Chapter 16: Confinement Problem

- What is the problem?
- Isolation: virtual machines, sandboxes
- Detecting covert channels
- Analyzing covert channels
- Mitigating covert channels

Overview

- The confinement problem
- Isolating entities
  - Virtual machines
  - Sandboxes
- Covert channels
  - Detecting them
  - Analyzing them
  - Mitigating them

Example Problem

- Server balances bank accounts for clients
- Server security issues:
  - Record correctly who used it
  - Send *only* balancing info to client
- Client security issues:
  - Log use correctly
  - Do not save or retransmit data client sends

Generalization

- Client sends request, data to server
- Server performs some function on data
- Server returns result to client
- Access controls:
  - Server must ensure the resources it accesses on behalf of client include *only* resources client is authorized to access
  - Server must ensure it does not reveal client's data to any entity not authorized to see the client's data

Confinement Problem

- Problem of preventing a server from leaking information that the user of the service considers confidential

Total Isolation

- Process cannot communicate with any other process
- Process cannot be observed

Impossible for this process to leak information

  – Not practical as process uses observable resources such as CPU, secondary storage, networks, etc.

Example

- Processes *p*, *q* not allowed to communicate
  - But they share a file system!
- Communications protocol:
  - *p* sends a bit by creating a file called *0* or *1*, then a second file called *send*
    - *p* waits until *send* is deleted before repeating to send another bit
  - *q* waits until file *send* exists, then looks for file *0* or *1*; whichever exists is the bit
    - *q* then deletes *0*, *1*, and *send* and waits until *send* is recreated before repeating to read another bit

Covert Channel

- A path of communication not designed to be used for communication
- In example, file system is a (storage) covert channel

Rule of Transitive Confinement

- If *p* is confined to prevent leaking, and it invokes *q*, then *q* must be similarly confined to prevent leaking
- Rule: if a confined process invokes a second process, the second process must be as confined as the first

Lipner's Notes

- All processes can obtain rough idea of time
  - Read system clock or wall clock time
  - Determine number of instructions executed
- All processes can manipulate time
  - Wait some interval of wall clock time
  - Execute a set number of instructions, then block

Kocher's Attack

- This computes $x = a^z \bmod n$, where $z = z_0 \ldots z_{k-1}$

```
x := 1; atmp := a;
for i := 0 to k–1 do begin
  if z_i = 1 then
     x := (x * atmp) mod n;
  atmp := (atmp * atmp) mod n;
end
result := x;
```

- Length of run time related to number of 1 bits in z

Isolation

- Virtual machines
  - Emulate computer
  - Process cannot access underlying computer system, anything not part of that computer system
- Sandboxing
  - Does not emulate computer
  - Alters interface between computer, process

Virtual Machine (VM)

- A program that simulates hardware of computer system
- *Virtual machine monitor* (VMM) provides VM on which conventional OS can run
  - Each VM is one subject; VMM knows nothing about processes running on each VM
  - VMM mediates all interactions of VM with resources, other VMS
  - Satisfies rule of transitive closure

Example: KVM/370

- Security-enhanced version of IBM VM/370 VMM
- Goals
  - Provide virtual machines for users
  - Prevent VMs of different security classes from communicating
- Provides minidisks; some VMs could share some areas of disk
  - Security policy controlled access to shared areas to limit communications to those allowed by policy

DEC VAX VMM

- VMM is security kernel
  - Can run Ultrix OS or VMS OS
- Invoked on trap to execute privileged instruction
  - Only VMM can access hardware directly
  - VM kernel, executive levels both mapped into physical executive level
- VMM subjects: users, VMs
  - Each VM has own disk areas, file systems
  - Each subject, object has multilevel security, integrity labels

Sandbox

- Environment in which actions of process are restricted according to security policy
  - Can add extra security-checking mechanisms to libraries, kernel
    - Program to be executed is not altered
  - Can modify program or process to be executed
    - Similar to debuggers, profilers that add breakpoints
    - Add code to do extra checks (memory access, etc.) as program runs (*software fault isolation*)

