Running Untrusted Application Code: Sandboxing

Running untrusted code

- We often need to run buggy/unstrusted code:
  - programs from untrusted Internet sites:
    - toolbars, viewers, codecs for media player
  - old or insecure applications: ghostview, outlook
  - legacy daemons: sendmail, bind
  - honeypots
- Goal: if application "misbehaves," kill it

Approach: confinement

- **Confinement**: ensure application does not deviate from pre-approved behavior

- Can be implemented at many levels:
  - **Hardware**: run application on isolated hw (air gap)
    - difficult to manage
  - **Virtual machines**: isolate OS's on single hardware
  - **System call interposition**:
    - Isolates a process in a single operating system
  - Isolating threads sharing same address space:
    - **Software Fault Isolation** (SFI)
  - Application specific: e.g. browser-based confinement

Implementing confinement

- Key component: **reference monitor**
  - **Mediates requests** from applications
    - Implements protection policy
    - Enforces isolation and confinement
  - Must **always** be invoked:
    - Every application request must be mediated
  - **Tamperproof**:
    - Reference monitor cannot be killed
    - … or if killed, then monitored process is killed too
  - **Small** enough to be analyzed and validated

A simple example: chroot

- Often used for "guest" accounts on ftp sites

- To use do: (must be root)

chroot /tmp/guest	root dir "/" is now "/tmp/guest"
su guest	EUID set to "guest"

- Now "/tmp/guest" is added to file system accesses for applications in jail

  open("/etc/passwd", "r") ⇒

  open("/tmp/guest/etc/passwd", "r")

⇒ application cannot access files outside of jail

Jailkit

Problem: all utility progs (ls, ps, vi) must live inside jail

- **jailkit** project: auto builds files, libs, and dirs needed in jail environment
  - **jk_init**: creates jail environment
  - **jk_check:** checks jail env for security problems
    - checks for any modified programs,
    - checks for world writable directories, etc.
  - **jk_lsh**: restricted shell to be used inside jail

- **note:** simple chroot jail does not limit network access

Escaping from jails

- Early escapes: relative paths

  open("../../etc/passwd", "r") ⇒

  open("/tmp/guest/../../etc/passwd", "r")

- **chroot** should only be executable by root
  - otherwise jailed app can do:
    - create dummy file "/aaa/etc/passwd"
    - run chroot "/aaa"
    - run su root to become root

(bug in Ultrix 4.0)

Many ways to escape jail as root

- Create device that lets you access raw disk
- Send signals to non chrooted process
- Reboot system
- Bind to privileged ports

Freebsd jail

- Stronger mechanism than simple chroot
- To run:

  jail jail-path hostname IP-addr cmd

  - calls hardened chroot (no "../../" escape)
  - can only bind to sockets with specified IP address and authorized ports
  - can only communicate with process inside jail
  - root is limited, e.g. cannot load kernel modules

Problems with chroot and jail

- Coarse policies:
  - All or nothing access to file system
  - Inappropriate for apps like web browser
    - Needs read access to files outside jail
      (e.g. for sending attachments in gmail)

- Do not prevent malicious apps from:
  - Accessing network and messing with other machines
  - Trying to crash host OS

System call interposition:

