DSCI 519: Foundations and Policy for Information Security

Theoretical Limits on System Security

Tatyana Ryutov

Reminders

- Complete Quiz 2 to by September 25, midnight
  - 45 minutes
  - Covers lectures 3 and 4
- Lab2 is posted, due by October 23, midnight
- Project proposal is due by October 2nd, midnight
  - If you decide to do the assigned project, you need to submit nothing

Outline

- Review
- Bell-LaPadula Multics interpretation
- General undecidability of security
  - Preliminaries
    - Halting problem
    - Turing machines
    - Safety question
  - Harrison-Ruzzo-Ullman result

L5.Q6

- Think about all the material covered in lectures 1-5
- Indicate specific topics/questions for midterm review next lecture
  - focused questions/topics I can give you an answer to
- If everything is clear, it's OK to say: "none"

Presentation 3

Multiple Independent Levels of Security Architecture (MILS)

Ashley Ma & Miliano Mikol

Presentation 4

Your Muddy Points

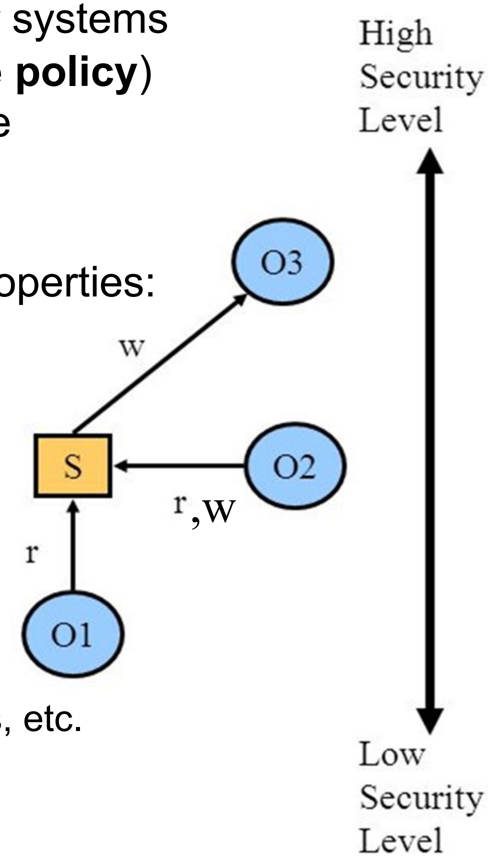
- What roles traditionally or formally are responsible for security mathematical formulas? Are the formulas derived during design/threat-modeling and then verified dynamically during integration and deployment?
- BLP's communication down solution - if a Trojan horse is embedded into the system while the subject has maximum security level, then it can copy data and when the security level downgrades, the Trojan can create a new document with lower security level with the previously copied data which had higher security clearance
- Not sure about how deep about basic security theorem we should understand. Just the concept or we should also understand the proof?
- Lattices, LUB and GLB
- BLP abstraction for system state

Outline

- Review
- Bell-LaPadula Multics Interpretation
- General undecidability of security
  - Preliminaries
    - Halting problem
    - Turing machines
    - Safety question
  - Harrison-Ruzzo-Ullman result

Review: BLP

- BLP a useful particularization of a class of systems
- The goal is to enforce confidentiality policy
- Methodology
  - Define an abstract **model** that can be used to describe computer systems
  - Define what it means for a system in the model to be secure (**the policy**)
  - Develop techniques to prove that a system in the model is secure
- Approach:
  - Use state-transition systems to describe a computer system
  - Define system as secure IFF every reachable state satisfies 3 properties:
    1. simple-security property: S can read O IFF S *dom* O
    2. *-property: S can write O IFF O *dom* S
    3. discretionary-security property
  - Prove a Basic Security Theorem (BST)
- Main contributions:
  - The overall methodology to show that a system is secure
    - adopted in many later works
  - The state-transition model
    - which includes an access matrix, subject security levels, object levels, etc.
  - The introduction of *-property
    - ss-property is not enough to stop illegal information flow



USC Viterbi School of Engineering

Review: BLP Abstraction for System State

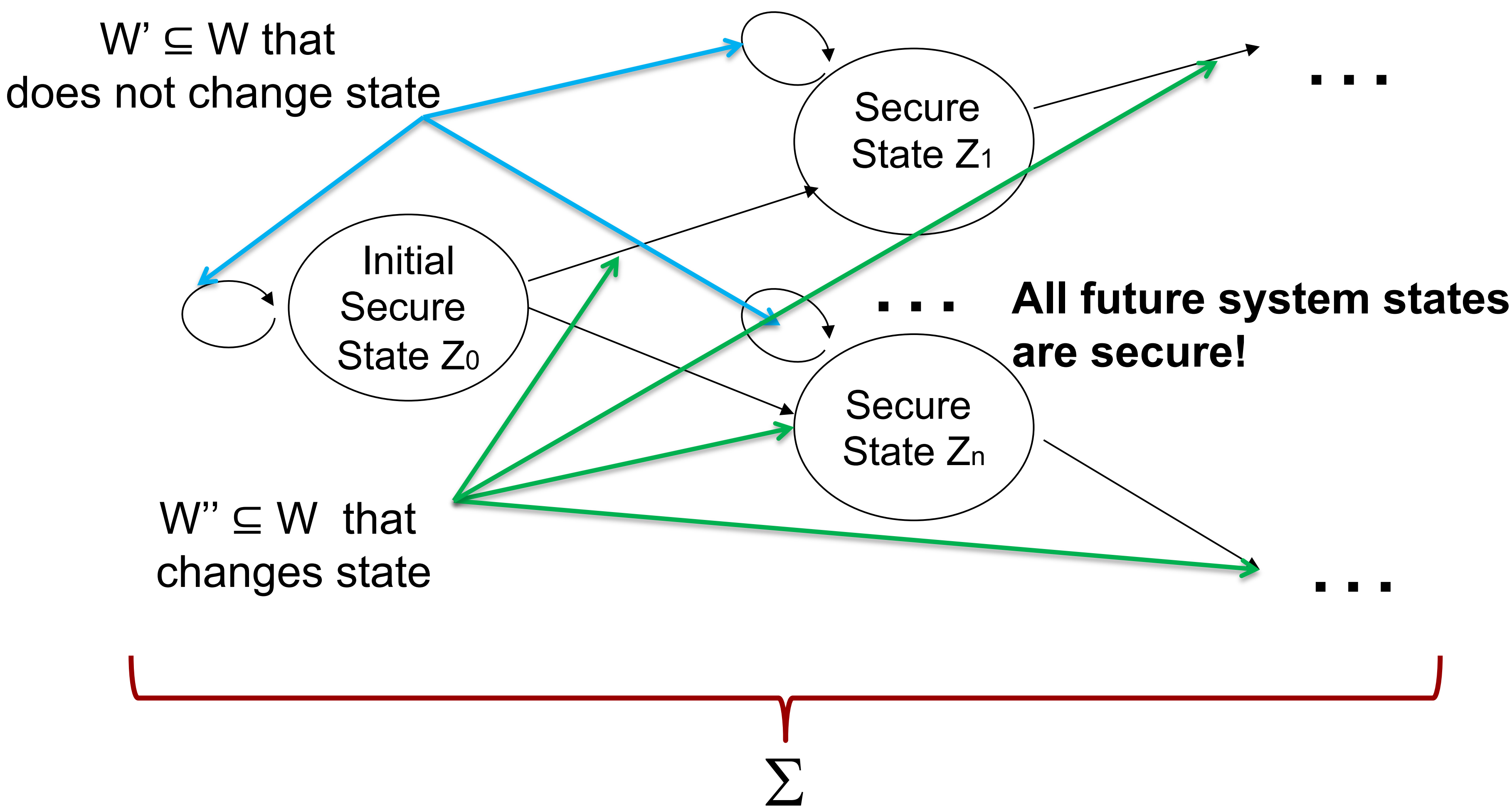
- The **state** of the system is (b, M, f, H) where:
  - **b** indicates which subjects can access which objects
    - The set of rights that may actually be exercised in the **current** state
  - **M** is the access control matrix for the current state
    - DAC
    - MAC can make some rights unusable
  - **f** is tuple indicating subject and object access classes
    - MAC
    - For subjects: max and current clearances
  - **H** is the *hierarchy* of objects (for naming objects)
    - Can not do access control unless can uniquely name objects
- Recall that we have defined a "secure" state
  - MAC (Simple security, *-property) and DAC

b is a subset of all accesses that a subject can have in the system based on DAC (M) and MAC, why here it is subset but not the exactly the same set?

Review: System Representation

- System is represented by $\Sigma(R, D, W, z_0)$ where:
  - R denotes the set of requests for access (inputs)
  - D denotes the set of outcomes (outputs)
    - (y)es, (n)o, (i)llegal, (o) error
  - W is the set of actions of the system
    - At any time *t* the system is in state $v \in V$, where V is set of all system states
      - Recall that the v = (b, m, f, h)
    - W moves system from a state in V to another (possibly different) state in V
      - Each action is of the form ((r,d,(b,m,f,h), (b',m',f',h'))
    - Example of W – kernel calls
  - z_0 is the initial state of the system
- System is all sequences of <request, decision, state> triples, with initial state z_0

Review: BLP as a State Machine System View

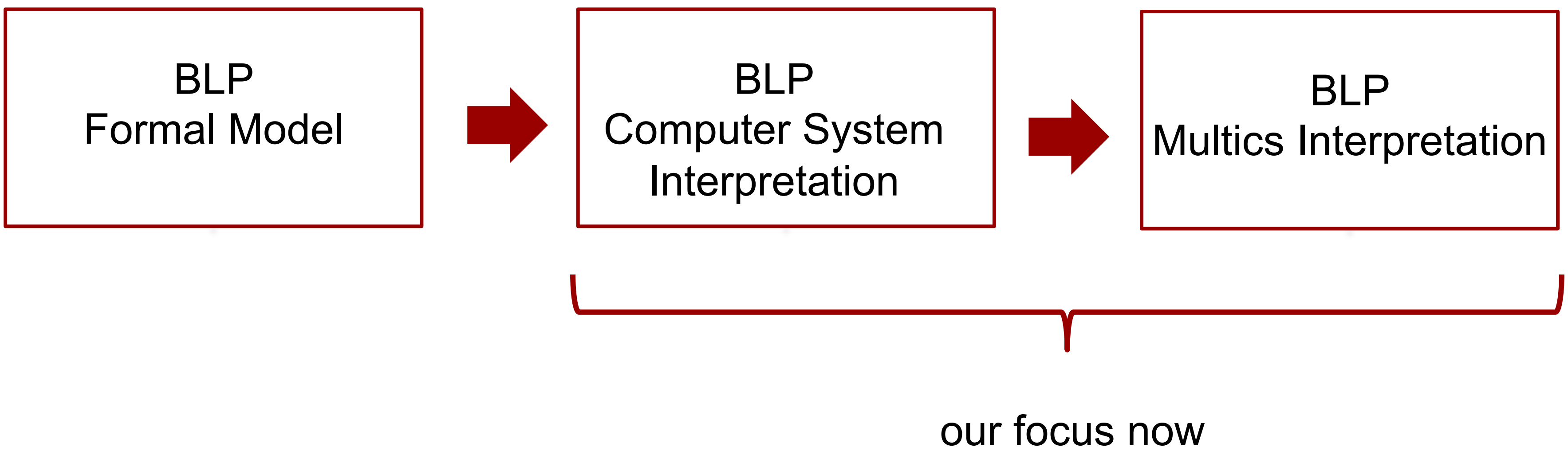


- Define **all possible** transitions (W), each transition satisfies the 3 properties

W is described as the set of actions of the system. Does it mean transitions between states?