Example: Limiting Execution

- Sidewinder
  - Uses type enforcement to confine processes
  - Sandbox built into kernel; site cannot alter it
- Java VM
  - Restricts set of files that applet can access and hosts to which applet can connect
- DTE, type enforcement mechanism for DTEL
  - Kernel modifications enable system administrators to configure sandboxes

Example: Trapping System Calls

- Janus: execution environment
  - Users restrict objects, modes of access
- Two components
  - *Framework* does run-time checking
  - *Modules* determine which accesses allowed
- Configuration file controls modules loaded, constraints to be enforced

Janus Configuration File

```
# basic module
basic
```
— *Load basic module*
```
# define subprocess environment variables
putenv IFS="\t\n " PATH=/sbin:/bin:/usr/bin TZ=PST8PDT
```
— *Define environmental variables for process*
```
# deny access to everything except files under /usr
path deny read,write *
path allow read,write /usr/*
```
— *Deny all file accesses except to those under /usr*
```
# allow subprocess to read files in library directories
# needed for dynamic loading
path allow read /lib/* /usr/lib/* /usr/local/lib/*
```
— *Allow reading of files in these directories (all dynamic load libraries are here)*
```
# needed so child can execute programs
path allow read,exec /sbin/* /bin/* /usr/bin/*
```
— *Allow reading, execution of subprograms in these directories*

Janus Implementation

- System calls to be monitored defined in modules
- On system call, Janus framework invoked
  - Validates system call *with those specific parameters* are allowed
  - If not, sets process environment to indicate call failed
  - If okay, framework gives control back to process; on return, framework invoked to update state
- Example: reading MIME mail
  - Embed "delete file" in Postscript attachment
  - Set Janus to disallow Postscript engine access to files

Covert Channels

- Channel using shared resources as a communication path
- *Covert storage channel* uses attribute of shared resource
- *Covert timing channel* uses temporal or ordering relationship among accesses to shared resource

Example: File Manipulation

- Communications protocol:
  - *p* sends a bit by creating a file called *0* or *1*, then a second file called *send*
    - *p* waits until *send* is deleted before repeating to send another bit
  - *q* waits until file *send* exists, then looks for file *0* or *1*; whichever exists is the bit
    - *q* then deletes *0*, *1*, and *send* and waits until *send* is recreated before repeating to read another bit
- Covert storage channel: resource is directory, names of files in directory

Example: Real-Time Clock

- KVM/370 had covert timing channel
  - VM1 wants to send 1 bit to VM2
  - To send 0 bit: VM1 relinquishes CPU as soon as it gets CPU
  - To send 1 bit: VM1 uses CPU for full quantum
  - VM2 determines which bit is sent by seeing how quickly it gets CPU
  - Shared resource is CPU, timing because real-time clock used to measure intervaps between accesses

Example: Ordering of Events

- Two VMs
  - Share cylinders 100–200 on a disk
  - One is *High*, one is *Low*; process on *High* VM wants to send to process on *Low* VM
- Disk scheduler uses SCAN algorithm
- *Low* process seeks to cylinder 150 and relinquishes CPU
  - Now we know where the disk head is

Example (*con't*)

- *High* wants to send a bit
  - To send 1 bit, *High* seeks to cylinder 140 and relinquish CPU
  - To send 0 bit, *High* seeks to cylinder 160 and relinquish CPU
- *Low* issues requests for tracks 139 and 161
  - Seek to 139 first indicates a 1 bit
  - Seek to 161 first indicates a 0 bit
- Covert timing channel: uses ordering relationship among accesses to transmit information

Noise

- *Noiseless covert channel* uses shared resource available to sender, receiver only
- *Noisy covert channel* uses shared resource available to sender, receive, and others
  - Need to minimize interference enough so that message can be read in spite of others' use of channel

Key Properties

- Existence
  - Determining whether the covert channel exists
- Bandwidth
  - Determining how much information can be sent over the channel