a better approach to confinement

Sys call interposition

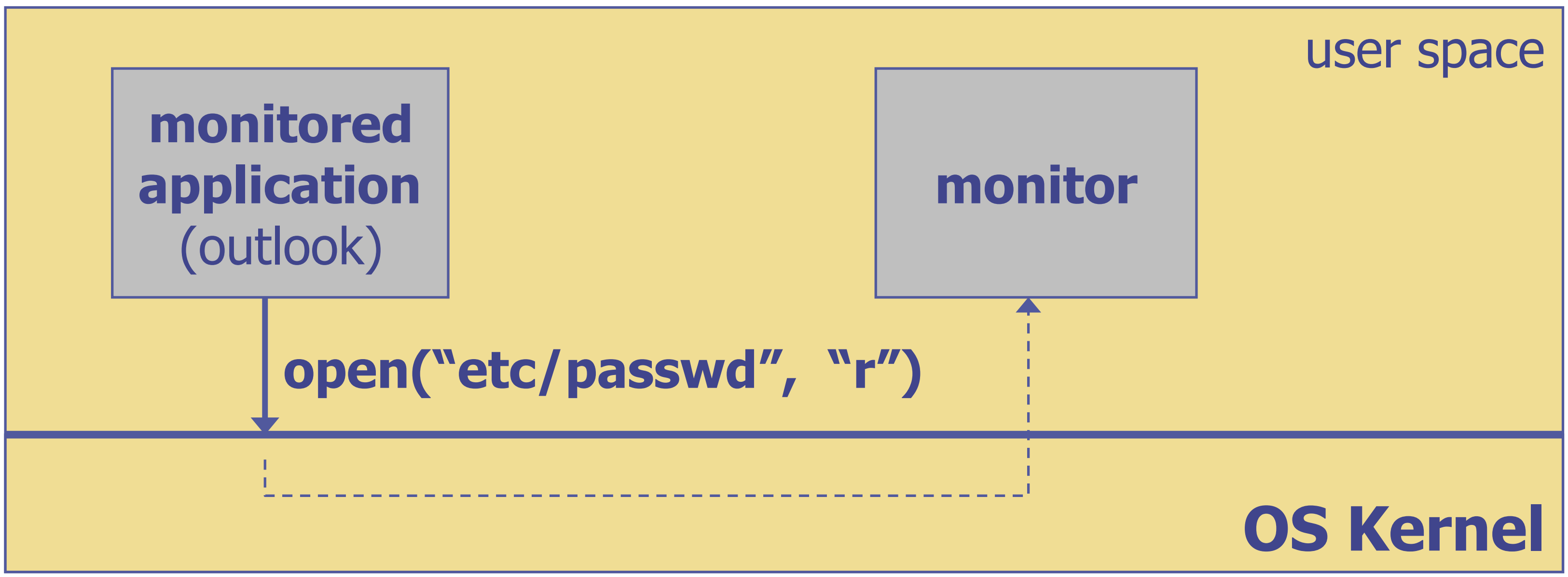
- Observation: to damage host system (i.e. make persistent changes) app must make system calls
  - To delete/overwrite files: unlink, open, write
  - To do network attacks: socket, bind, connect, send
- Idea:
  - monitor app system calls and block unauthorized calls
- Implementation options:
  - Completely kernel space (e.g. GSWTK)
  - Completely user space (e.g. program shepherding)
  - Hybrid (e.g. Systrace)

Initial implementation (Janus)

- Linux ptrace: process tracing

  tracing process calls: **ptrace (… , pid_t pid , …)**

  and wakes up when **pid** makes sys call.



- Monitor kills application if request is disallowed

Complications

- If app forks, monitor must also fork
  - Forked monitor monitors forked app
- If monitor crashes, app must be killed
- Monitor must maintain all OS state associated with app
  - current-working-dir (CWD), UID, EUID, GID
  - Whenever app does “cd path” monitor must also update its CWD
    - otherwise: relative path requests interpreted incorrectly

Problems with ptrace

- Ptrace too coarse for this application
  - Trace all system calls or none
    - e.g. no need to trace “close” system call
  - Monitor cannot abort sys-call without killing app

