations

Security relation

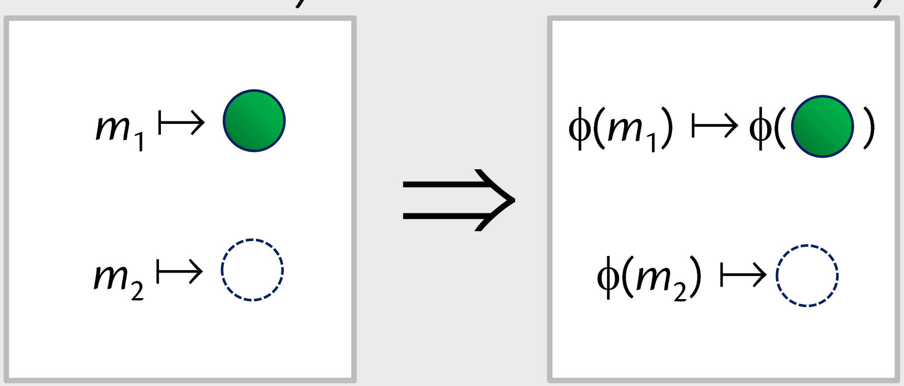
- For expressions: $e_1 \approx_{\alpha}^{\phi} e_2$
  - Expressions are equivalent when locations are transformed by ϕ

Security relation

- For expressions: $e_1 \approx_\alpha^\phi e_2$
  - Expressions are equivalent when locations are transformed by ϕ
- For memories: $M_1 \approx_\alpha^\phi M_2$

with adversary

without adversary



where m_1 is mapped by ϕ

where m_2 is high-authority and high-persistence

Referential security

- **Theorem:**
  Security relation is preserved by computation

$$
\begin{array}{rccc}
 & \langle e_1, M_1\rangle & \longrightarrow_{\alpha} & \langle e'_1, M'_1\rangle \\
\text{with adversary} & & & \\
 & \wr\wr_{\phi,\alpha} & & \wr\wr_{\phi',\alpha} \\
\text{without adversary} & & & \\
 & \langle e_2, M_2\rangle & \longrightarrow^{*} & \langle e'_2, M'_2\rangle
\end{array}
$$

  (assuming e_i well-typed and certain well-formedness conditions)

- **Lemma:** Adversary cannot cause more high-authority locations to become non-collectible

Related work

- System mechanisms (orthogonal to lang. model)
  - e.g., improving referential integrity of hyperlinks
- Liblit & Aiken
  - Type system for distributed data structs (no security)
- Riely & Hennessey
  - Type safety in distributed system w/ partial trust
- Chugh et al.
  - Dynamically loading untrusted JavaScript
- Information flow: non-interference

Defining and Enforcing

Referential Security

Jed Liu   Andrew C. Myers

Cornell University
Department of Computer Science

λ_{persist}

Referential security goals

1. Ensure referential integrity
2. Prevent accidental persistence
3. Prevent storage attacks

$$
\begin{tikzcd}
\langle e_1, M_1 \rangle \arrow[r, "\alpha"'] \arrow[d, phantom, "\wr_{\phi,\alpha}"] & \langle e'_1, M'_1 \rangle \arrow[d, phantom, "\wr_{\phi',\alpha}"] \\
\langle e_2, M_2 \rangle \arrow[r, "*"] & \langle e'_2, M'_2 \rangle
\end{tikzcd}
$$