USC Viterbi School of Engineering

BLP Interpretation



MULTICS System

- Multiplexed Information and Computing Service (MULTICS)
  - Joint project by MIT and Bell Telephone Laboratories
  - Extraordinarily influential early time-sharing operating system
    - Unix, Linux, Paging, Segmentation, Protection, etc. had their birth in this project
- MULTICS was commercialized 1973-1985, in use until 2000
  - Honeywell commercial product beginning ~ 1970
- Designed for security from outset
  - But repeatedly broken into: penetrate & patch
    - Current industry "best practice"
  - Eventually led to redesign based on FSPM and RM
- Proof of performance/usability of secure systems

USC Viterbi School of Engineering

Multics Correspondence to BLP State

b $\rightarrow$ segment descriptor words (SDW)

M $\rightarrow$ access control lists (ACL)

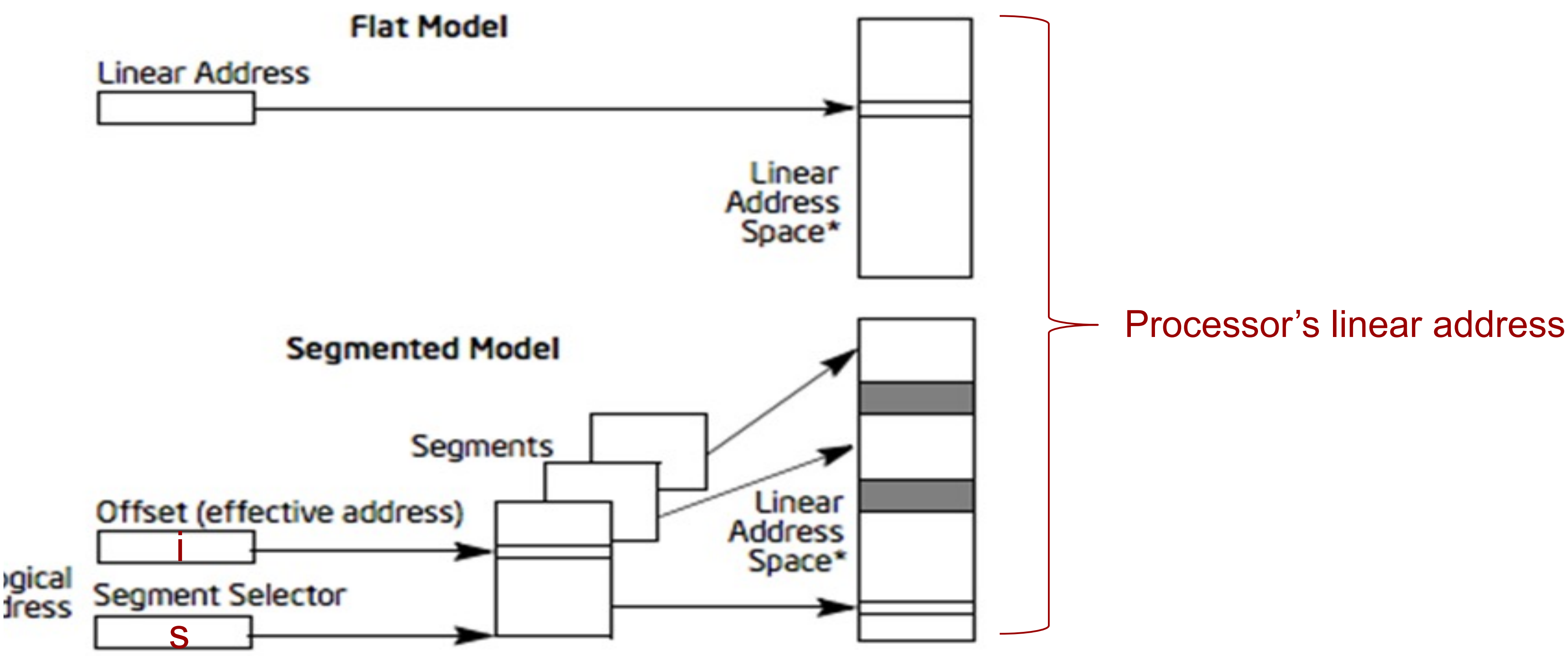
f $\rightarrow$ $\begin{cases} \text{segment security level in directory} \\ \text{process security level tables} \end{cases}$

H $\rightarrow$ branches in a directory

USC Viterbi School of Engineering

Sidebar: Flat vs. Segmented Memory

- Segmented memory model: memory appears to a program as a group of independent address spaces called **segments**

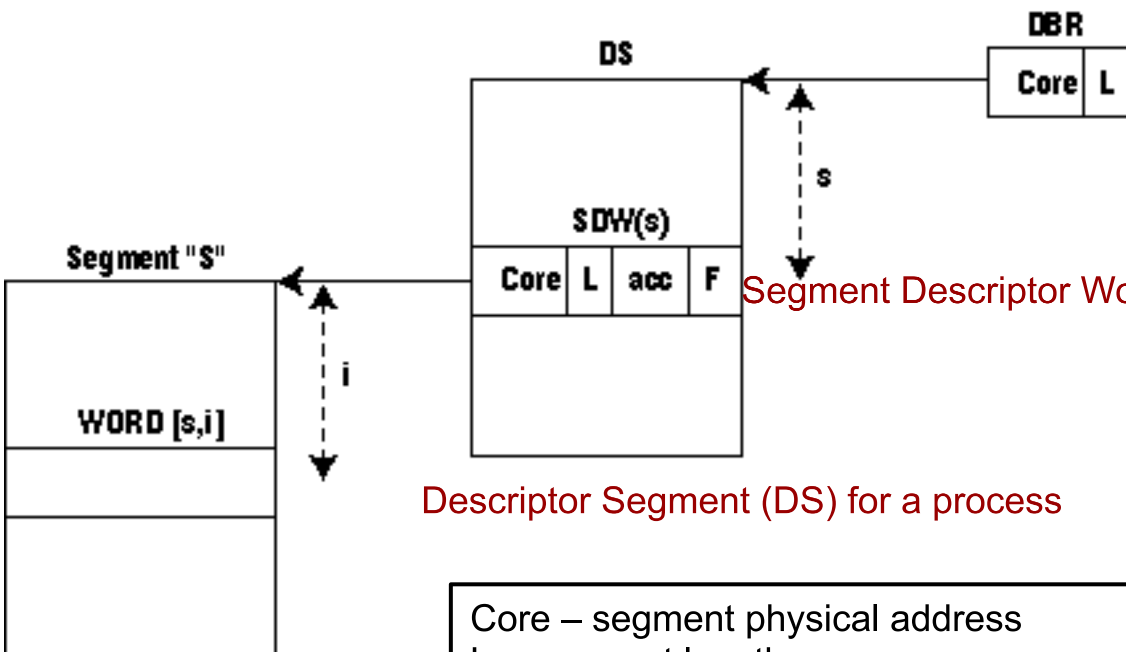


Virtual Address of a segment = [s, i]

USC Viterbi School of Engineering

Multics Hardware Memory Referencing

- All code, data, I/O devices, etc. that may be accessed by a process are stored as segments
- A segment virtual address consists of a pair of integers [s, i]
  - "s" is called the segment number (descriptor)
  - "i" the index within the segment (offset)
- Descriptor Segment (DS) stores Segment Descriptor Words (SDW) that reference each of the process's active segments



Descriptor Base Register (reference to descriptor segment of a process)

Segment Descriptor Word (SDW)

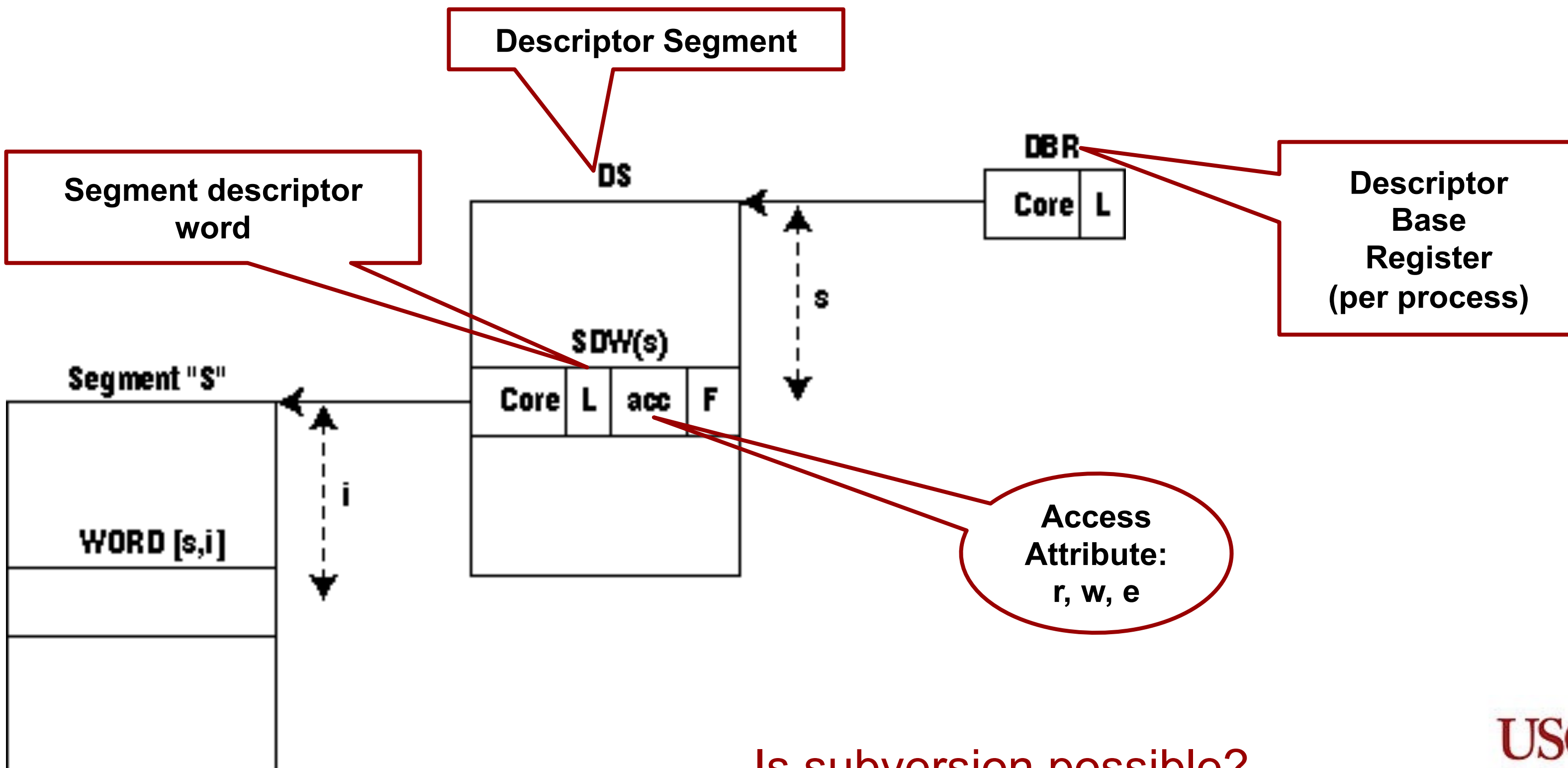
Descriptor Segment (DS) for a process

Segment in main memory

Core – segment physical address
L – segment length
F – flags
acc – segment access permissions (read, write, execute)

State for Hardware Current Access (b)

- The set **b** is interpreted as a descriptor segment (DS)
- The set **b** has elements (S, O, p) – interpreted as a SDW
  - Subject S has current access to object O in mode p
- Each subject S is a distinct process
- Each distinct data segment is an object O



Is subversion possible?