Detection

- Covert channels require sharing
- Manner of sharing controls which subjects can send, which subjects can receive information using that shared resource
- Porras, Kemmerer: model flow of information through shared resources with a tree
  - Called *covert flow trees*

Goal Symbol Tree Nodes

- Modification: attribute modified
- Recognition: attribute modification detected
- Direct recognition: subject can detect attribute modification by referencing attribute directly or calling function that returns it
- Inferred recognition: subject can detect attribute modification without direct reference
- Inferred-via: info passed from one attribute to another via specified primitive (e.g. syscall)
- Recognized-new-state: modified attribute specified by inferred-via goal

Other Tree Nodes

- Operation symbol represents primitive operation
- Failure symbol indicates information cannot be sent along path
- And symbol reached when for all children
  - Child is operation; and
  - If child goal, then goal is reached
- Or symbol reached when for any child:
  - Child is operation; or
  - If child goal, then goal is reached

Constructing Tree

- Example: files in file system have 3 attributes
  - *locked*: true when file locked
  - *isopen*: true when file opened
  - *inuse*: set containing PID of processes having file open
- Functions:
  - *read_access*(*p*, *f*): true if *p* has read rights over file *f*
  - *empty*(*s*): true if set *s* is empty
  - *random*: returns one of its arguments chosen at random

Locking and Opening Routines

```
(* lock the file if it is not locked and
not opened; otherwise indicate it is
locked by returning false *)
procedure Lockfile(f: file): boolean;
begin
   if not f.locked and empty(f.inuse)
then
      f.locked := true;
end;
(* unlock the file *)
procedure Unlockfile(f: file);
begin
   if f.locked then
      f.locked := false;
end;
(* say whether the file is locked *)
function Filelocked(f: file): boolean;
begin
   Filelocked := f.locked;
end;
```

```
(* open the file if it isn't locked and
the process has the right to read the
file *)
procedure Openfile(f: file);
begin
   if not f.locked and
        read_access(process_id, f) then
      (* add process ID to inuse set *)
      f.inuse = f.inuse + process_id;
end;
(* if the process can read the file, say
if the file is open, otherwise return a
value at random *)
function Fileopened(f: file): boolean;
begin
   if not read_access(process_id, f) then
      Fileopened := random(true, false);
   else
      Fileopened := not isempty(f.inuse);
end
```

Attributes and Operations

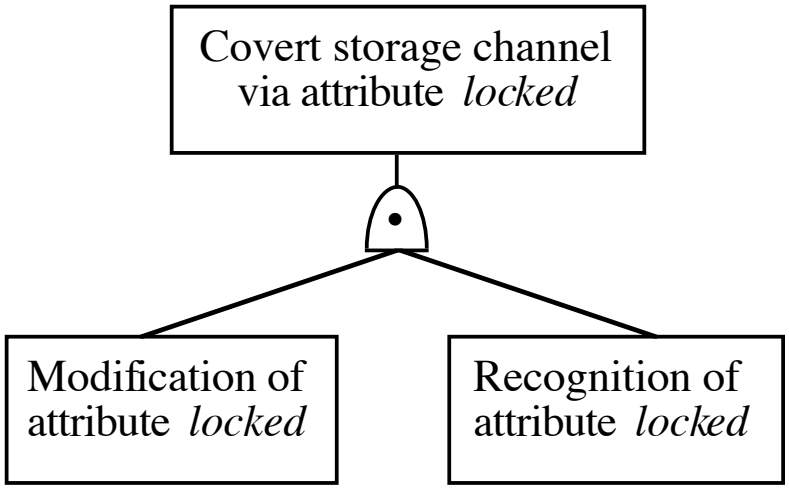
	Lockfile	Unlockfile	Filelocked	Openfile	Fileopened
reference	*locked, inuse*	*locked*	*locked*	*locked, inuse*	*inuse*
modify	*locked*	∅	∅	*inuse*	∅
return	∅	∅	*locked*	∅	*inuse*

