Lecture 11, April 22

ECS 235B, Foundations of Computer and Information Security

Spring Quarter 2026

Basic Security Theorem

- Analogue to Bell-LaPadula BST
- Define *secure*:
  - System meeting preconditions is secure
- Idea of theorem:
  - Begin in secure state
  - Apply transitions (rules)
  - Resulting system in secure state

Theorem

Let R be a rule, s be a state of a system, and s' be the state obtained by applying R to s. Let the system in state s satisfy Preconditions 1 and 2, and let O and O' be the set of objects in states s and s', respectively. Then:

1. If there is an object o' such that
   a) $o' \notin O$
   b) $o' \in O'$
   c) $O' = O \cup \{o'\}$
   d) $o'(AS) = \{u\}$ for some subject u
   e) $o'(SS) = \varnothing$

   then s' satisfies Preconditions 1 and 2.

Theorem

2. If there is an object $o \in O$ such that
   a) $o'(AS) = \{u\} \cup o(AS)$ for some subject u
   b) $o'(SS) = \varnothing$

   then s' satisfies Preconditions 1 and 2.
3. If there is an object $o \in O$ such that
   a) $o'(AS) = o(AS)$
   b) $o'(SS) = \{u\} \cup o(SS)$ for some subject u

   then s' satisfies Preconditions 1 and 2.

Theorem

4. If there is an object $x' \in O'$ such that:
   a) $x' \notin O$
   b) there is an object $o \in O$
   c) $x'(AS) = o(AS)$
   d) $x'(SS) = o(SS)$

   then s' satisfies Preconditions 1 and 2.

Proof (First Case Only)

- s satisfies Preconditions 1 and 2
- For each $o \in O$, $o(AS)$ identifies all users who created or modified o
- For each $o \in O$, $o(SS)$ identifies all users who approve o
- $o' \notin O$ but $o' \in O' \Rightarrow o'$ created
  - Let u be the creator

Proof (*con't*)

- $o'(AS) = \{u\}$
  - $o'(AS)$ contains user who created o'
- $o'(AS)$ identifies all users who created, modified o', satisfying precondition 1
- $o'(SS) = \varnothing$
  - o' just created, so no-one yet approves its contents
- $o'(SS)$ identifies all users who approved it, satisfying precondition 2

Naming

- How do you identify authors, signers?
  - Important as if two have the same name, you lose accountability
- Leads to *domain rule*: the authors contained in the author group shall be given unique names
  - Problem is understood, lots of approaches to solving it (X.509 certificate hierarchies, etc.)
  - Call these *fully qualified names* (*FQN*)

Authorship Integrity

- Definition of terms
  - *domain* collection of systems
  - *subdomain* an inferior domain
  - *parent domain* a superior domain

  Each domain has its own administrative authority

Note: theorems hold as long as signers use FQNs

Goal: Record Information

An object o is *recorded* when

1. $o(AS) \subseteq o(SS)$; and

2. the recorder's office executes a recordation transformation on the object.

Designated repository: stores a copy of every recorded object in its domain.

Review Requirements

1. A signed document cannot be altered (although new signatures may be appended);
    - See alteration rule
2. A document may require multiple signatures;
    - See signature rule
3. A document submitted to the recorder's office may be revoked by any signatory until the document is recorded, but is no longer eligible for additional signatures;
    - See alteration rule
    - Definition of recorder's transformation