Multics Correspondence to BLP State

b $\rightarrow$ segment descriptor words (SDW)

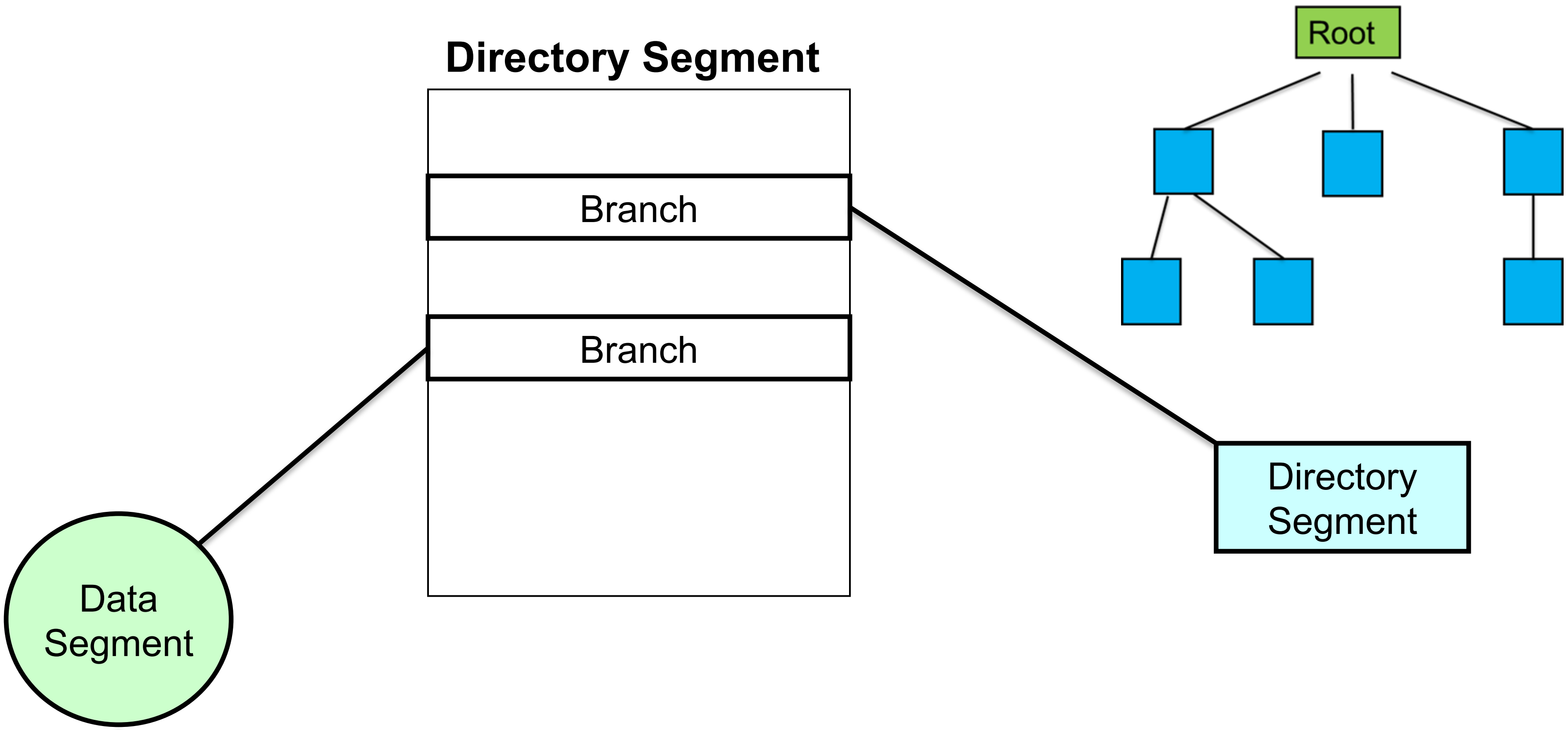
M $\rightarrow$ access control lists (ACL)

$$f \rightarrow \begin{cases} \text{segment security level in directory} \\ \text{process security level tables} \end{cases}$$

H $\rightarrow$ branches in a directory

USC Viterbi School of Engineering

Multics Hierarchy Interpretation



Relationship H between objects:
Both data and directory segments stored in a “branch” of directory

Multics Correspondence to BLP State

b $\rightarrow$ segment descriptor words (SDW)

M $\rightarrow$ access control lists (ACL)

f $\rightarrow$ $\begin{cases} \text{segment security level in directory} \\ \text{process security level tables} \end{cases}$

H $\rightarrow$ branches in a directory

USC Viterbi School of Engineering

ACL: Interpret Access Matrix

Objects / Subject	Segment 1	Segment2	Segment 3	Segment4
Bob Process	read	read, write		write
Flo Process	read, write	write		
Alice Process	read	read	read	read, write
Dan Process	read		read, write	read

ACL for Segment 1:

Bob	read
Flo	read, write
Alice	read
Dan	read

Access Control List (ACL):

- A column of Matrix (M)
- Stored in a “branch” of directory
- Branch contains physical location and ACL for the object

Multics Correspondence to BLP State

b $\rightarrow$ segment descriptor words (SDW)

M $\rightarrow$ access control lists (ACL)

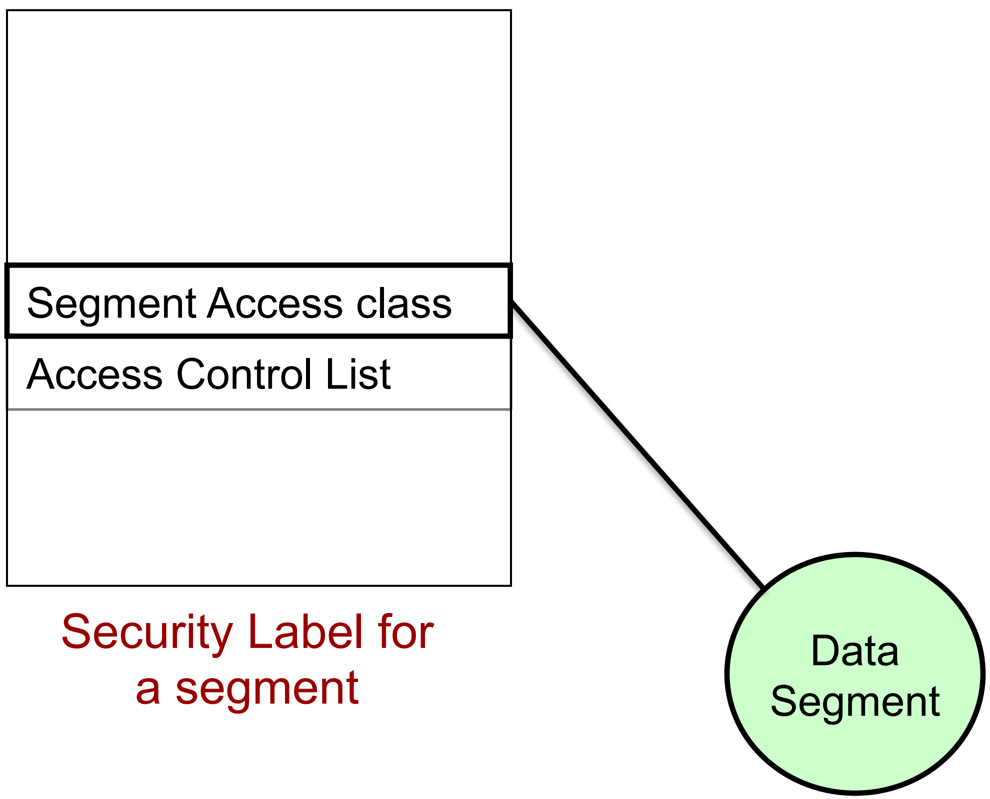
f $\rightarrow$ segment security level in directory
process security level tables

H $\rightarrow$ branches in a directory

Interpret Security Level Function (f)

- Segment Access Class and ACL are stored in parent directory of the segment
- Process Access Class is stored in TCB (internal kernel table)

Directory Segment



Security Label for a segment

TCB internal table

Bob	Top Secret
Flo	Top Secret
Alice	Unclassified
Dan	Unclassified

Security Label for a process

USC Viterbi School of Engineering

Interpretations of RM Access

- Use access modes to describe access control
  - Must interpret access mode
- Fundamentally, access devolves to read & write
  - Memory chips have only read and write functionality
- Other derivative modes ultimately map to that

Model	**Multics access rights**
r $\rightarrow$	read
r,e $\rightarrow$	execute
w $\rightarrow$	read and write
a $\rightarrow$	write

Recall: BLP System Representation

- Represented by an initial state and a **sequence** of request, decisions, and states
- In formal terms, system $\Sigma(R, D, W, z_0)$ where:
  - R denotes the set of requests for access
  - D denotes the set of outcomes
  - W is the set of actions of the system
  - z_0 is the initial state of the system
- W notation in formal terms:
  - $W \subseteq R \times D \times V \times V$ is the set of actions of system
    - System moves from a state in V to another (possibly different) state in V
    - Usually “yes “means state change, but if segment is already in b, no change in state
  - Each action is of the form ((r,d, (b,m,f,h), (b’,m’,f’,h’))
    - Recall that the set (b,m,f,h) is a state of the system

USC Viterbi School of Engineering

Basic Security Theorem

- $\Sigma(R, D, W, z_0)$ is a secure system if
  - z_0 is a secure state
  - Every action (r,d,(b,m,f,h), (b',m',f',h')) in W satisfies:

new state

old state

a. Every $(s,o,p) \in b - b'$
   - simple security condition
   - *-property
   - ds-property