- Security problems: **race conditions**
  - Example: symlink: me -> mydata.dat

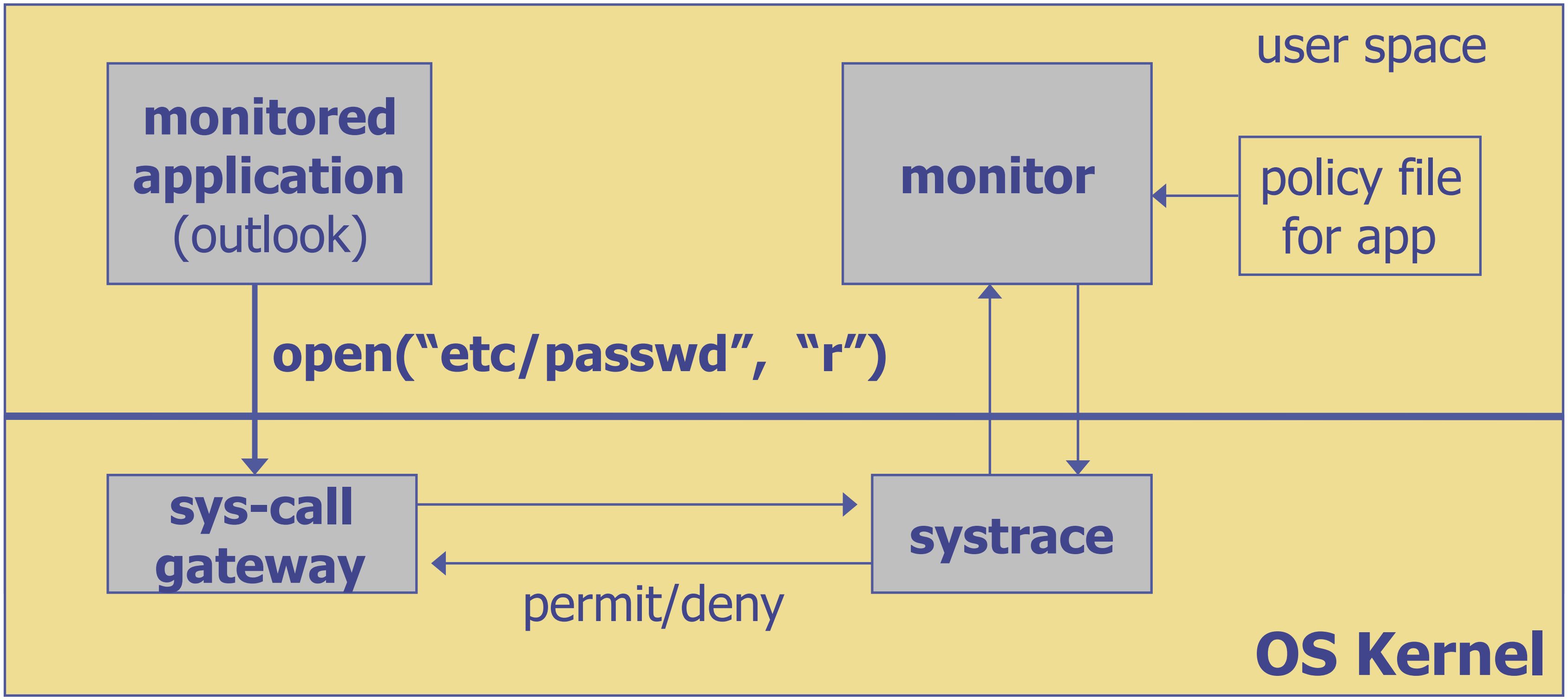
time ↓

proc 1: open(“me”)
monitor checks and authorizes
proc 2: me -> /etc/passwd
OS executes open(“me”)

not atomic

  - Classic TOCTOU bug: time-of-check / time-of-use

Alternate design: systrace



- systrace only forwards monitored sys-calls to monitor (saves context switches)
- systrace resolves sym-links and replaces sys-call path arguments by full path to target
- When app calls execve, monitor loads new policy file

Policy

- Sample policy file:

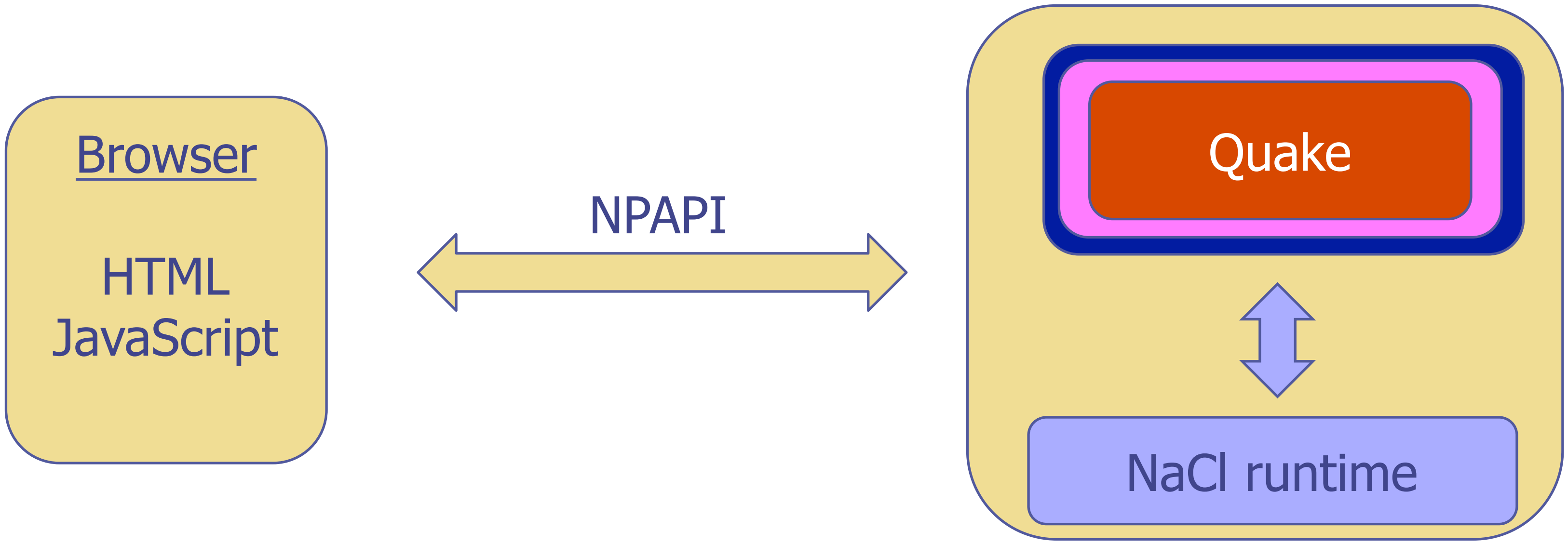
```
path allow  /tmp/*
path deny  /etc/passwd
network deny all
```