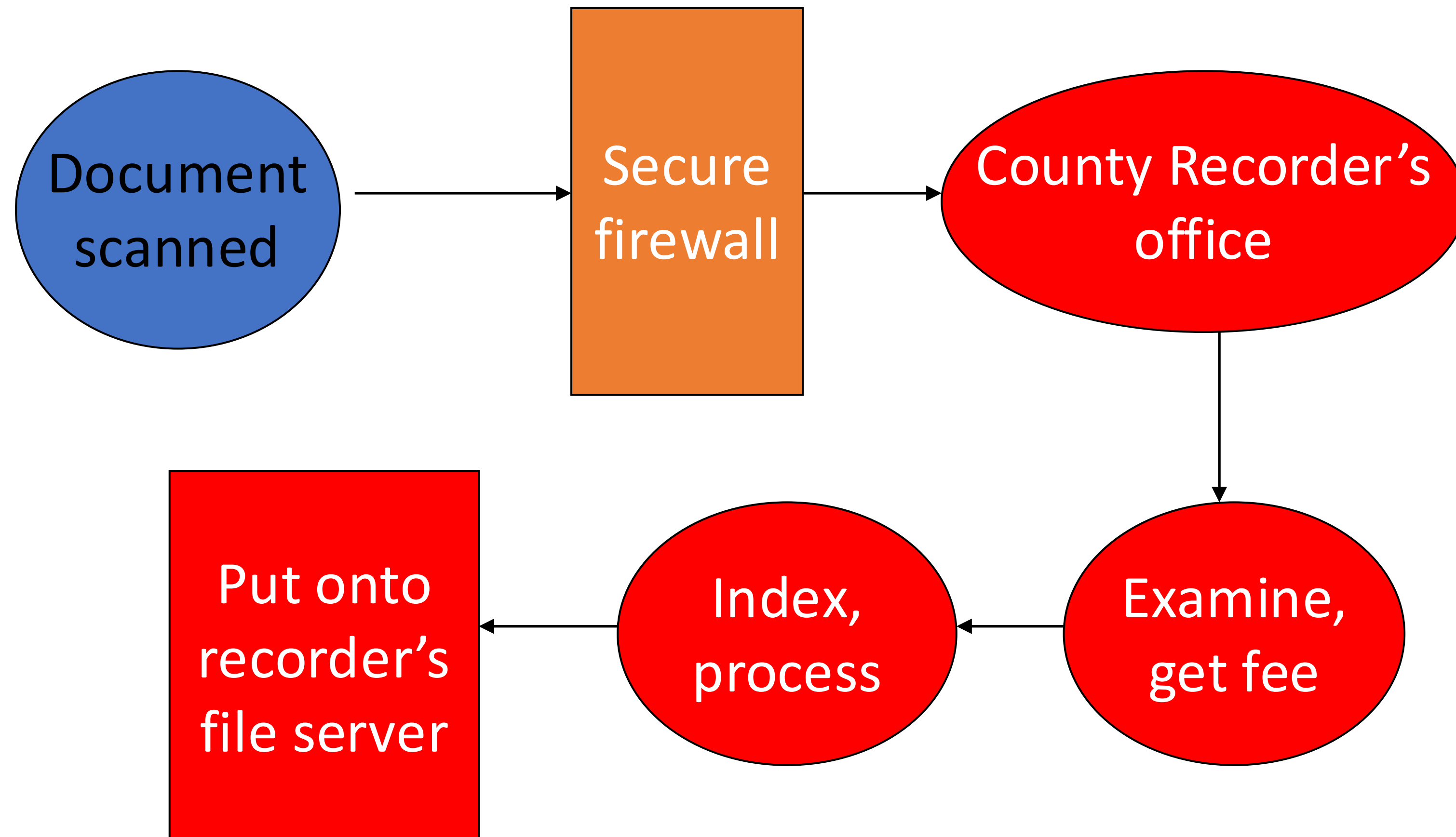
Review Requirements

4. The recorder may only append information to the document (*i.e.*, sign it); and
5. If the document is recorded, it becomes a public record immutable to all parties.
    - Definition of *recorder's transformation*

Now What?

