Isolation

The confinement principle

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Running untrusted code

We often need to run buggy/unstrusted code:

- programs from untrusted Internet sites:
  - apps, extensions, plug-ins, codecs for media player
- exposed applications: pdf viewers, outlook
- legacy daemons: sendmail, bind
- honeypots

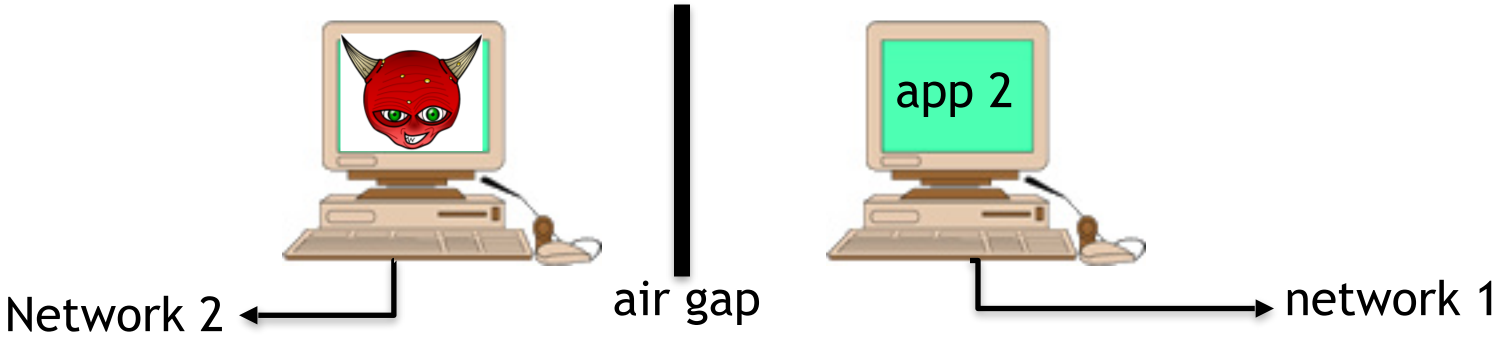
<u>Goal</u>: if application "misbehaves" ⇒ kill it

Approach: confinement

<u>**Confinement**</u>: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Hardware**: run application on isolated hw (air gap)