i.e., any **new** accesses in b must satisfy the policy

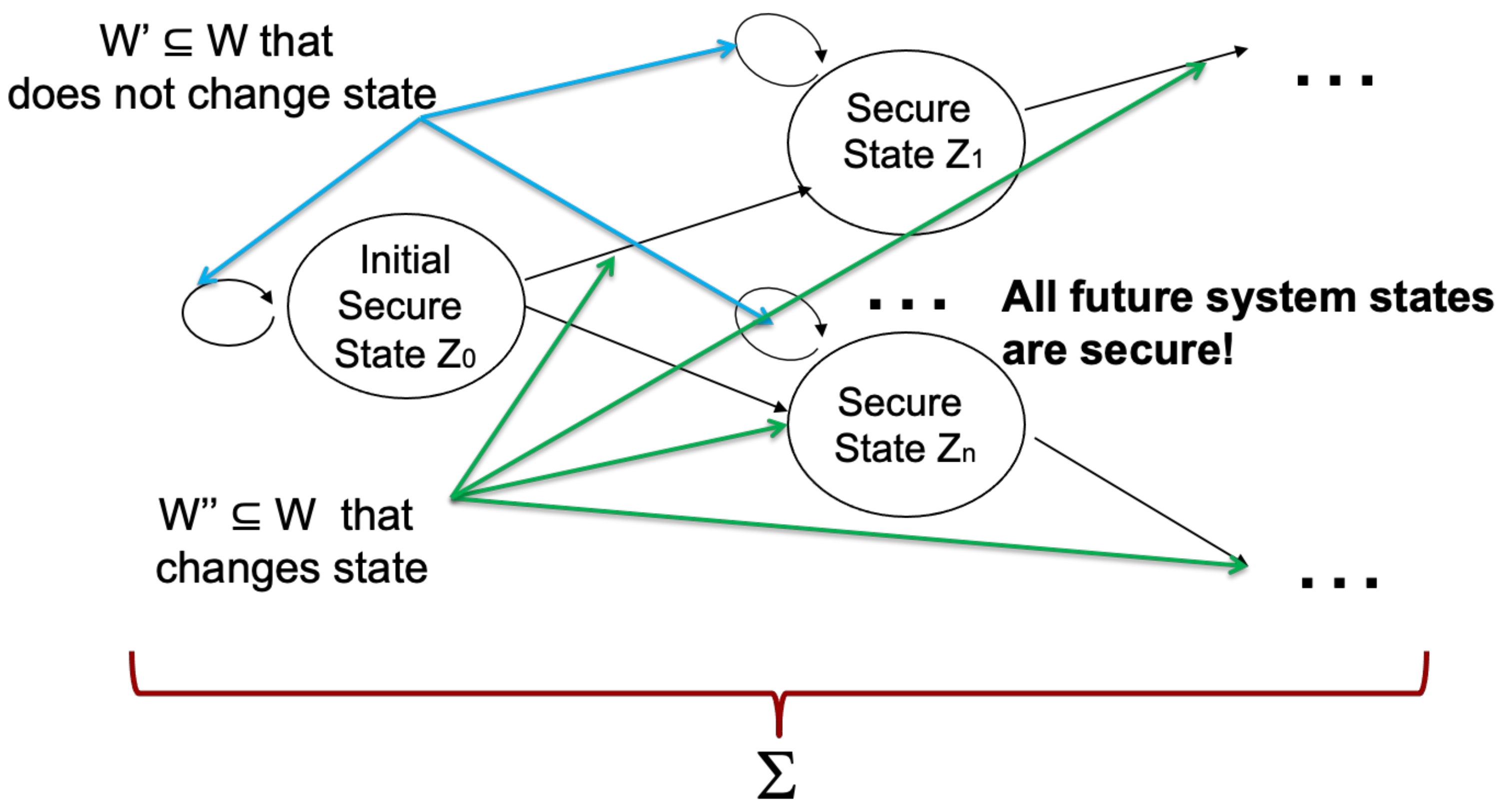
b. Every $(s,o,p) \in b'$ not satisfying the following, not in b
   - simple security condition
   - *-property
   - ds-property

i.e., must revoke **old** accesses that would no longer satisfy the policy (due to changes in subject level, or DAC permissions)

USC Viterbi School of Engineering

What is missing?

- Given a system, can you show that **every** action (usually means kernel calls and their effects) maps to an FSPM transition (a member of set W)?



- Define **all possible** transitions (W), each transition satisfies the 3 properties

General Mechanism of Model

- "Inductive nature" of security
  - Each alteration of state does not breach security
  - If preserved from one state to next, system is secure
  - Total system security guaranteed systematically
    - For every access tuple (s,o,p) added to current access set b
    - Meets ss, *-property, and ds-property
- Modular specification of system capabilities
  - Only one rule for each (request, current state) pair
- Goal: relate rule properties to system properties
  - Preservation of properties over one transition

USC Viterbi School of Engineering

Recall: Reference Monitor (RM) Functions

- Two classes
  - Reference and Authorization
- Reference functions
  - Subject wants to access an object – is it OK?
- Authorization functions
  - Change in authorization data base
    - E.g., add/remove user or access right

Reference Functions in BLP

- What state elements (b, M, f, H) do the RM Reference Functions use and change?

- Reference Functions consult b, M, and f before making changes
- Check b (enforcement in hardware), b acts as “cache” for granted access but in hardware
- If not authorized by b, check f to see if the MAC policy permits access
- If MAC permits, the last step before changing b is to consult M
- If access is granted, b will be changed

Authorization Functions in BLP

- What state elements (b, M, f, H) do Authorization Functions use and change?

- Authorization Functions consult b, M, f, and H before making changes
- Adding or deleting subjects and objects changes M, f (for subjects and objects) and H
- If a subject's current level changes, that changes f
- Granting or revoking discretionary access - changes M
- May change b under certain interpretations (e.g., remove revoked access if M or current level changed)

USC Viterbi School of Engineering

Four Classes of General Functions

- Four classes of functions that can change system state:
  1. Functions to alter the current access (set b)
  2. Functions to alter current access permission (M)
  3. Functions to alter the level functions (set f)
     - Level (subject)
     - Level (object)
     - Current-level (subject)
  4. Functions to alter object structure (hierarchy H)
- **Next**: instantiate model rules in kernel primitives
  – Use notation in BLP Multics Interpretation

USC Viterbi School of Engineering

Morphisms from Multics to Model

- Instantiate BLP rules as 11 **specific** kernel primitives
  - Rule numbers are from BLP model

1. Alter current access (set b) → SDW
   1. Kernel function: get-read
   2. Kernel function: get-write-only
   3. Kernel function: get-execute
   4. Kernel function: get-read-write
   5. Release read/execute/write

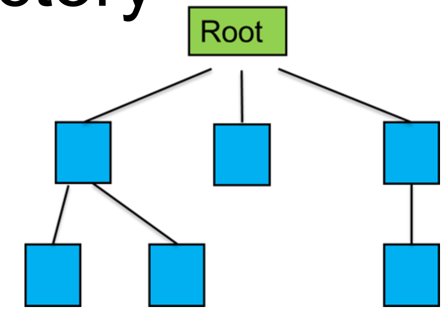
2. Alter access permission (matrix M) → ACL
   6. Give read/execute/write
   7. Revoke read/execute/write

A **morphism** is a structure-preserving mapping from one mathematical structure to another

USC Viterbi
School of Engineering

Morphisms from Multics to Model

3. Alter hierarchy (set H) $\rightarrow$ branch in directory
   8. Create-object
   9. Delete-object-group
4. Alter level function (set f)
   – For subject $\rightarrow$ current-security-level table
   10. Change-subject-current-security-level
   – For object $\rightarrow$ branch in directory
   11. Change-object-security-level



In practice, not used due to the tranquility principle

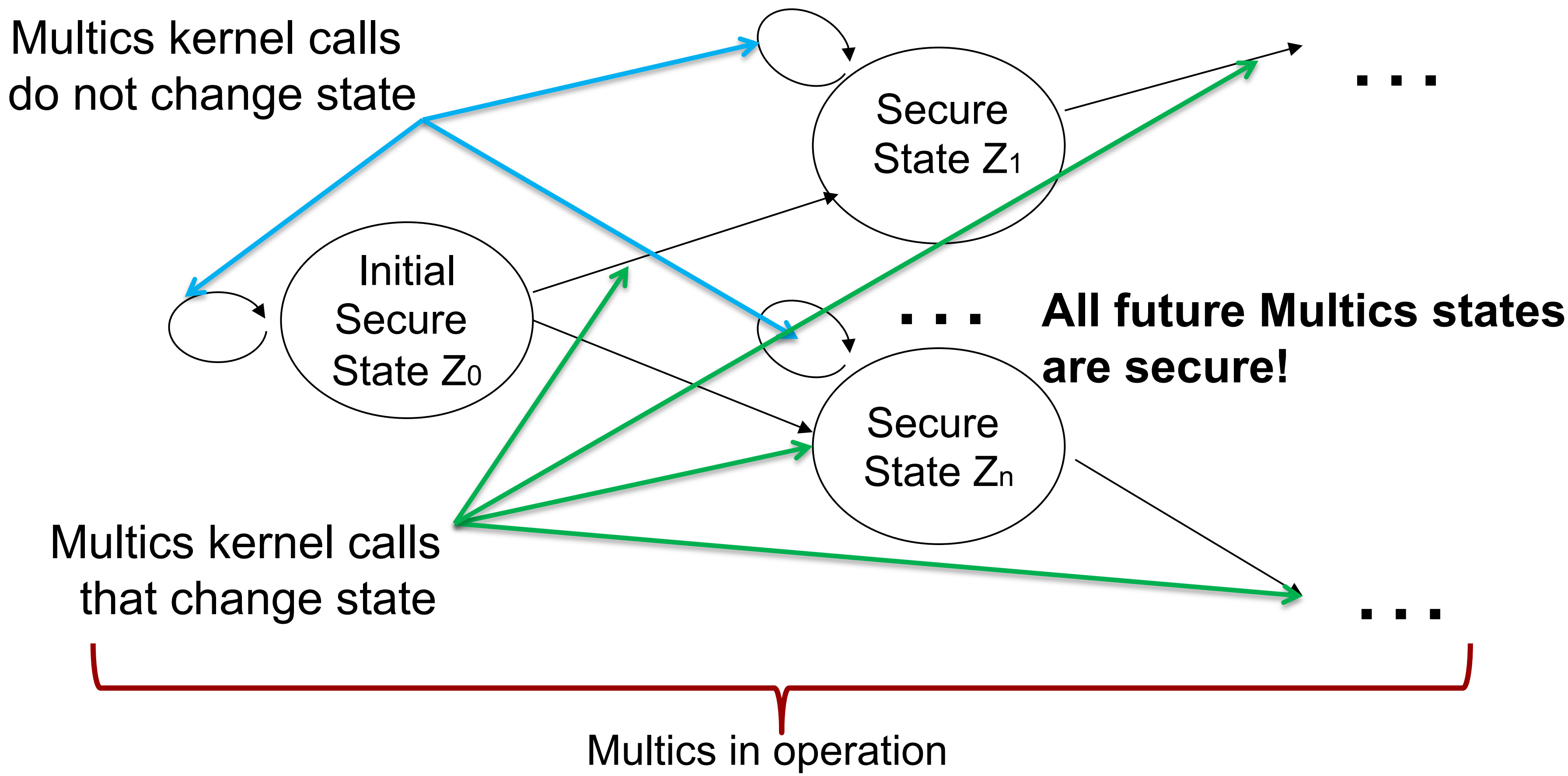
USC Viterbi School of Engineering

Paraphrased Multics BST

- Basic Security Theorem – Multics (Σ) is secure
  - If z_0 is a secure initial Multics state and
  - All Multics kernel primitives (rules for actions in set W)
    - Satisfy the simple security condition
    - Satisfy the *-property
    - Satisfy the discretionary security property

- ***Means all future Multics states are secure***

Multics as a State Machine

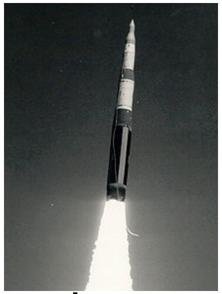


- Defined **all possible** transitions (Multics kernel calls), each transition satisfies the 3 properties

USC Viterbi School of Engineering

Other BLP Interpretation Experience

- Legacy of past security kernel deployment
  - Different deployments over periods of 30-40 years, no security patches
- Class A1 (formally verified) Security Kernel worked examples:
  1. SACDIN - Strategic Air Command Digital Network
     - Minuteman missile control (IBM)
     - LGM-30 Minuteman is a U.S. land-based intercontinental ballistic missile
  2. SCOMP for Multics comms (Honeywell)
     - Secure front-end communications processor for Multics
  3. Secure Ethernet LAN (Boeing)
  4. GTNP/GEMSOS (Gemini Computers, Inc.)
  5. Blacker "VPN" front-end (Unisys for NSA)
     - End-to-end encryption on the United States' Defense Data Network
  6. VAX secure VMM - not shipped (DEC)
     - Developed with a heavy emphasis on performance and on system management tools



Properties of FSPM of RM Abstraction

- Only known high assurance cyber security basis
  - Provides the basic theory of computer security
  - Essential for trustworthy response to subversion
- Also useful to assess low to medium assurance
  - When fails to meet all three RM principles
  - E.g., improving an OS
    - Improve” OS, not necessarily make it “secure”
  - E.g., RM functions in an application program
    - E.g., a DBMS
      - But note that it is still on low-assurance OS