- Can identify characteristics of a solution
  - If designing a solution, it must have those characteristics
- Know what to look for on a claimed solution

Basic Approach In Use



Assumptions

- Trusted relationship between author of images and recording authority
  - Encryption, acknowledgements
  - NB: Acknowledgement is “standard form wherein the author of the image acknowledges in writing that the documents submitted have original seals and signatures”

Submission of Documents

- How do you know the document received was the same as the one intended to be recorded?
  - Threat: I change the document in transit, before, or after it was sent
  - Digital signature assures document unchanged since signed and binds document to a public key
  - Public key infrastructure (PKI) binds public keys to principles (users)

Questions

- Is the user signing lawfully authorized to sign?
  - Albert di Salvo gets a real estate license …
- Is the user requesting the signature the one authorized to request the signature?
  - Sharing passwords, sharing a system … spoofing
- Is document changed between the user requesting the signature and the document being signed?
  - Virus-like programs change it first (use Adobe Photoshop-like program to change stamps, for example), unbeknownst to the user

More Questions

- Is the right public key used to sign the document?
  - PKI assumes certificates, binding keys to users, are issued to the right people
- Did the submitter change the document without the other party’s consent?
  - On paper, this can usually be detected
  - Electronically, no way, unless original document digitally signed (see above)

Validation and Storage

- Document arrives at server
  - Stored in one area; validated here
  - When recorded, moved to permanent area
    - Burned onto CD or some other WORM media
    - Also saved to a vault in the Sierra Nevada
- Operating system, web servers, other supporting applications provide security

Questions

- What is the system connected to?
  - Where can attackers come from?
- How well will the operating system withstand penetration attempts?
  - Lots of vulnerabilities in all software, OSes
- What operational security procedures are in place to maintain the security?
  - Bad procedures can weaken the best system
  - Who installs security patches, keeps up to date with new attacks, holes?

More Questions

- Is digital signature stored with document?
  - On the validation server
    - If not, it can be changed there
  - On the archive server
    - If not, no way to revalidate that document was same as sent

Return Documents

(Read this as retrieval of documents)

- Someone requests a title or copies of liens
    - Retrieval system gets it and presents it

Questions

- How do you know it gets the right one?

  Example: three documents about your house
  - The first (real) one says you have paid off all liens on your house.
  - The second (bogus) one puts a lien on your house.
  - The third (bogus) one forecloses on your house.
  - Which one is returned?

Solving the Problem

- AB 578 directed CA Attorney General to establish standards for electronic recordation systems
    - Includes security testing, and *explicitly* requires penetration testing
    - May be the first law to do so!
- National efforts under way, too

The Problem With Solutions

- Vendor: “This system is designed and built using standard industrial software engineering techniques”
- Customer: “We installed and run this following the vendor’s instructions”

During a penetration test:

- Took 5 minutes to gain illicit, unauthorized access to system
- Took 10 minutes to compromise system’s functioning so it reported incorrect results
- Took 20 minutes to find all “hidden” passwords embedded in programs

Moral: current software and systems are not secure!

Break-the-Glass Policies

- Motivation: when security requirements conflict, some access controls may need to be overwritten in an unpredictable manner
  - Example: a doctor may need access to a medical record to treat someone, yet that person is unable to give consent (without which access would be denied)
- User overrides the denial
  - Controls notify some people about the override
  - Controls log override for later audit

Example: Rumpole

- Implements a break-the-glass policy
- *Evidential rules*: how to assemble evidence to create context for request
- *Break-glass rules*: define permissions
  - Includes constraints such as obligations to justify need for actions
- *Grant policies*: how rules are combined to determine whether to grant override

Example: Rumpole Enforcement Model

- *Request*: subject, desired action, resource, obligations acceptable to subject
- Decision point:
  - Grants request
  - Denies request
  - Returns request with set of obligations subject must accept; subject then can send a new request with that set of obligations, if they are acceptable

Composition of Policies

- Two organizations have two security policies
- They merge
  - How do they combine security policies to create one security policy?
  - Can they create a coherent, consistent security policy?