∅ means no attribute affected in specified manner

Tree Construction

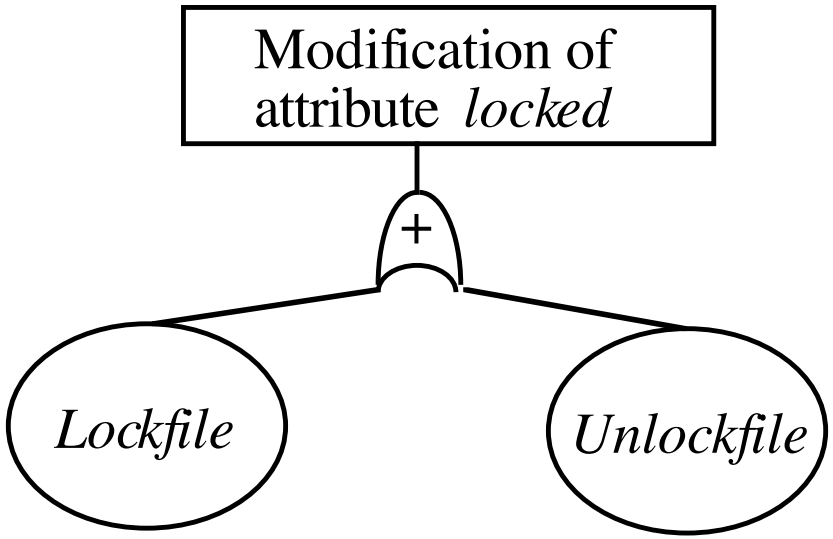
- This is for attribute *locked*
  - Goal state: "covert storage channel via attribute *locked*"
  - Type of goal controls construction
- "And" node has 2 children, a "modification" and a "recognition"
  - Here, both "of attribute *locked*"

First Step



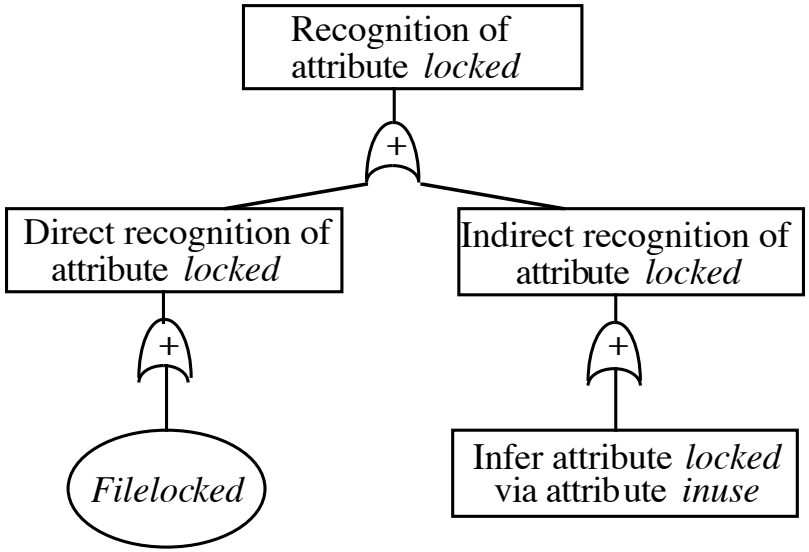
- Put "and" node under goal
- Put children under "and" node

Second Step



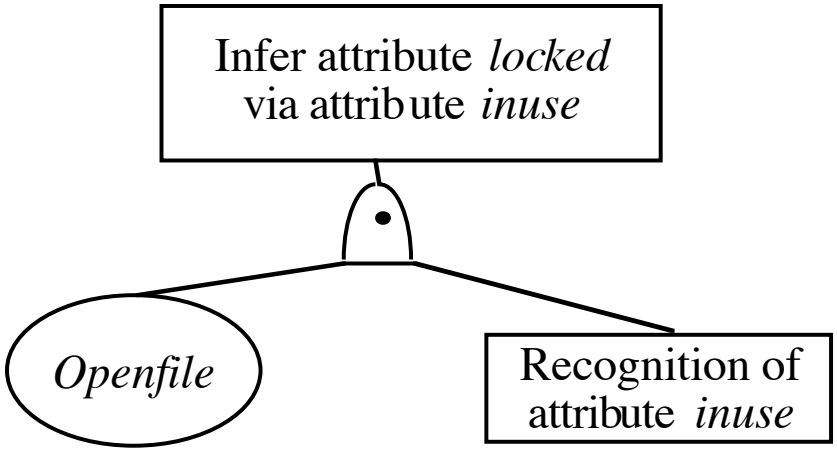
- Operations *Lockfile* and *Unlockfile* modify *locked*
  - See attribute and operations table