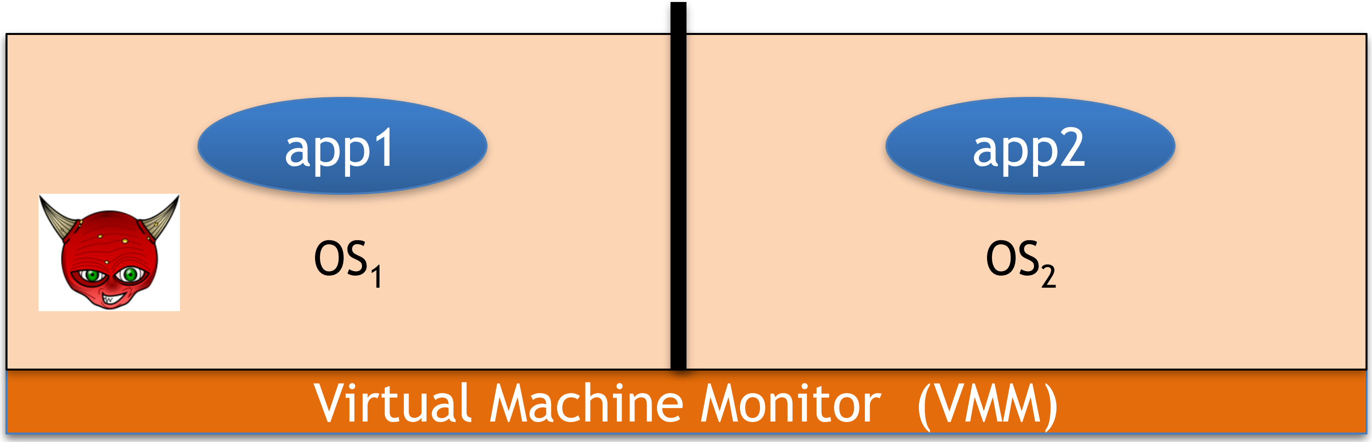
⇒ difficult to manage

Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

– **Virtual machines**:   isolate OS's on a single machine



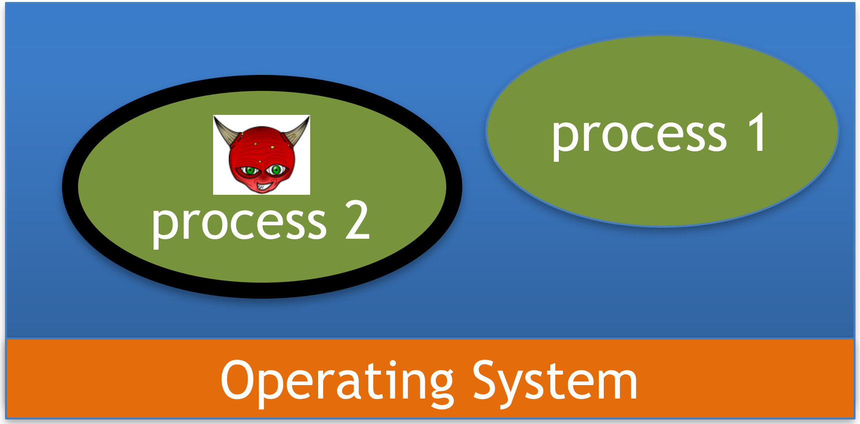
Approach: confinement

<u>Confinement</u>: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

– **Process:** System Call Interposition

Isolate a process in a single operating system



Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

- **Threads:** Software Fault Isolation (SFI)
  - Isolating threads sharing same address space

- **Application**: e.g. browser-based confinement

Implementing confinement

Key component: **reference monitor**

- **Mediates requests** from applications
  - Implements protection policy
  - Enforces isolation and confinement
- Must <u>**always**</u> 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 old example: chroot

Often used for "guest" accounts on ftp sites

To use do: (must be root)

```
chroot /tmp/guest
su guest
```

root dir "/" is now "/tmp/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 env
  - **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)

Freebsd jail

Stronger mechanism than simple chroot

<u>To run</u>: **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 processes inside jail
- root is limited, e.g. cannot load kernel modules

Problems with chroot and jail

Coarse policies:

- All or nothing access to parts of file system
- Inappropriate for apps like a web browser
  - Needs read access to files outside jail (e.g. for sending attachments in Gmail)

Does not prevent malicious apps from:

- Accessing network and messing with other machines
- Trying to crash host OS

Isolation

System Call Interposition

System call interposition

Observation: to damage host system (e.g. 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's system calls and block unauthorized calls

Implementation options:

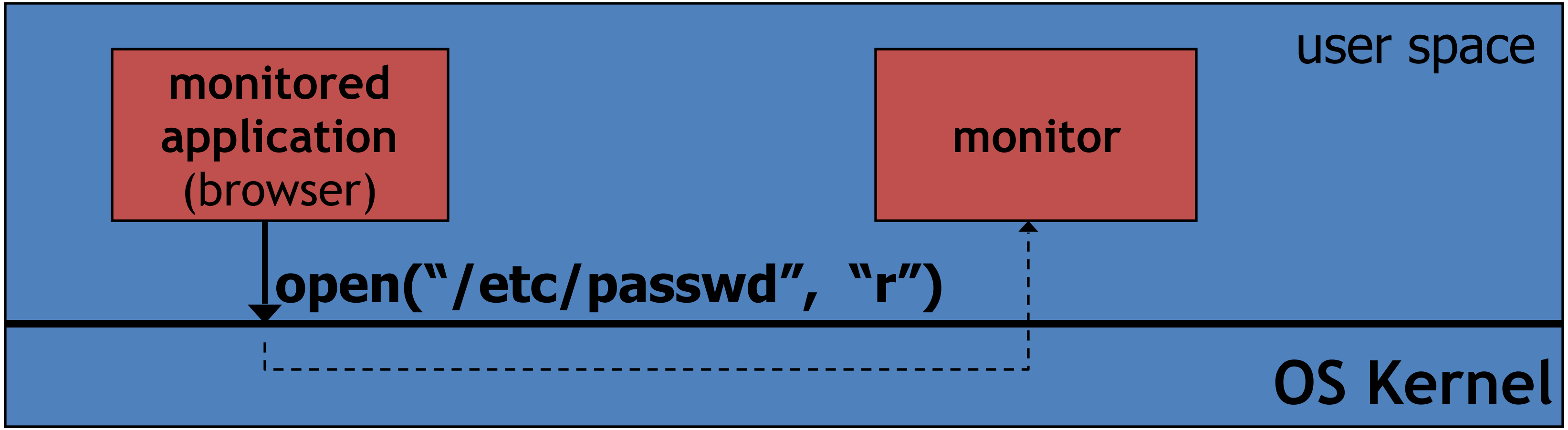
- Completely kernel space (e.g. GSWTK)
- Completely user space (e.g. program shepherding)
- Hybrid (e.g. Systrace)