- Specifying policy for an app is quite difficult
  - Systrace can auto-gen policy by learning how app behaves on "good" inputs
  - If policy does not cover a specific sys-call, ask user

    … but user has no way to decide

- Difficulty with choosing policy for specific apps (e.g. browser) is main reason this approach is not widely used

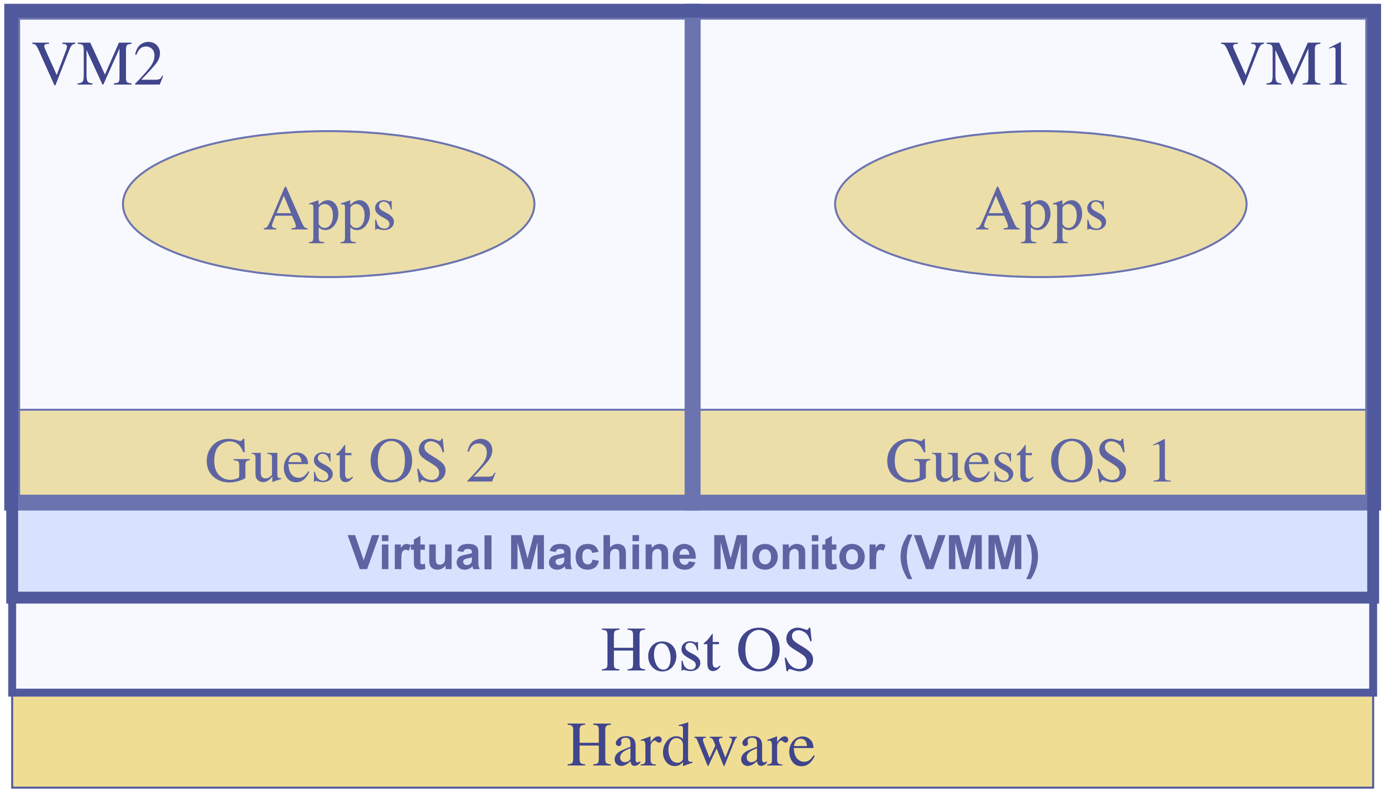
NaCl: a modern day example



- Quake: untrusted x86 code
- Two sandboxes:
  - outer sandbox: restricts capabilities using sys call interposition
  - Inner sandbox: uses x86 memory segmentation to isolate application memory from one another

Confinement using Virtual Machines

Virtual Machines



Example: **NSA NetTop**

- single HW platform used for both classified and unclassified data

Why so popular now?