Third Step



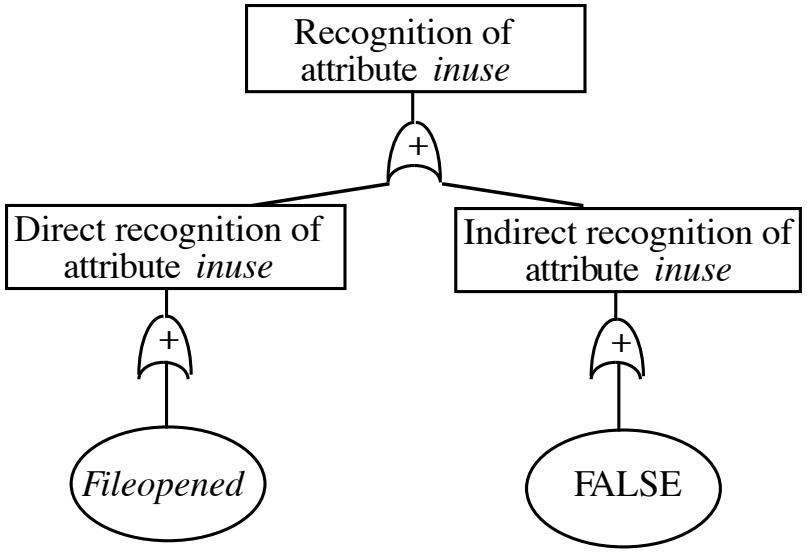
- "Recognition" had direct, inferred recognition children
- Direct recognition child: "and" node with *Filelocked* child
  - *Filelocked* returns value of *locked*
- Inferred recognition child: "or" node with "inferred-via" node
  - Infers *locked* from *inuse*