Initial implementation (Janus) [GWTB'96]

Linux **ptrace**: process tracing

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

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



Monitor kills application if request is disallowed

Example policy

Sample policy file (e.g., for PDF reader)

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

Manually specifying policy for an app can be difficult:

– Recommended default policies are available

… can be made more restrictive as needed.

Complications

```
cd("/tmp")
open("passwd", "r")

cd("/etc")
open("passwd", "r")
```

- 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**
  - When app does "cd path" monitor must update its CWD
    - otherwise: relative path requests interpreted incorrectly

Problems with ptrace

Ptrace is not well suited for this application:

– Trace all system calls or none

inefficient: no need to trace "close" system call

– Monitor cannot abort sys-call without killing app

Security problems: **race conditions**

– <u>Example</u>: symlink: me ⟶ mydata.dat

time

proc 1: open("me")

monitor checks and authorizes

proc 2: me ⟶ /etc/passwd

OS executes open("me")

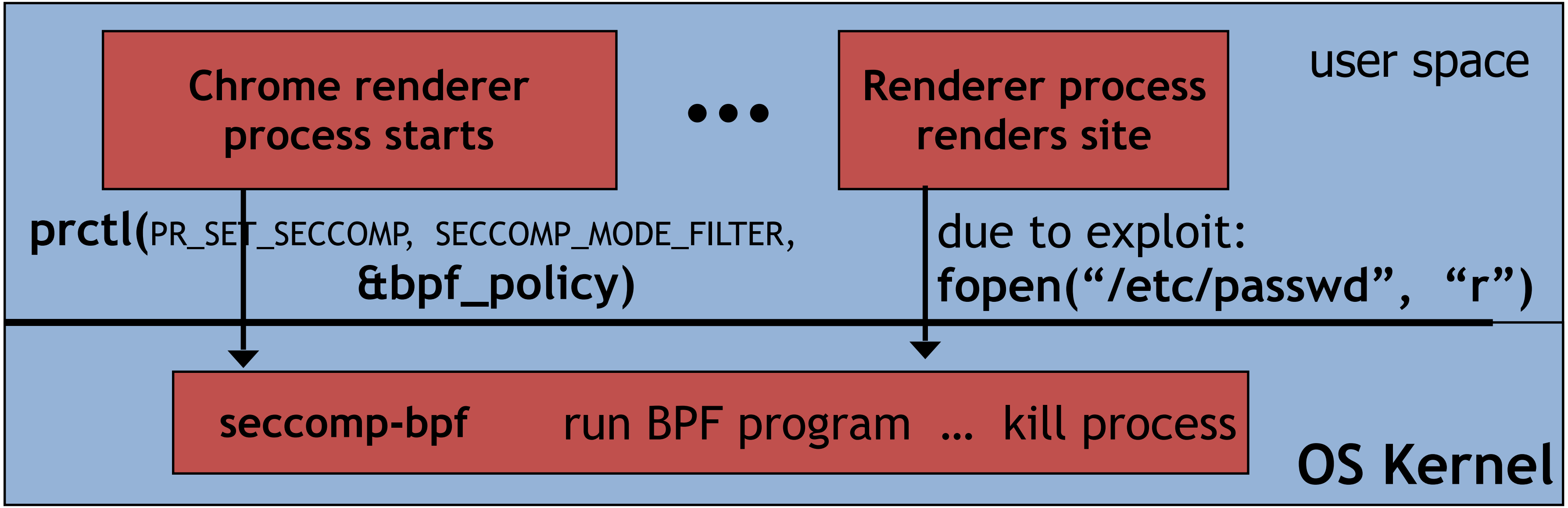
not atomic

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

SCI in Linux: seccomp-bpf

Seccomp-BPF: Linux kernel facility used to filter process sys calls

- Sys-call filter written in the BPF language (use BPFC compiler)
- Used in **Chromium**, in **Docker containers**, …