- VMs in the 1960's:
  - Few computers, lots of users
  - VMs allow many users to shares a single computer
- VMs 1970's – 2000:   non-existent
- VMs since 2000:
  - Too many computers, too few users
    - Print server, Mail server, Web server,
      File server, Database server, …
  - Wasteful to run each service on a different computer
    - VMs save hardware while isolating services
  - More generally: VMs heavily used in cloud computing

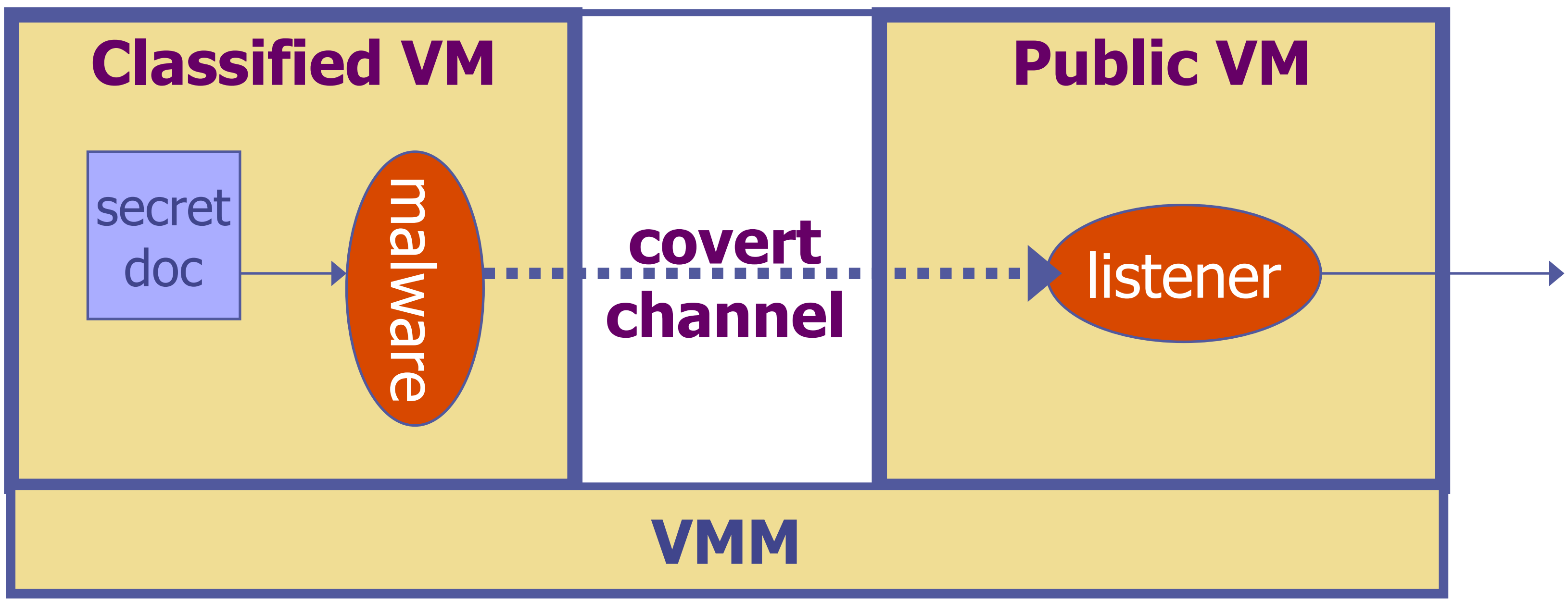
VMM security assumption

- VMM Security assumption:
  - Malware can infect guest OS and guest apps
  - But malware cannot escape from the infected VM
    - Cannot infect host OS
    - Cannot infect other VMs on the same hardware

- Requires that VMM protect itself and is not buggy
  - VMM is much simpler than full OS
  - … but device drivers run in Host OS

Problem: covert channels

- Covert channel: unintended communication channel between isolated components
  - Can be used to leak classified data from secure component to public component



An example covert channel

- Both VMs use the same underlying hardware

- To send a bit $b \in \{0,1\}$ malware does:
  - b= 1: at 1:30.00am do CPU intensive calculation
  - b= 0: at 1:30.00am do nothing

- At 1:30.00am listener does a CPU intensive calculation and measures completion time
  - Now $b = 1 \Leftrightarrow$ completion-time > threshold