Fourth Step



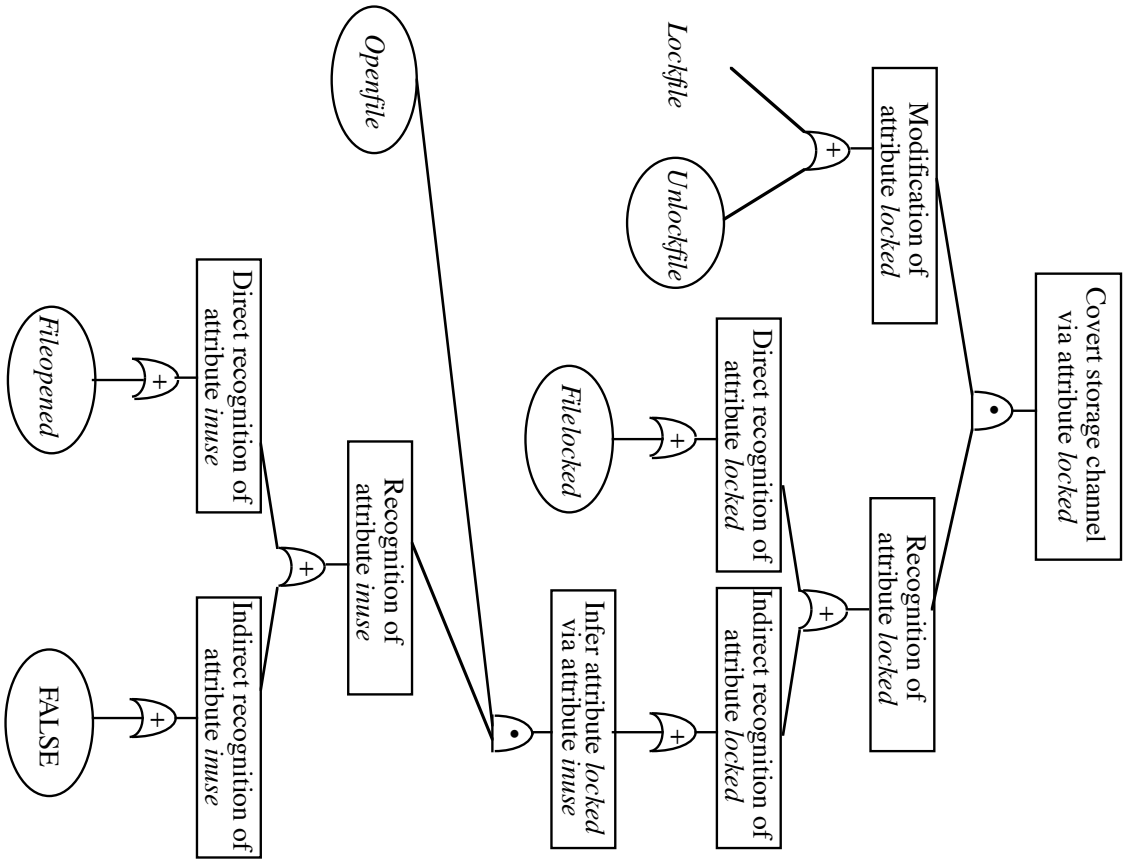
- "Inferred-via" node requires *Openfile*
  - Change in attribute *inuse* represented by recognize-new-state goal

Fifth Step



- "Recognize-new-state" node
  - Direct recognition node: "or" child, *Fileopened* node beneath (recognizes change in *inuse* directly)
  - Inferred recognition node: "or" child, FALSE node beneath (nothing recognizes change in inuse indirectly)

Final Tree



Finding Covert Channels

- Find sequences of operations that modify attribute
  - (*Lockfile*), (*Unlockfile*)
- Find sequences of operations that recognize modifications to attribute
  - (*Filelocked*), (*Openfile*, *Fileopened*))