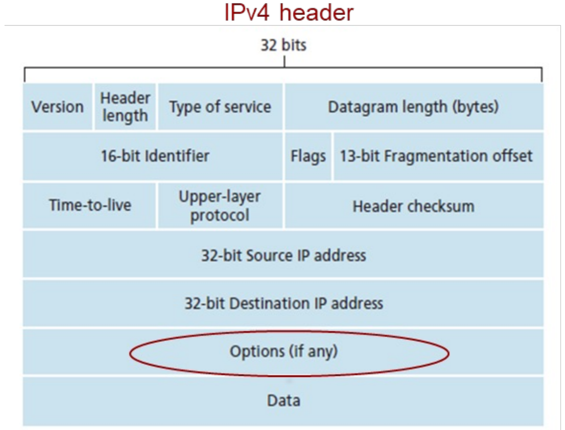
Case Study: Target Breach

- 2013 Target breach: 70M customers' personally identifiable information (PII) and 40M credit cards stolen (CC)
  1. Initial penetration through stolen vendor's credentials
  2. Gain access to a Target web services for vendors
  3. Deploy malware on many Target's POS machines which were used to steal credit card info
  4. Sent stolen credit cards periodically to a central repository within Target's network using standard Windows protocols
  5. Exfiltrated stolen data from the central repository to the attackers' controlled server via FTP

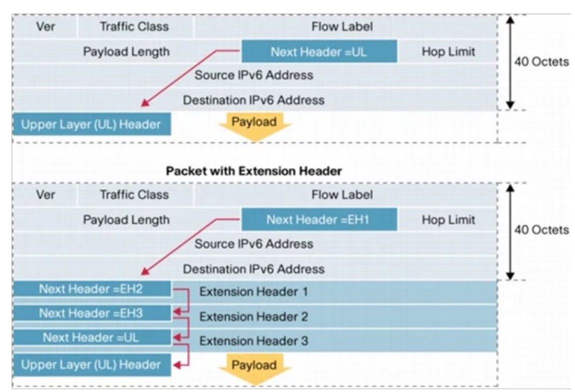
Proposed Solution for the Target Case

- Use BLP to enforce information flow
- Assign access classes to data:
  - PII data – (Secret, {Account})
  - CC data – (Secret, {Payment})
  - Vendor data – (Confidential, {HVAC})
  - Public data: Unclassified
- MLS DBs for PII and CC
- NTCB concept – partition network into TCB subsets (covered later in the course)

IPv4 header

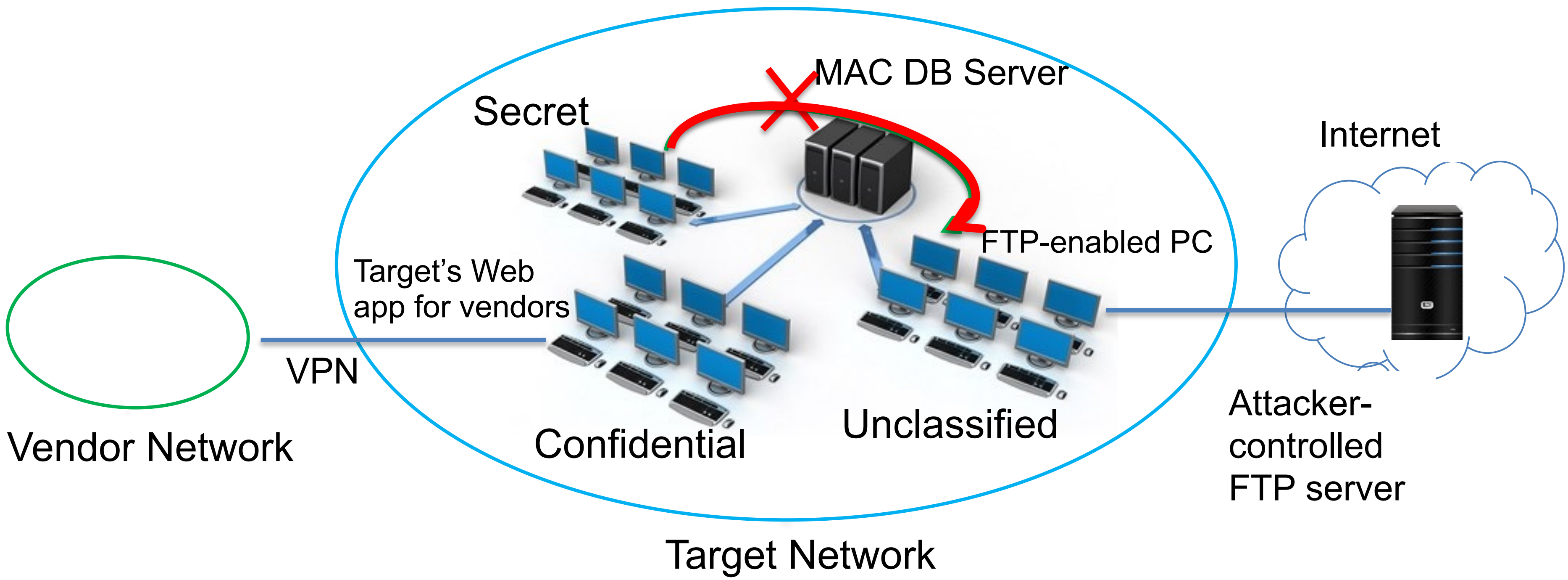


IPv6 header



USC Viterbi School of Engineering

MAC Solution for the Target Case



Can't exfiltrate sensitive data!

Outline

- Review
- Bell-LaPadula Multics Interpretation
- General undecidability of security
  - Preliminaries
    - Halting problem
    - Turing machines
    - Safety question
  - Harrison-Ruzzo-Ullman result

What are we trying to do?

- Can we determine if a computer system is "secure" (i.e., implements our policy with high assurance)?
- Our goal is to answer the key question: is this protection problem decidable?

Show that halting problem is undecidable → Reduce protection problem to halting problem → Arrive to the answer: undecidable in general case

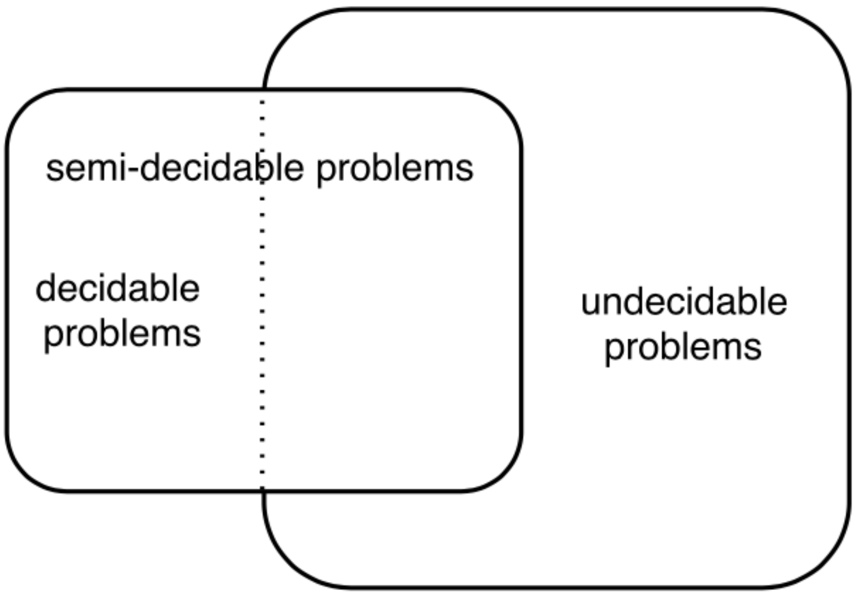
our focus now

General Undecidability of Security

- Is there a generic algorithm to determine if computer system is “secure”?
  - This is the core question for information assurance
- Foundational results from 1976 paper
  - Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman. "Protection in operating systems." Communications of the ACM 19.8 (1976): 461-471
  - Based on Turing machines and undecidability
  - Based on reduction to “halting problem”

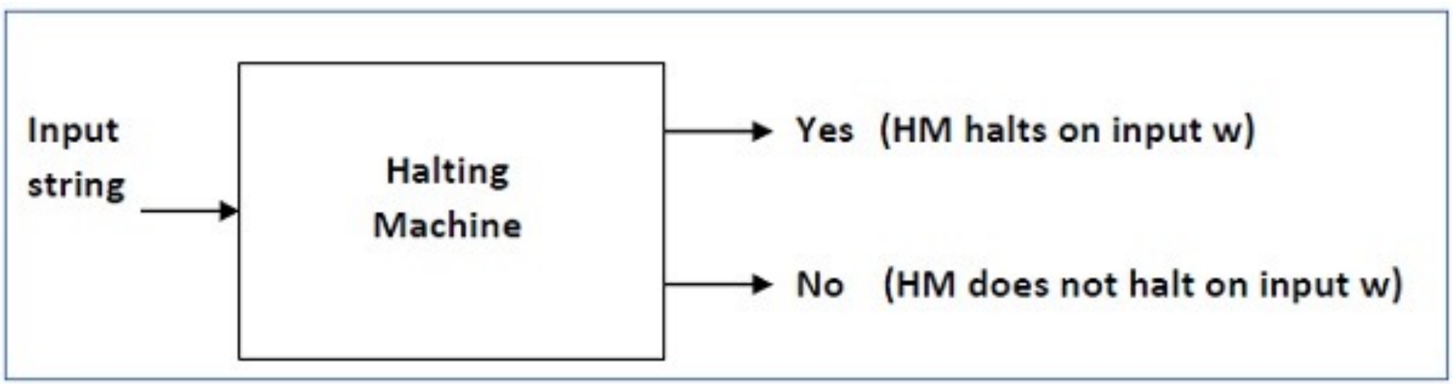
Decidability of Decision Problems

- Decidable problem
  - A decision problem can be solved by an algorithm that halts on **all** inputs in a **finite** number of steps
- Undecidable problem
  - A problem that cannot be solved for **all** cases by **any** algorithm
  - For a problem to be undecidable we just have to prove that there is **one** case it cannot produce an answer for



Halting Problem

- Is there a function (algorithm) that can look at **any** computer program and **any input** to that program and decide if the program will halt (not run infinitely)?
  - Question posed by Alan Turing in 1936 to prove that there are unsolvable problems



USC Viterbi School of Engineering

Proof by Contradiction

- Proof by contradiction: something that leads to a contradiction can not be true, and if so, the opposite must be true
  - To prove a statement by contradiction:
    - Start by assuming the **opposite** of what you want to prove
    - Then show that the consequences of this premise are impossible
      - This means that your original statement must be true
  - Example: prove that there is no largest number

Halting Problem: Proof

- Is there a function (algorithm) that can look at any computer program and any input to that program and decide if the program will halt?