The Problem

- Single system with 2 users
    - Each has own virtual machine
    - Holly at system high, Lara at system low so they cannot communicate directly
- CPU shared between VMs based on load
    - Forms a *covert channel* through which Holly, Lara can communicate

Example Protocol

- Holly, Lara agree:
  - Begin at noon
  - Lara will sample CPU utilization every minute
  - To send 1 bit, Holly runs program
    - Raises CPU utilization to over 60%
  - To send 0 bit, Holly does not run program
    - CPU utilization will be under 40%
- Not “writing” in traditional sense
  - But information flows from Holly to Lara

Policy vs. Mechanism

- Can be hard to separate these
- In the abstract: CPU forms channel along which information can be transmitted
  - Violates *-property
  - Not “writing” in traditional sense
- Conclusion:
  - Bell-LaPadula model does not give sufficient conditions to prevent communication, *or*
  - System is improperly abstracted; need a better definition of “writing”

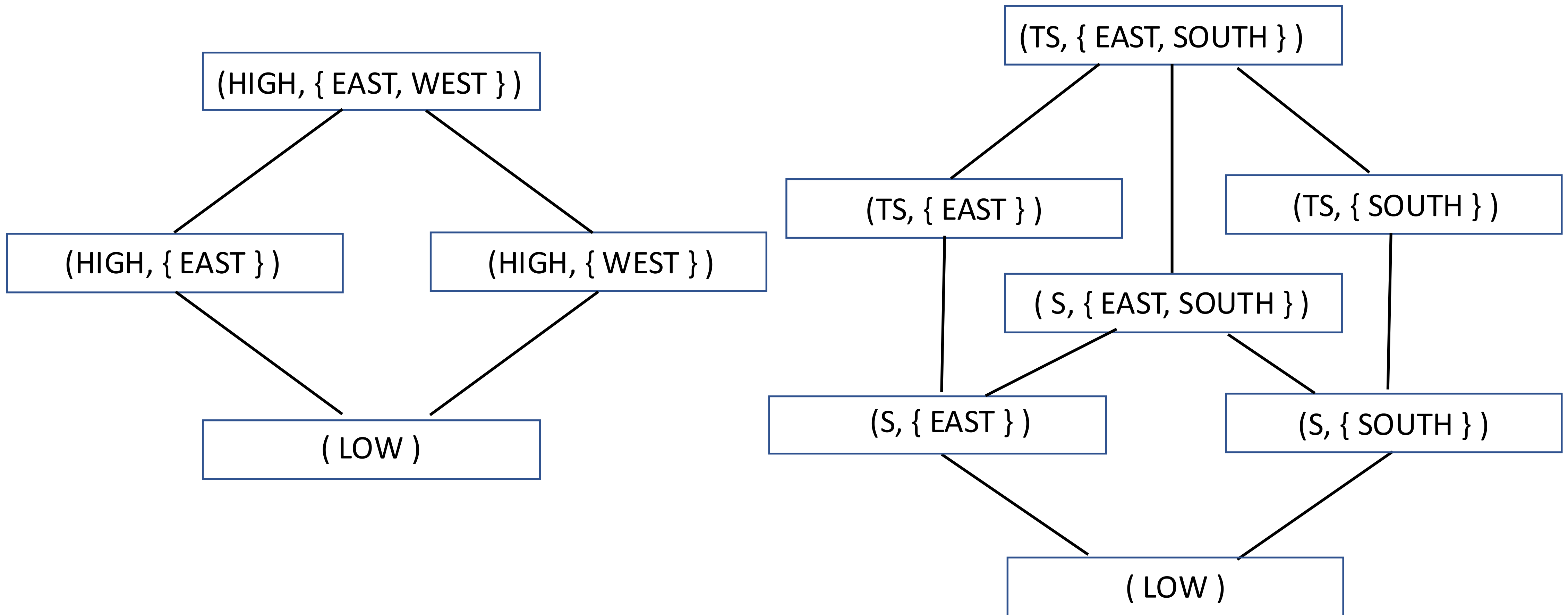
Composition of Bell-LaPadula

- Why?
  - Some standards require secure components to be connected to form secure (distributed, networked) system
- Question
  - Under what conditions is this secure?
- Assumptions
  - Implementation of systems precise with respect to each system's security policy