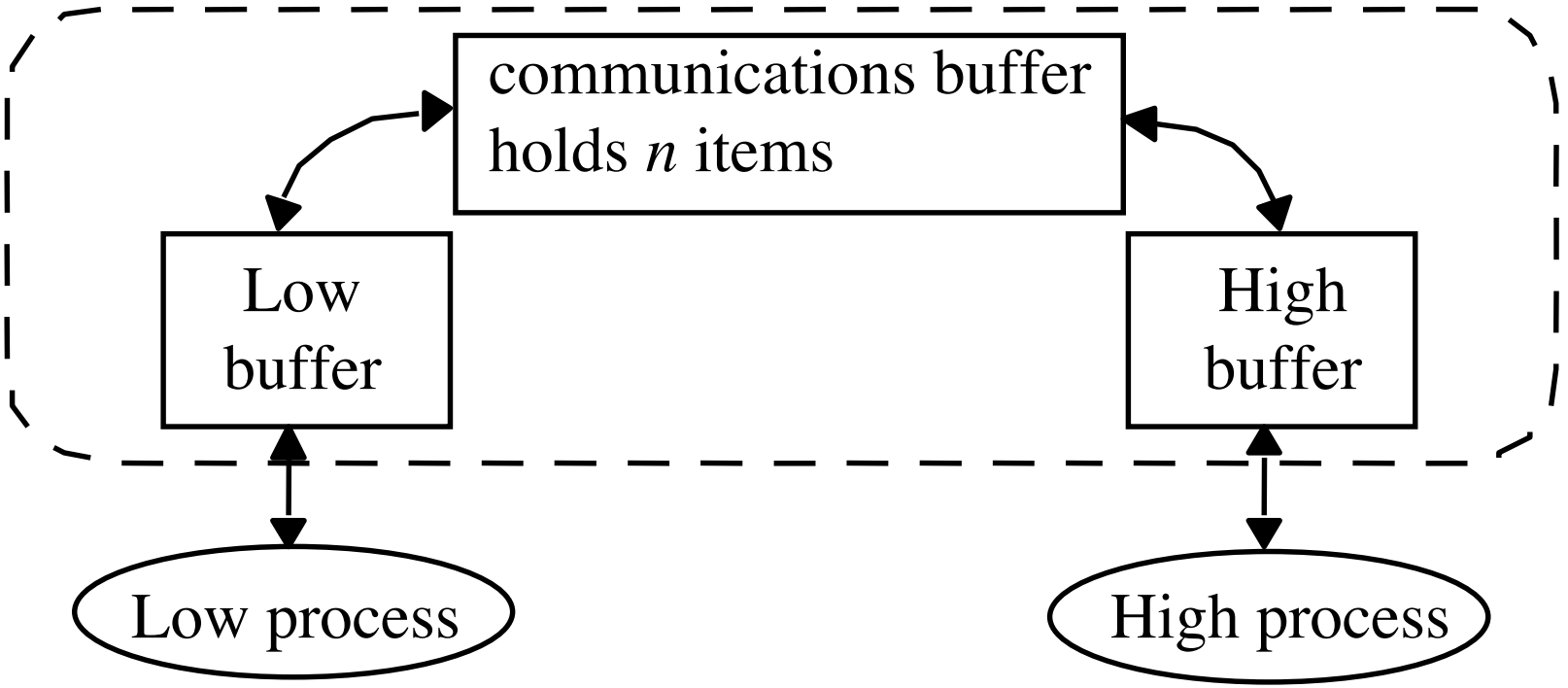
Covert Channel Commands

- Sequences with first element from first list, second element from second list
  - *Lockfile*, then *Filelocked*
  - *Unlockfile*, then *Filelocked*
  - *Lockfile*, then *Openfile*, then *Fileopened*
  - *Unlockfile*, then *Openfile*, then *Fileopened*

Mitigation

- Goal: obscure amount of resources a process uses
  - Receiver cannot determine what part sender is using and what part is obfuscated
- How to do this?
  - Devote uniform, fixed amount of resources to each process
  - Inject randomness into allocation, use of resources

Example: Pump



Covert Timing Channel

- High process can control rate at which pump sends it messages
- Initialization: Low sends messages to pump until communications buffer full
  - Low gets ACK for each message put into the buffer; no ACK for messages when communications buffer full
- Protocol: sequence of trials; for each trial
  - High sends a 1 by reading a message
    - Then Low gets ACK when it sends another message
  - High sends a 0 by not reading a message
    - Then Low doesn't gets ACK when it sends another message

How to Fix

- Assume: Low process, pump can process messages faster than High process
- Case 1: High process handles messages more quickly than Low process gets acknowledgements
  - Pump artificially delaying ACKs
    - Low process waits for ACK regardless of whether buffer is full
  - Low cannot tell whether buffer is full
    - Closes covert channel
  - Not optimal (processes may wait even when unnecessary)

How to Fix (*con't*)

- Case 2: Low process sends messages faster than High process can remove them
  - Maximizes performance
  - Opens covert channel
- Case 3: Pump, processes handle messages at same rate
  - Decreases bandwidth of covert channel, increases performance
  - Opens covert channel, sub-optimal performance

Adding Noise to Direct Channel

- Kang, Moskowitz: do this in such a way as to approximate case 3
  - Reduces covert channel's capacity to $1/nr$
    - r time between Low process sending message and receiving ACK when buffer not full
  - Conclusion: pump substantially reduces capacity of covert channel between High, Low processes when compared with direct connection

Key Points

- Confinement problem: prevent leakage of information
  - Solution: separation and/or isolation
- Shared resources offer paths along which information can be transferred
- Covert channels difficult if not impossible to eliminate
  - Bandwidth can be greatly reduced, however!