- Many covert channel exist in running system:
  - File lock status, cache contents, interrupts, …
  - Very difficult to eliminate

VMM Introspection: [GR’03]

protecting the anti-virus system

Intrusion Detection / Anti-virus

- Runs as part of OS kernel and user space process
  - Kernel root kit can shutdown protection system
  - Common practice for modern malware
- Standard solution: **run IDS system in the network**
  - Problem: insufficient visibility into user's machine
- Better: **run IDS as part of VMM (protected from malware)**
  - VMM can monitor virtual hardware for anomalies
  - VMI: Virtual Machine Introspection
    - Allows VMM to check Guest OS internals

Sample checks

Stealth malware:

- Creates processes that are invisible to "ps"
- Opens sockets that are invisible to "netstat"

1. Lie detector check
   - Goal: detect stealth malware that hides processes and network activity
   - Method:
     - VMM lists processes running in GuestOS
     - VMM requests GuestOS to list processes (e.g. ps)
     - If mismatch, kill VM

Sample checks

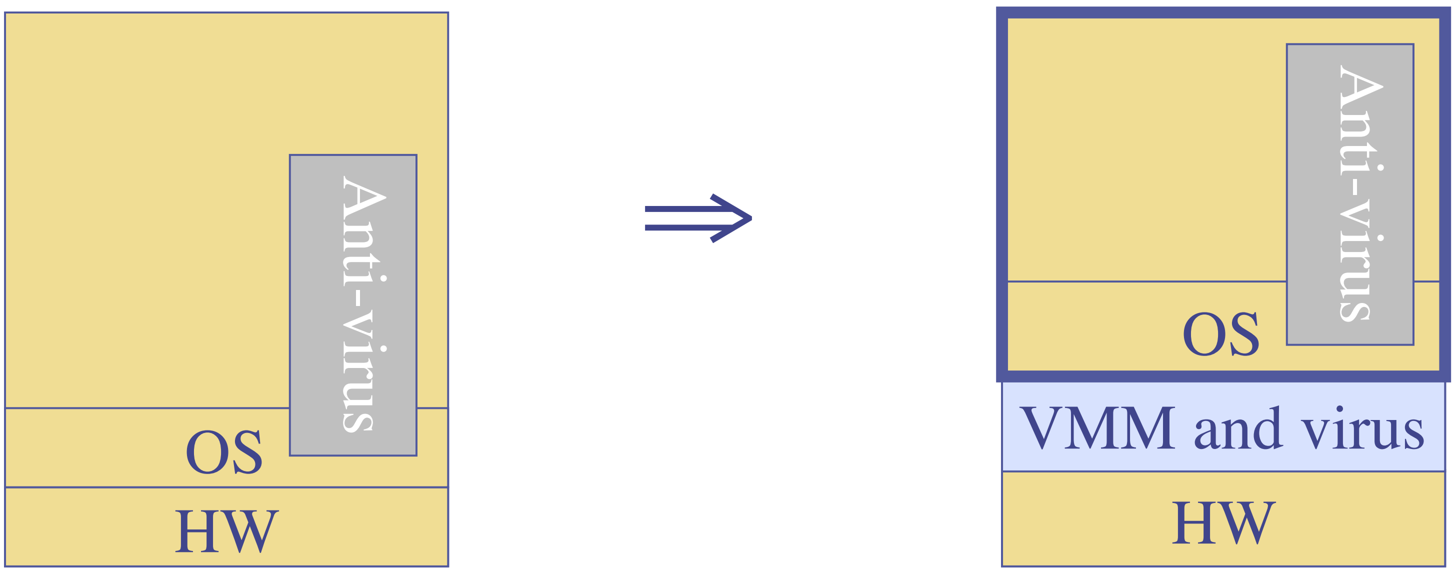
2. Application code integrity detector
   - VMM computes hash of user app-code running in VM
   - Compare to whitelist of hashes
     - Kills VM if unknown program appears
3. Ensure GuestOS kernel integrity
   - example: detect changes to sys_call_table
4. Virus signature detector
   - Run virus signature detector on GuestOS memory
5. Detect if GuestOS puts NIC in promiscuous mode

Subvirt:

subvirting VMM confinement

Subvirt

- Virus idea:
  - Once on the victim machine, install a malicious VMM
  - Virus hides in VMM
  - Invisible to virus detector running inside VM



The MATRIX

The Red Pill
The Blue Pill

VM Based Malware (blue pill virus)