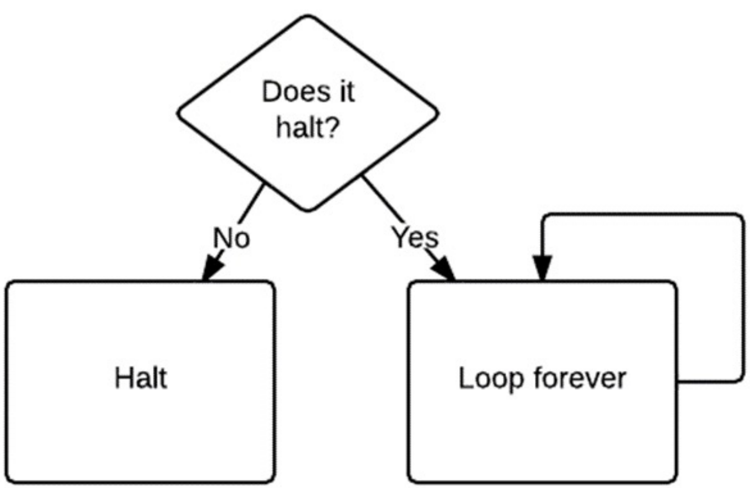
- Assume existence of such function:
  `HALT(program, input)` → `Boolean`, where
  - **`program`** is a code of a program we want to test
  - **`input`** is the input to the program we want to test
  - **`output`**: true if the program halts with this input, false otherwise

- Now define a new function:

```
SELF-HALT(program)
 {
     if (HALT(program, program))
          infinite loop
     else
          halt
  }
```

Halting Problem: Proof

- What if we use **`SELF-HALT`** to analyze itself?
  1. **`SELF-HALT(SELF-HALT)`** loops forever if **`HALT(SELF-HALT, SELF-HALT)`** is true
     - So if **`SELF-HALT`** halts on itself, it loops forever. A contradiction!
  2. **`SELF-HALT(SELF-HALT)`** halts if **`HALT(SELF-HALT, SELF-HALT)`** is false
     - So, if **`SELF-HALT`** does not halt on itself, it halts. A contradiction!
- This contradiction is unavoidable thus proving that the halting problem is undecidable



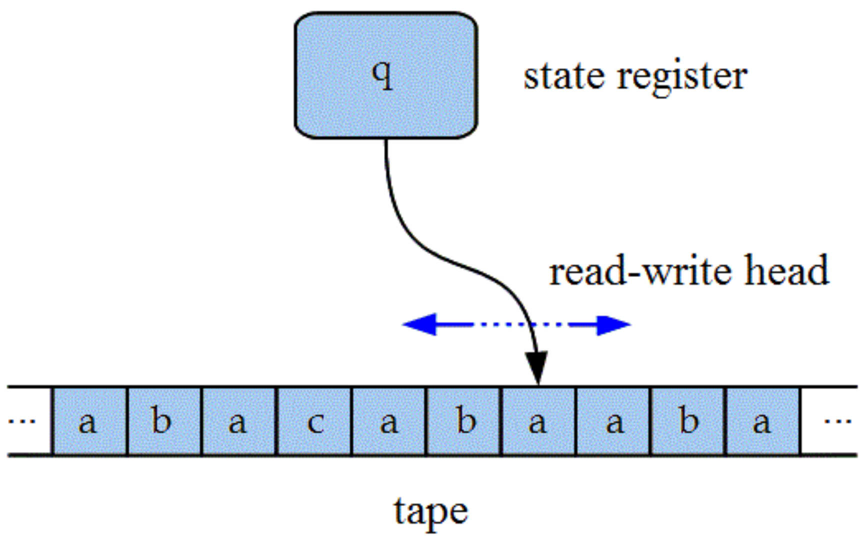

```
SELF-HALT(program)
    {
        if (HALT(program, program))
            infinite loop
        else
            halt
    }
```

Importance of The Halting Problem

- This is the first computer problem proven to be undecidable (unsolvable)
  - Impossible to construct a **single** algorithm that always leads to a correct yes/no answer to a decision problem on infinite set of inputs
- To show that a problem is undecidable, it is sufficient to show that it is **equivalent** to the halting problem
  - More precisely, it is sufficient to show that if a solution to the new problem existed, it could be used to solve the halting problem, which is known to be undecidable

Turing Machine (TM)

- A Turing machine is a **theoretical** computing machine invented by Alan Turing in 1936 to serve as an idealized model of computing
- TM is a finite state machine with an infinite tape from which it reads its input and on which it records its computations one cell at a time
- TM can perform a special action – it can stop (halt)
- Despite its simplicity, the machine can simulate ANY computer algorithm, no matter how complicated it is!



USC Viterbi School of Engineering

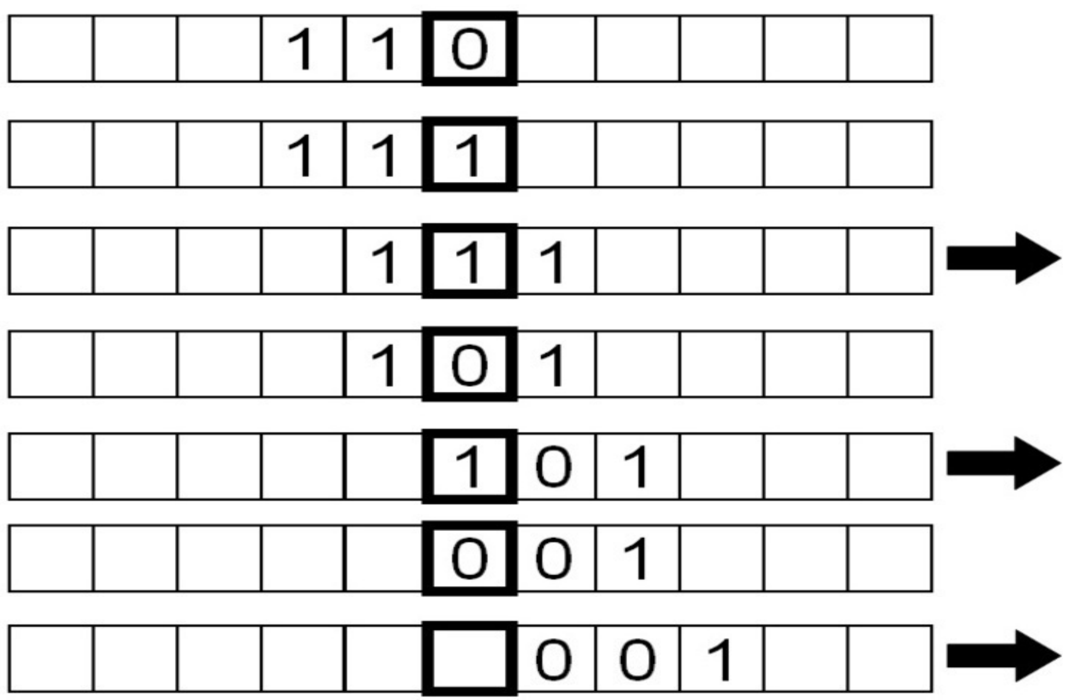
TM: Informally

- TM takes input symbol and according to it and the current state does the following:
  1. it reads the symbol on the tape and writes another (new or the same) symbol on the tape
  2. moves the tape right or left by one cell
  3. changes to a new state

TM: Simple Example

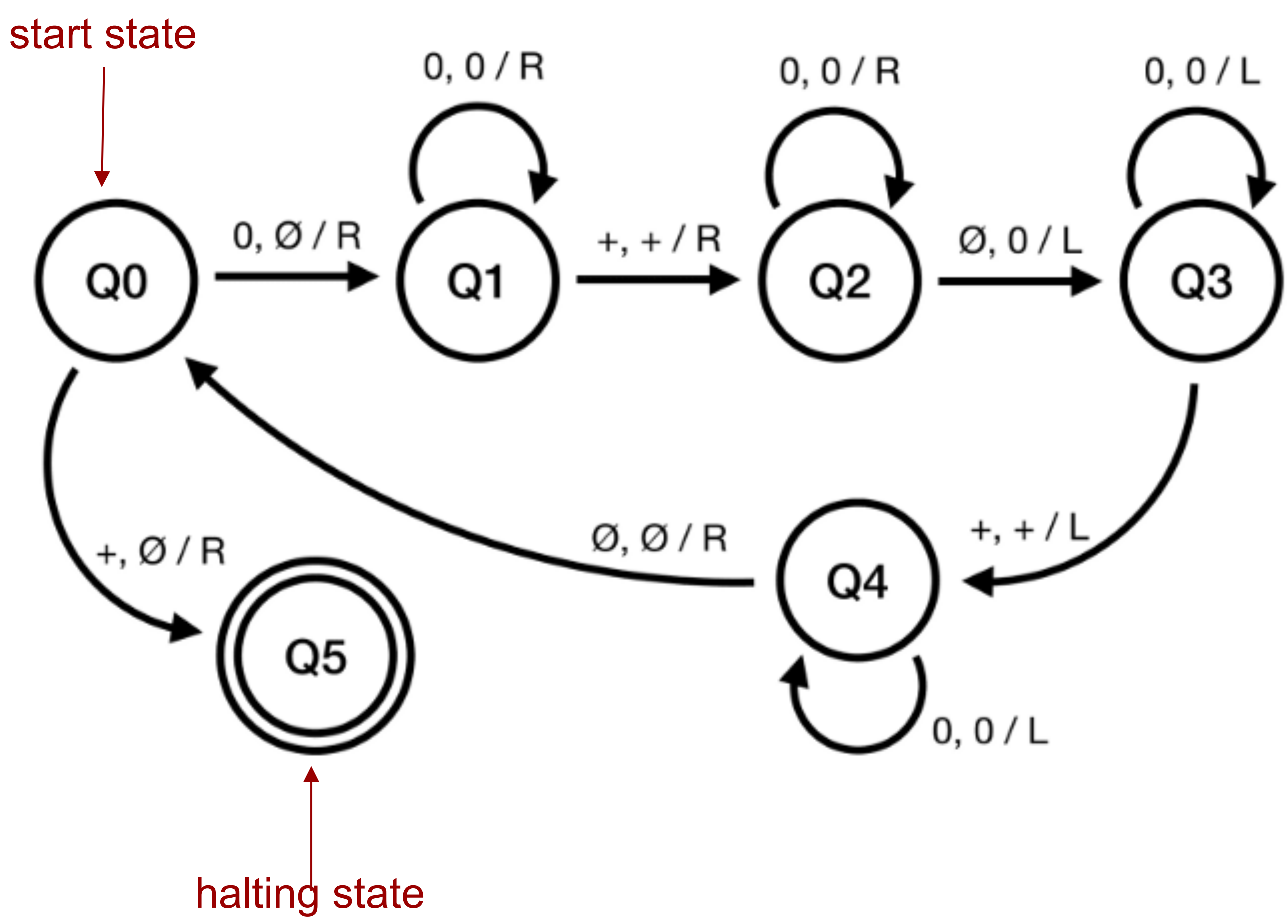
- Task: convert the 1s to 0s and vice versa

State	Symbol read	Write instruction	Move instruction	*Next state*
State 0	Blank	None	None	*Stop state*
	0	Write 1	Move the tape to the right	*State 0*
	1	Write 0	Move the tape to the right	*State 0*



TM: Another Example

- Task: perform addition, given input “00+000” (2 + 3) output “000000” (5)
- Use four symbols: “0”, “1”, “+” and “Ø”
  - “Ø” represents a blank space