BPF filters (policy programs)

Process can install multiple BPF filters:

- once installed, filter cannot be removed (all run on every syscall)
- if program forks, child inherits all filters
- if program calls execve, all filters are preserved

BPF filter input: syscall number, syscall args., arch. (x86 or ARM)

Filter returns one of:

- SECCOMP_RET_KILL: kill process
- SECCOMP_RET_ERRNO: return specified error to caller
- SECCOMP_RET_ALLOW: allow syscall

Installing a BPF filter

- Must be called before setting BPF filter.
- Ensures set-UID, set-GID ignored on subequent execve()

  ⇒ attacker cannot elevate privilege

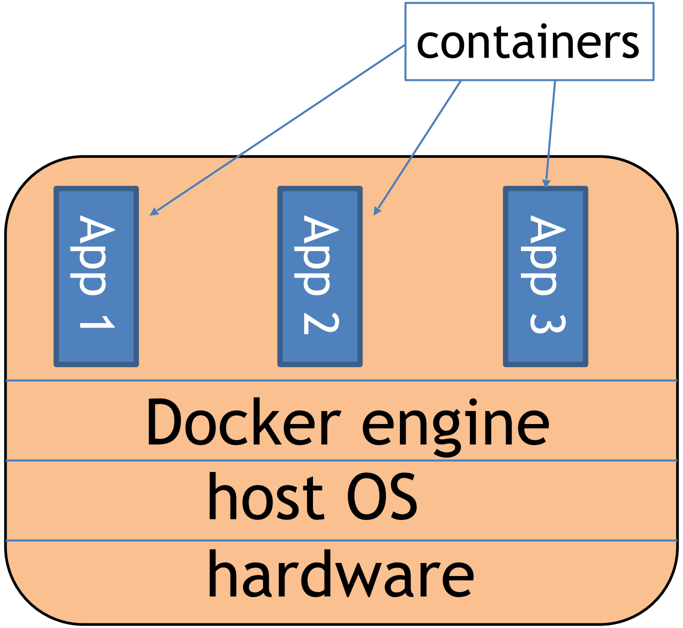
```
int main (int argc , char **argv )  {
      prctl(PR_SET_NO_NEW_PRIVS , 1);
      prctl(PR_SET_SECCOMP,  SECCOMP_MODE_FILTER,  &bpf_policy)
      fopen("file.txt",  "w");
      printf("... will not be printed. \n"
}
```

Kill if call open() for write

Docker: isolating containers using seccomp-bpf

Container: process level isolation

- Container prevented from making sys calls filtered by secomp-BPF
- Whoever starts container can specify BPF policy
  - default policy blocks many syscalls, including ptrace



Docker sys call filtering

Run nginx container with a specific filter called filter.json:

$ docker run --security-opt seccomp=filter.json nginx

Example filter:

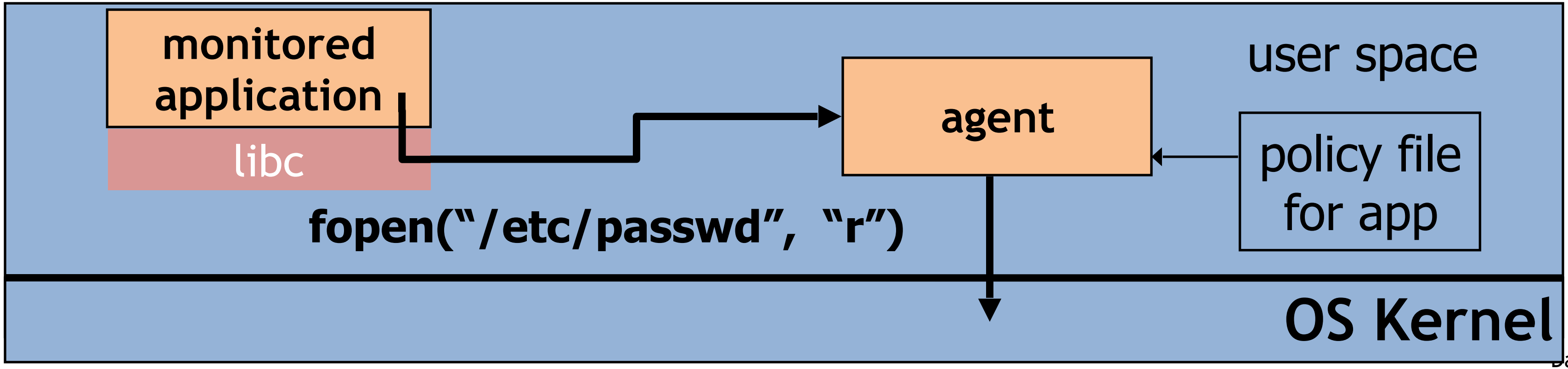
```
"defaultAction":  "SCMP_ACT_ERRNO",   //  deny by default
"syscalls": [
      { "names":  ["accept"],              //  sys-call name
        "action":  "SCMP_ACT_ALLOW",       //  allow (whitelist)
        "args": [ ]  } ,                   // what args to allow
          …
     ]
```