Issues

- Compose the lattices
- What is relationship among labels?
  - If the same, trivial
  - If different, new lattice must reflect the relationships among the levels

Example



Analysis

- Assume S < HIGH < TS
- Assume SOUTH, EAST, WEST different
- Resulting lattice has:
  - 4 clearances (LOW < S < HIGH < TS)
  - 3 categories (SOUTH, EAST, WEST)

Same Policies

- If we can change policies that components must meet, composition is trivial (as above)
- If we *cannot*, we must show composition meets the same policy as that of components; this can be very hard

Different Policies

- What does “secure” now mean?
- Which policy (components) dominates?
- Possible principles:
  - Any access allowed by policy of a component must be allowed by composition of components (*autonomy*)
  - Any access forbidden by policy of a component must be forbidden by composition of components (*security*)

Implications

- Composite system satisfies security policy of components as components' policies take precedence
- If something neither allowed nor forbidden by principles, then:
  - Allow it (Gong & Qian)
  - Disallow it (Fail-Safe Defaults)

Example

- System X: Bob can’t access Alice’s files
- System Y: Eve, Lilith can access each other’s files
- Composition policy:
  - Bob can access Eve’s files
  - Lilith can access Alice’s files
- Question: can Bob access Lilith’s files?

Solution (Gong & Qian)

- Notation:
  - (a, b): a can read b's files
  - AS(x): access set of system x
- Set-up:
  - AS(X) = $\varnothing$
  - AS(Y) = { (Eve, Lilith), (Lilith, Eve) }
  - AS(X$\cup$Y) = { (Bob, Eve), (Lilith, Alice), (Eve, Lilith), (Lilith, Eve) }

Solution (Gong & Qian)

- Compute transitive closure of AS($X \cup Y$):
  - AS($X \cup Y$)$^+$ = { (Bob, Eve), (Bob, Lilith), (Bob, Alice), (Eve, Lilith), (Eve, Alice), (Lilith, Eve), (Lilith, Alice) }
- Delete accesses conflicting with policies of components:
  - Delete (Bob, Alice)
- (Bob, Lilith) in set, so Bob can access Lilith's files

Idea

- Composition of policies allows accesses not mentioned by original policies
- Generate all possible allowed accesses
  - Computation of transitive closure
- Eliminate forbidden accesses
  - Removal of accesses disallowed by individual access policies
- Everything else is allowed
- Note: determining if access allowed is of polynomial complexity

Interference

- Think of it as something used in communication
  - Holly/Lara example: Holly interferes with the CPU utilization, and Lara detects it — communication
- Plays role of writing (interfering) and reading (detecting the interference)

Model

- System as state machine
  - Subjects $S = \{ s_i \}$
  - States $\Sigma = \{ \sigma_i \}$
  - Outputs $O = \{ o_i \}$
  - Commands $Z = \{ z_i \}$
  - State transition commands $C = S \times Z$
- Note: no inputs
  - Encode either as selection of commands or in state transition commands

Functions

- State transition function T: $C \times \Sigma \rightarrow \Sigma$
  - Describes effect of executing command c in state σ
- Output function P: $C \times \Sigma \rightarrow O$
  - Output of machine when executing command c in state σ
- Initial state is σ_0

Example: 2-Bit Machine

- Users Heidi (high), Lucy (low)
- 2 bits of state, *H* (high) and *L* (low)
  - System state is (*H*, *L*) where *H*, *L* are 0, 1
- 2 commands: *xor0*, *xor1* do *xor* with 0, 1
  - Operations affect *both* state bits regardless of whether Heidi or Lucy issues it