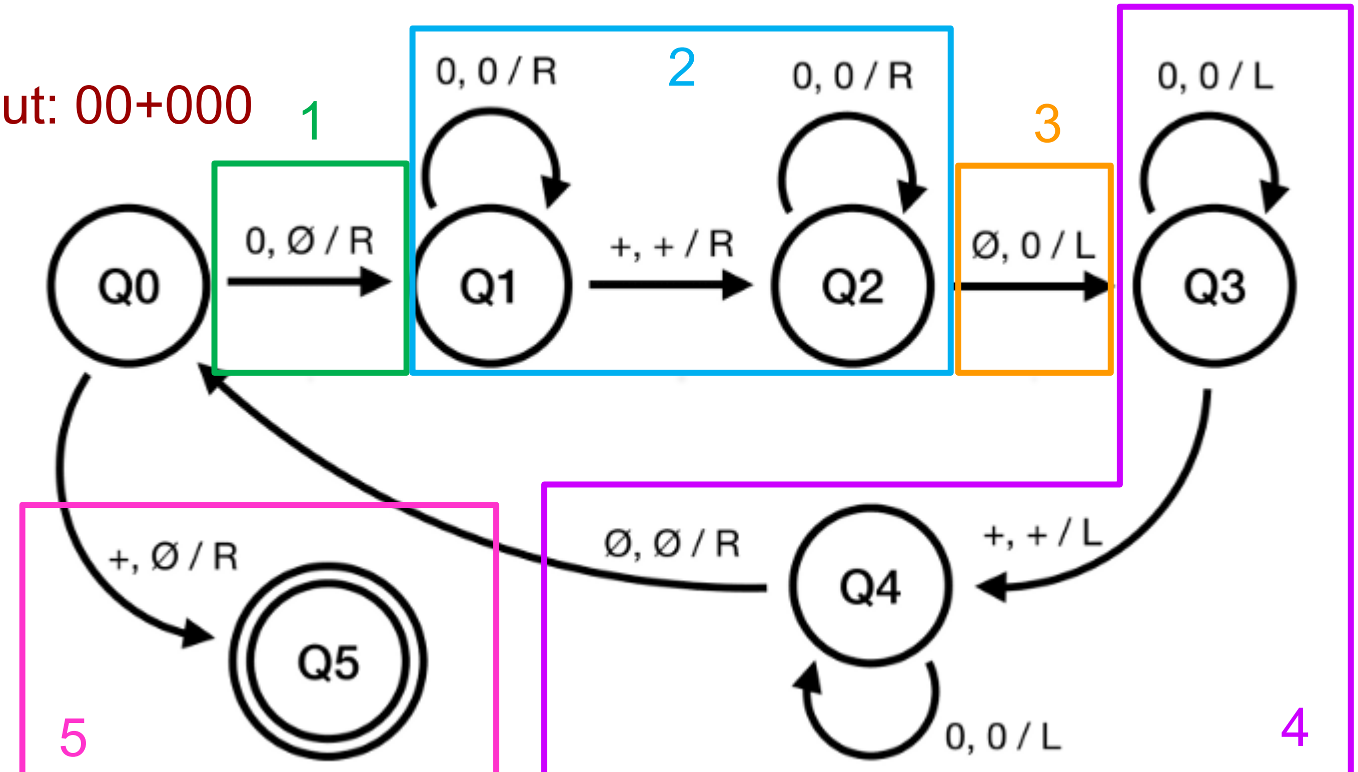


USC Viterbi School of Engineering

TM: Another Example

1. Replace the first 0 with a blank space
2. Move to the end of the string
3. Add a “0” to the end of the string
4. Go back to the start of the string and back to step 1
5. If the first symbol is a “+”, remove the “+” and complete

Input: 00+000



Loop 1: 00+000
Loop 2: 0+0000
Loop 3: +00000 ^Q1
Replace + with ø ^Q5

Output: 00000

TM: Formally

- A transition $\delta(q_i, x) = (q_j, y, d)$ depends on:
  1. the current state q_i, and
  2. the current symbol x under the tape head
- A transition consists of three actions:
  1. change state to q_j
  2. over-write tape symbol x by y, and
  3. move the tape in direction d = {L, R} (Left or Right)
  – If the tape head is on the leftmost symbol, moving left has no effect
- TM halts if it enters a state where there is **no move** – i.e., $\delta(q, x)$ is undefined

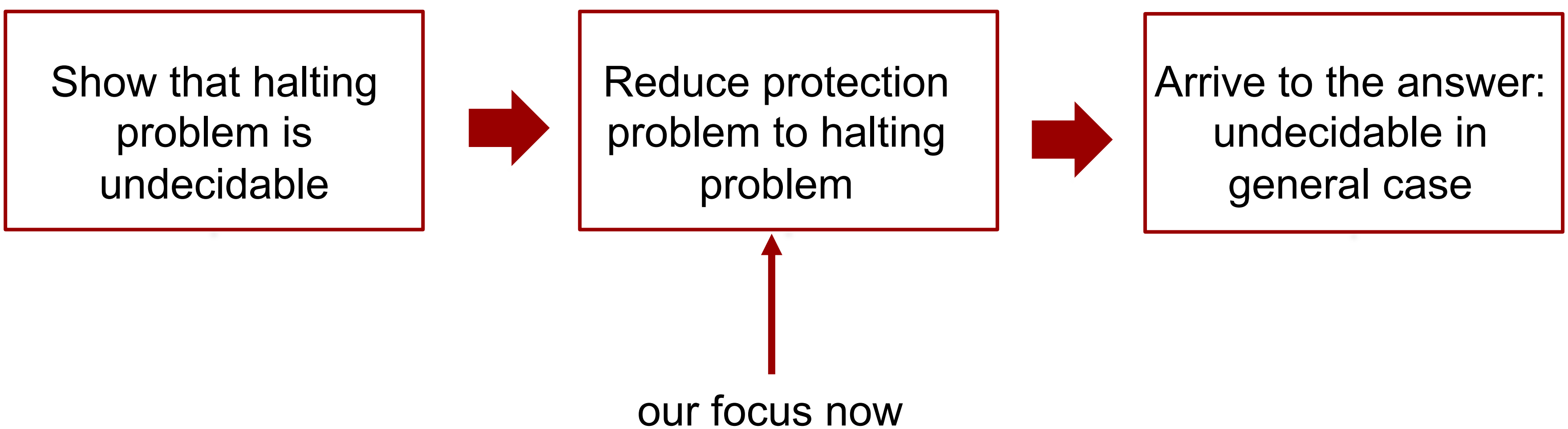
USC Viterbi School of Engineering

Outline

- Review
- Bell-LaPadula Multics interpretation
- General undecidability of security
  - Preliminaries
    - Halting problem
    - Turing machines
    - Safety question
  - Harrison-Ruzzo-Ullman result

What are we trying to do?

- Can we determine if a computer system is “secure” (i.e., implements our policy with high assurance)?
- Our goal is to answer the key question: is this protection problem decidable?



Objectives of the HRU Work

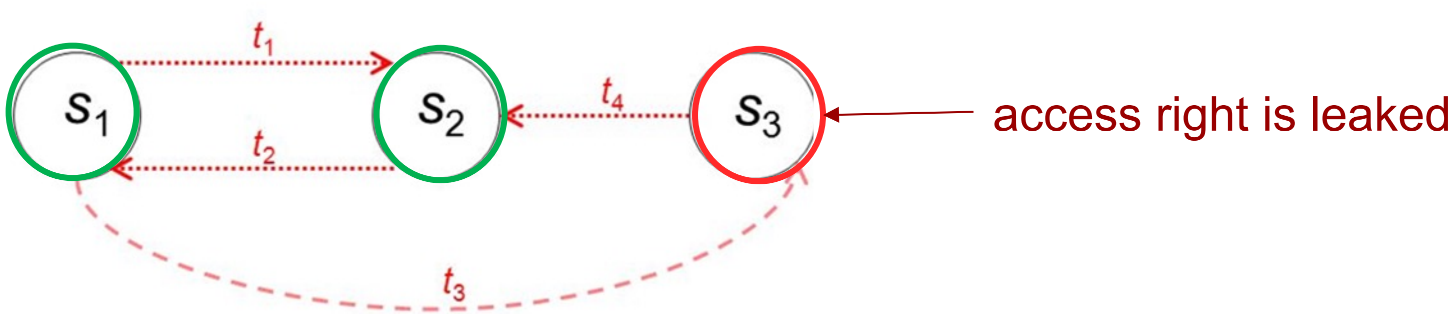
- Provide a model that is sufficiently powerful to encode several access control approaches, and precise enough so that security properties can be analyzed
- Introduce the "safety problem"
  - Accurately and concisely expresses the essence of the protection problem
- Show that the safety problem
  - is undecidable in general
  - is undecidable in monotonic case
    - A monotonic system is one in which rights cannot be deleted, and subjects and objects cannot be destroyed
  - is decidable in certain cases

Overview of the HRU Model

- What is “Secure”?
  - Adding a generic right r where there was not one is “leaking”
  - If a system S, beginning in initial state s_0, cannot leak right r, it is **safe** with respect to the right r
- Safety Question
  - Does there exist an algorithm for determining whether a protection system S with initial state s_0 is safe with respect to a generic right r?
  - Here, “safe” = “secure” for an abstract model
- Answer for a general case is “no”
  - Reduce halting problem to safety problem
  - Show that a Turing machine can be “modelled” by a protection system with the “states" of the machine mapped to the “rights" of the protection system

Notion of a "Leaked" Right

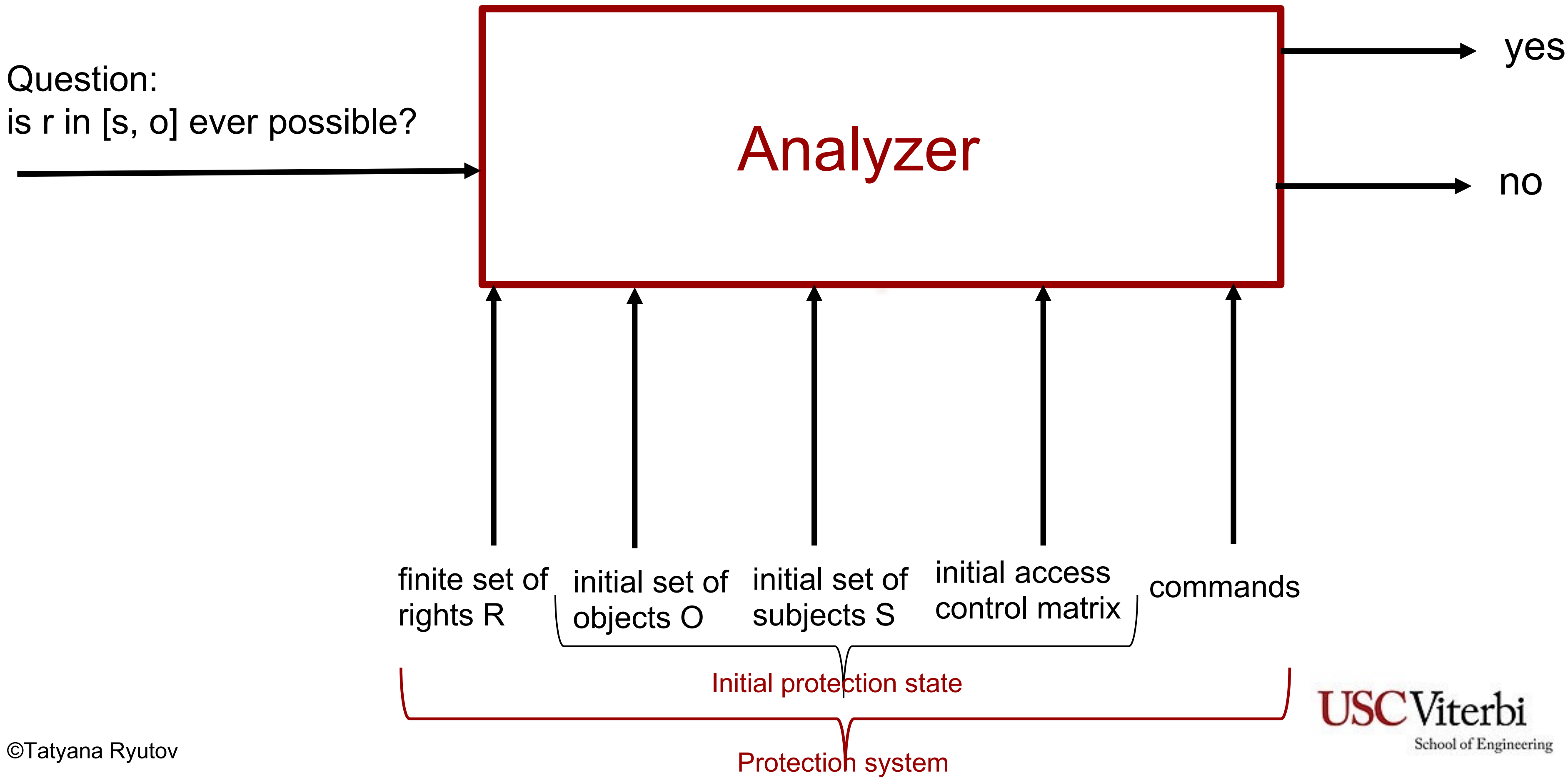
- Rights are the entries in access control matrix
  - Each subject has (or does not have) rights to an object
- Define **leaked**
  - Generic right added to element of access matrix when elements do not already contain the right
- Policy defines the authorized set of states
  - No command can leak a right *r*
- Define **safe** state with respect to a right *r*
  - System can never leak the right *r*
  - System is *unsafe* if it can enter unauthorized state