- VMBR: **a virus that installs a malicious VMM (hypervisor)**
- **Microsoft Security Bulletin: (Oct, 2006)** http://www.microsoft.com/whdc/system/platform/virtual/CPUVirtExt.mspx
  - Suggests disabling hardware virtualization features by default for client-side systems
- **But VMBRs are easy to defeat**
  - A guest OS can detect that it is running on top of VMM

VMM Detection

- Can an OS detect it is running on top of a VMM?
- Applications:
  - Virus detector can detect VMBR
  - Normal virus (non-VMBR) can detect VMM
    - refuse to run to avoid reverse engineering
  - Software that binds to hardware (e.g. MS Windows) can refuse to run on top of VMM
  - DRM systems may refuse to run on top of VMM

VMM detection (red pill techniques)

1. VM platforms often emulate simple hardware
   - VMWare emulates an ancient i440bx chipset
     … but report 8GB RAM, dual Opteron CPUs, etc.

2. VMM introduces time latency variances
   - Memory cache behavior differs in presence of VMM
   - Results in relative latency in time variations for any two operations

3. VMM shares the TLB with GuestOS
   - GuestOS can detect reduced TLB size

… and many more methods **[GAWF'07]**

VMM Detection

Bottom line: **The perfect VMM does not exist**

- VMMs today (e.g. VMWare) focus on:
  - Compatibility: ensure off the shelf software works
  - Performance: minimize virtualization overhead
- VMMs do not provide **transparency**
  - **Anomalies reveal existence of VMM**

Software Fault Isolation

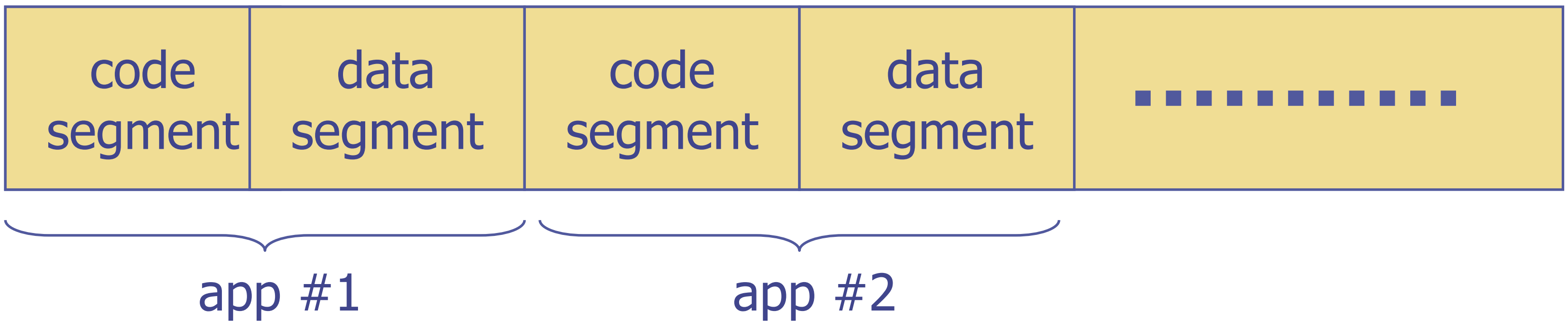
Software Fault Isolation

- Goal: confine apps running in same address space
  - Codec code should not interfere with media player
  - Device drivers should not corrupt kernel

- Simple solution: runs apps in separate address spaces
  - Problem: slow if apps communicate frequently
    - requires context switch per message

Software Fault Isolation

- SFI approach:
  - Partition process memory into segments



  - Locate unsafe instructions: jmp, load, store
    - At compile time, add guards before unsafe instructions
    - When loading code, ensure all guard are present

Segment matching technique

- Designed for MIPS processor.   Many registers available.

- dr1, dr2:   de
  - Compiler p
  - dr2 contai

> Guard ensures code does not load data from another segment

- Indirect load instructi **R12 ← [addr]** becomes:

```
dr1 ← addr
scratch-reg ← (dr1 >> 20)          : get segment ID
compare scratch-reg  and  dr2      : validate seg. ID
trap if not equal
R12 ← [addr]                       : do load
```