Example: 2-bit Machine

- S = { Heidi, Lucy }
- Σ = { (0,0), (0,1), (1,0), (1,1) }
- C = { *xor0*, *xor1* }

	Input States (H, L)			
	(0,0)	(0,1)	(1,0)	(1,1)
xor0	(0,0)	(0,1)	(1,0)	(1,1)
xor1	(1,1)	(1,0)	(0,1)	(0,0)

Outputs and States

- T is inductive in first argument, as

  $T(c_0, \sigma_0) = \sigma_1$; $T(c_{i+1}, \sigma_{i+1}) = T(c_{i+1}, \mathrm{T}(c_i, \sigma_i))$
- Let C^* be set of possible sequences of commands in C
- $T^*: C^* \times \Sigma \rightarrow \Sigma$ and

  $c_s = c_0 \ldots c_n \Rightarrow T^*(c_s, \sigma_i) = T(c_n, \ldots, T(c_0, \sigma_i) \ldots)$
- P similar; define $P^*: C^* \times \Sigma \rightarrow O$ similarly

Projection

- $T^*(c_s, \sigma_i)$ sequence of state transitions
- $P^*(c_s, \sigma_i)$ corresponding outputs
- $proj(s, c_s, \sigma_i)$ set of outputs in $P^*(c_s, \sigma_i)$ that subject *s* authorized to see
  - In same order as they occur in $P^*(c_s, \sigma_i)$
  - Projection of outputs for *s*
- Intuition: list of outputs after removing outputs that *s* cannot see

Purge

- $G \subseteq S$, G a group of subjects
- $A \subseteq Z$, A a set of commands
- $\pi_G(c_s)$ subsequence of c_s with all elements (s,z), $s \in G$ deleted
- $\pi_A(c_s)$ subsequence of c_s with all elements (s,z), $z \in A$ deleted
- $\pi_{G,A}(c_s)$ subsequence of c_s with all elements (s,z), $s \in G$ and $z \in A$ deleted

Example: 2-bit Machine

- Let σ_0 = (0,1)
- 3 commands applied:
  - Heidi applies *xor0*
  - Lucy applies *xor1*
  - Heidi applies *xor1*
- c_s = ((Heidi, *xor0*), (Lucy, *xor1*), (Heidi, *xor1*))
- Output is 011001
  - Shorthand for sequence (0,1) (1,0) (0,1)

Example

- *proj*(Heidi, c_s, σ_0) = 011001
- *proj*(Lucy, c_s, σ_0) = 101
- $\pi_{Lucy}(c_s)$ = (Heidi, *xor0*), (Heidi, *xor1*)
- $\pi_{Lucy,xor1}(c_s)$ = (Heidi, *xor0*), (Heidi, *xor1*)
- $\pi_{Heidi}(c_s)$ = (Lucy, *xor1*)
- $\pi_{Lucy,xor0}(c_s)$ = (Heidi, *xor0*), (Lucy, *xor1*), (Heidi, *xor1*)
- $\pi_{Heidi,xor0}(c_s) = \pi_{xor0}(c_s)$ = (Lucy, *xor1*), (Heidi, *xor1*)
- $\pi_{Heidi,xor1}(c_s)$ = (Heidi, *xor0*), (Lucy, *xor1*)
- $\pi_{xor1}(c_s)$ = (Heidi, *xor0*)

Noninterference

- Intuition: If set of outputs Lucy can see corresponds to set of inputs she can see, there is no interference
- Formally: $G, G' \subseteq S$, $G \neq G'$; $A \subseteq Z$; users in G executing commands in A are *noninterfering* with users in G' iff for all $c_s \in C^*$, and for all $s \in G'$,

$$proj(s, c_s, \sigma_i) = proj(s, \pi_{G,A}(c_s), \sigma_i)$$

  - Written $A,G :| G'$