S_1 and S_2 - safe states, S_3 – unsafe state

HRU: General Idea

- Can we build an analyzer to answer the safety question?
  - Assume that the sole purpose of a system is to change access privileges to objects, ignoring other computations that might be occurring



HRU Protection System

- A protection system is a state-transition system
- A model for **protection** of computer system consists of:
  1. A finite set of generic rights R
  2. Initial protection state (initial set of objects O, initial set of subjects S, and initial access matrix P)
  3. A finite set of **commands** C of the form:

parameters

command $\boldsymbol{\alpha(X_1, X_2, \ldots, X_k)}$

if r_1 in (Xs_1, Xo_1) **and** r_2 in (Xs_2, Xo_2) **and** r_i in (Xs_i, Xo_i)

...

conditions (test presence of certain rights in certain positions in access matrix)

then

op_1; op_2; … op_n

end

command body is straight line code, no conditionals, no function invocation

- These commands are interpreted as a sequence of primitive **operations**

USC Viterbi School of Engineering

HRU Primitive Operations

- Primitive operations affect the state of access matrix:
  1. enter r into (Xs, Xo)
     - Condition: Xs $\in$ S and Xo $\in$ O
     - r may already exist in (Xs, Xo)
  2. delete r from (Xs, Xo)
     - Condition: Xs $\in$ S and Xo $\in$ O
     - r does not need to exist in (Xs, Xo)
  3. create subject Xs
     - Condition: Xs $\notin$ O
  4. create object Xo
     - Condition: Xo $\notin$ O
  5. destroy subject Xs
     - Condition: Xs $\in$ S
  6. destroy object Xo
     - Condition: Xo $\in$ O and Xo $\notin$ S

The State of A Protection System

- HRU define the "configuration" of a system
- Instantaneous description of protection system is a triple (S, O, P):
  - S is the set of current subjects,
  - O is the set of current objects, $S \subseteq O$ (subjects can be objects)
  - P is an access control matrix
    - one row for each subject
    - one column for each object
    - each cell contains a set of rights from R

USC Viterbi School of Engineering

How does state transition work?

- Given a protection system (R, C), state q_1 can reach state q_2 IFF there is an instance of a command in C so that all conditions are true at state q_1 and executing the primitive operations one by one results in state q_2
  - R is a finite set of generic rights
  - C is a finite set of commands

these sets do not change

- A command is executed as a whole (similar to a transaction), if one step fails, then nothing changes

Example 1

[Unix] process p creates file f with owner *read* and *write* (r, w) will be represented by the following:

Command $create\_file(p, f)$

Create object f ← create a column in the access matrix

Enter own into $a[p,f]$

Enter r into $a[p,f]$

Enter w into $a[p,f]$

End

Example 2

- Process p creates a new process q

  Command *spawn_process*(p, q)

    Create subject q; ← create a row in the access matrix

    Enter *own* into $a[p,q]$

    Enter r into $a[p,q]$

    Enter w into $a[p,q]$

    Enter r into $a[q,p]$ ← parent and child can signal each other

    Enter w into $a[q,p]$

  End

USC Viterbi School of Engineering

HRU: The Safety Problem

- Given a protection system and generic right r, we say that the initial configuration Q_0 is unsafe for r (or leaks r) if there is a configuration Q and a command α such that:
  - Q is reachable from Q_0
  - α leaks r from Q
- We say that a command $\alpha(x_1,\ldots,x_k)$ leaks generic right r from Q if α, when run on Q, can execute a primitive operation which enters r into a cell of the access matrix which did not previously contain r

USC Viterbi School of Engineering

Relationship between TM and HRU

- It is undecidable (no generic algorithm) to determine whether an arbitrary TM halts or not
  - Or enters any arbitrary state q_f
- Idea: reduce protection problem (HRU) to TM
  - If TM enters state q_f, then the protection system can leak generic right r, otherwise, it is safe for r
  - Generic right *r* is arbitrary and hence yielding state q_f is also arbitrary
  - Since it is undecidable whether the TM enters arbitrary state q_f, it must be undecidable whether the protection system is safe for r
- Next question: how to map HRU to TM?

Mapping a Tape to an Access Matrix

- Create the protection matrix from the TM's tape
  - Encode the contents of the TM's tape on the diagonal of the protection matrix: element [s, s] will contain the s^{th} tape square
  - How can we represent **sequential** tape?
    - Subject s_i represents cell i^{th} cell on the tape
    - Each subject s_i “owns” subject s_{i+1}
    - Sequential ownership relation represents sequential tape
  - Any cell (in TM) holding a symbol indicates subject s_i gave that right to itself
  - Last subject s_k has right *end*, indicating that subject s_{k+1} (which s_k owns), has not yet been created
  - The head is at the i^{th} cell and the current state is $q \Rightarrow q \in (s_i, s_i)$

	s1	s2	s3	. . .				
s1	A	own						
s2		B	own					
s3			C	own				
.				D	own			
.					E	own		
.						F	own	
.								

A	B	C	D	E	F	G	H	

USC Viterbi School of Engineering

Mapping a Tape to an Access Matrix

- We also need to encode the state of the TM
  - The set of generic rights represent states and tape symbols
  - Two special rights: *own* and *end*
    - *end* is the last cell before blanks
  - For example, rights g and c are elements of $[s_3, s_3]$ when the TM is in state q, the read/write head is on square 3, and symbol c is in $[s_3, s_3]$
- Turing Machine instructions are mapped to commands of the HRU protection system

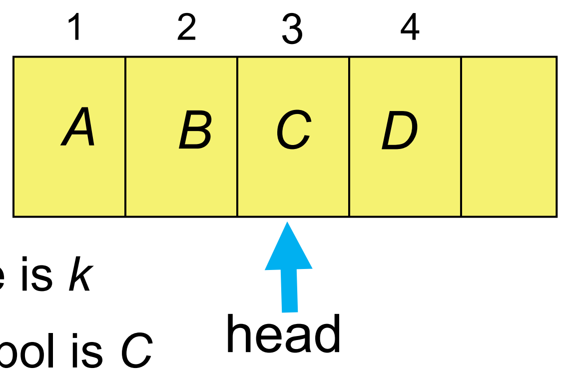
Representing Head Moves

- The moves of TM are represented as HRU commands
  - Changing the state of the TM is equivalent to commands that delete and add rights, objects, and subjects
- Choose HRU commands to represent TM moves
  - Right: $\delta(k, \mathrm{C}) = (k_1, \mathrm{X}, \mathrm{R})$
    - E.g., if cell k is the current position, command (k_1,C/X,R) substitutes access right C for access right X in the cell, and k_1 is the cell to the immediate right of k
    - When move right till end (blanks), need to create a new subject
  - Left: $\delta(k, \mathrm{C}) = (k_1, \mathrm{X}, \mathrm{L})$
    - E.g., if cell k is the current position, command (k_1,C/X,L) substitutes access right C for access right X in the cell, and k_1 is the cell to the immediate left of k
- For any possible TM transition, can have corresponding HRU command

USC Viterbi School of Engineering

Mapping Depiction

TM



Current state is *k*

Current symbol is *C*

HRU Matrix

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

Symbols, States $\Rightarrow$ rights

Tape cell $\Rightarrow$ subject

Cell s_i has *A* $\Rightarrow$ s_i has *A* right on itself

Cell s_k $\Rightarrow$ s_k has *end* right on itself

State *p*, head at s_i $\Rightarrow$ s_i has *p* right on itself

Distinguished right *own*: s_i *owns* s_i*+1* for $1 \leq i < k$

USC Viterbi School of Engineering

Right Move (Left is Symmetrical)

command $c_{k,C}(s_3, s_4)$
if *own* **in** $A[s_3, s_4]$ **and** k **in** $A[s_3, s_3]$
  and C **in** $A[s_3, s_3]$
then
  delete k **from** $A[s_3, s_3]$;
  delete C **from** $A[s_3, s_3]$;
  enter X **into** $A[s_3, s_3]$;
  enter k_1 **into** $A[s_4, s_4]$;
end

1	2	3	4	
A	*B*	*C*	*D*	

head

Current state is *k*

Current symbol is *C*

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

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

USC Viterbi School of Engineering

After One Right Move

1	2	3	4	
A	*B*	*C*	*D*	...

Current state is k_1

Current symbol is *C*

head

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

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

Right Move at End

```
command crightmost_{k,C}(s4,s5)
if end in A[s4,s4] and k1 in A[s4,s4]
        and D in A[s4,s4]
then
   delete end from A[s4,s4];
   create subject s5;
   enter own into A[s4,s5];
   enter end into A[s5,s5];
   delete k1 from A[s4,s4];
   delete D from A[s4,s4];
   enter Y into A[s4,s4];
   enter k2 into A[s5,s5];
end
```

1	2	3	4	
A	*B*	*X*	*D*	…

head

Current state is k_1

Current symbol is *D*

$$\delta(k_1, \mathrm{D}) = (k_2, \mathrm{Y}, \mathrm{R})$$

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

After Right Move at End

1	2	3	4	
A	*B*	*X*	*D*	…

Current state is k_1

Current symbol is *D*

head

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

$$\delta(k_1, \mathrm{D}) = (k_2, \mathrm{Y}, \mathrm{R})$$

USC Viterbi School of Engineering

Rest of Proof

- Protection system exactly simulates actions of TM
  - Exactly one *end* right – blanks after that
  - Exactly one access right corresponds to a TM state
    - Thus, at most one applicable command
- If TM enters state q_f, then right is leaked
- Generic right *r* is **arbitrary** and hence yielding state q_f is also arbitrary
- If safety question is decidable, then we can represent TM as discussed and determine if q_f leaks
  - Implies halting problem is decidable. A contradiction!
- Conclusion: safety problem undecidable

USC Viterbi School of Engineering

What are we trying to do?

- Can we determine if a computer system is “secure” (i.e., implements our policy with high assurance)?
- Our goal is to answer the key question: is this protection problem decidable?

Show that halting problem is undecidable → Reduce protection problem to halting problem → Arrive to the answer: undecidable in general case

our focus now

Theoretical Limits on System Security Summary

- Harrison, Ruzzo and Ullman (HRU) defined "safety" problem for protection systems
  - Safety refers to some **abstract** model
  - Security refers to **actual** implementation
- System can only be secure if it implements a policy based on a safe model
  - But a safe model does NOT ensure a secure system
- DAC has fundamental flow control limitation
  - It is generally unsafe model
  - Consistent with analysis of Trojan horse threat
- MAC can be safe
  - Imposes sufficient restrictions on right propagation
- How can we use this in practice?
  - Balanced assurance