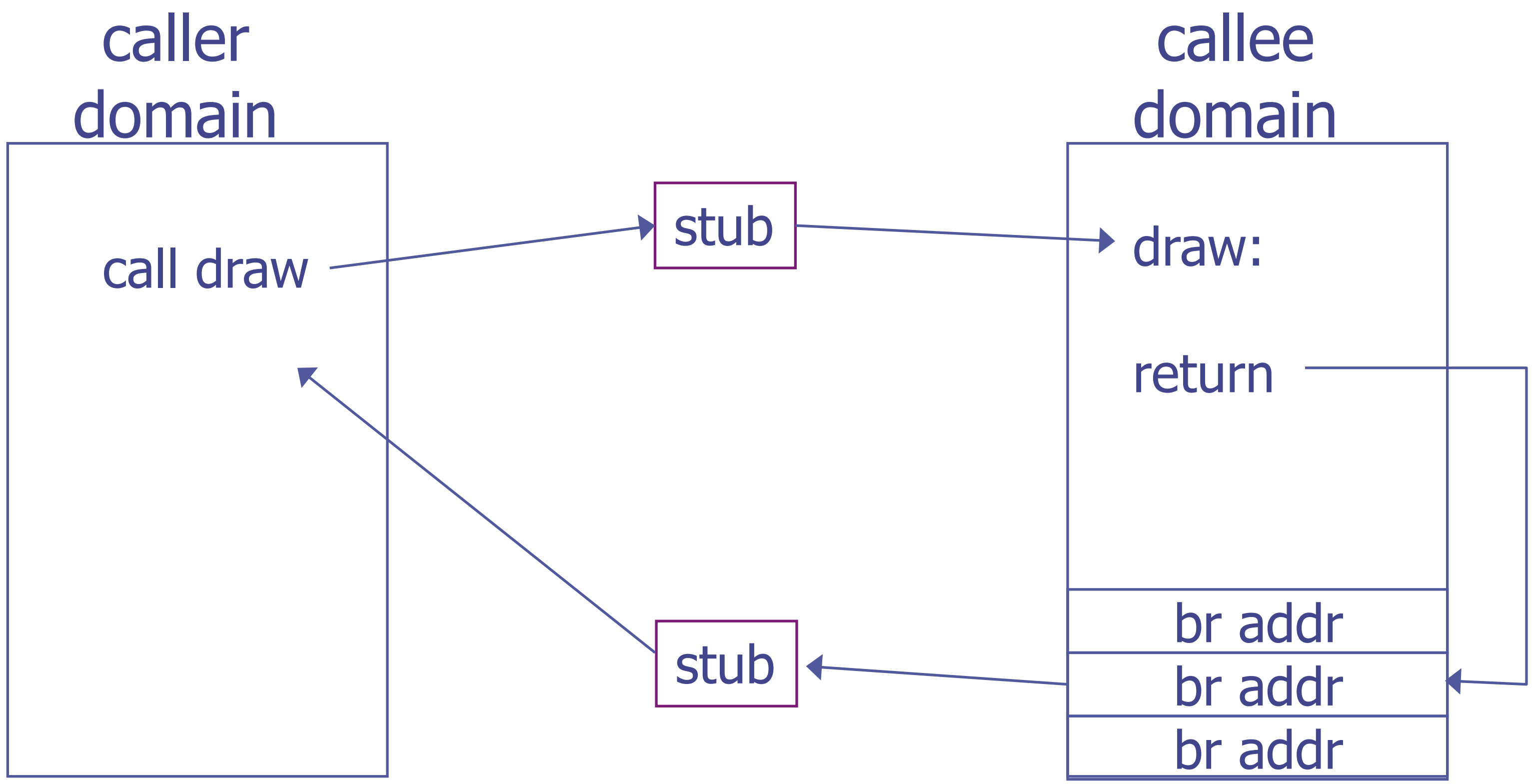
Address sandboxing technique

- dr2: holds segment ID
- Indirect load instruction **R12 ← [addr]** becomes:

dr1 ← addr & segment-mask	: zero out seg bits
dr1 ← dr1 | dr2	: set valid seg ID
R12 ← [dr1]	: do load

- Fewer instructions than segment matching
  … but does not catch offending instructions
- Lots of room for optimizations: reduce # of guards

Cross domain calls



- Only stubs allowed to make croos-domain jumps
- Jump table contains allowed exit points from callee
  - Addresses are hard coded, read-only segment

SFI: concluding remarks

- For shared memory: use virtual memory hardware
  - Map same physical page to two segments in addr space

- Performance
  - Usually good: mpeg_play, 4% slowdown

- Limitations of SFI: harder to implement on x86 :
  - variable length instructions: unclear where to put guards
  - few registers: can't dedicate three to SFI
  - many instructions affect memory: more guards needed

Summary

- Many sandboxing techniques:
  - Physical air gap,
  - Virtual air gap (VMMs),
  - System call interposition
  - Software Fault isolation
  - Application specific (e.g. Javascript in browser)

- Often complete isolation is inappropriate
  - Apps need to communicate through regulated interfaces

- Hardest aspect of sandboxing:
  - Specifying policy: what can apps do and not do

THE END