Ostia: SCI with minimal kernel support

Monitored app disallowed from making monitored sys calls
- Minimal kernel change   (… but app can call **close**() itself)

Sys-call delegated to an agent that decides if call is allowed
- Can be done without changing app … using a libc stub

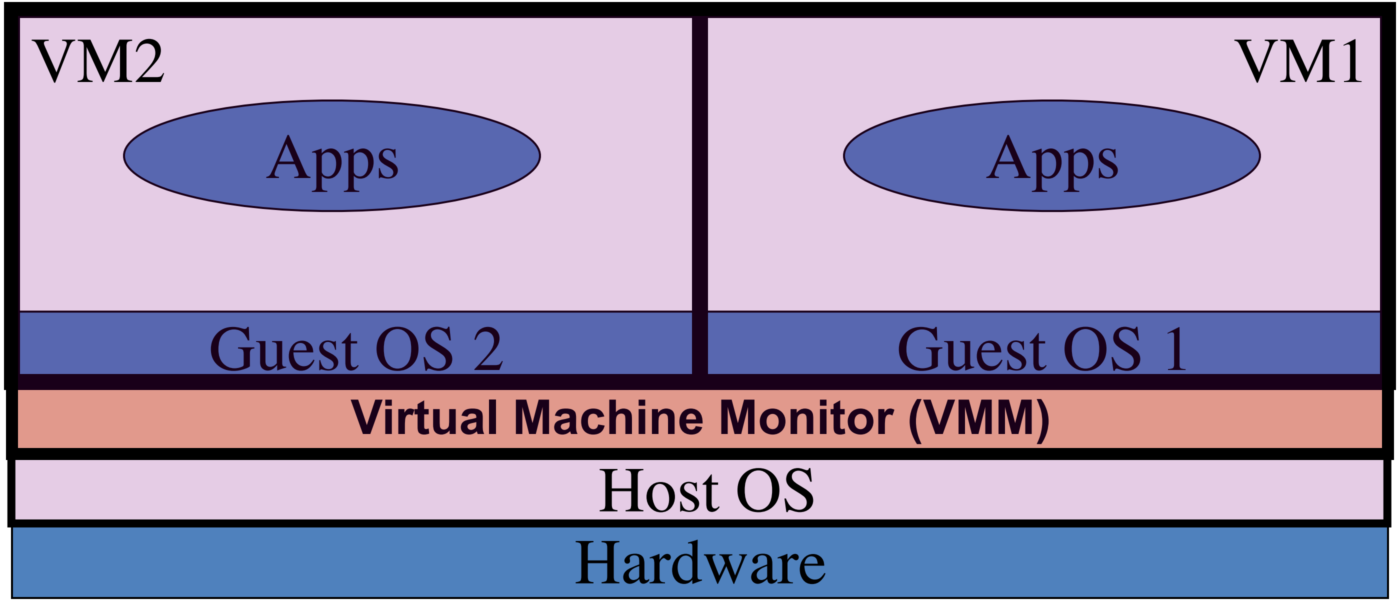
⇒ Incorrect state syncing will not result in policy violation



Isolation

Isolation via 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 , …
- Wasteful to run each service on different hardware
- More generally: VMs heavily used in cloud computing

Hypervisor security assumption

VMM Security assumption:

– Malware can infect guest OS and guest apps

– But malware cannot escape from the infected VM

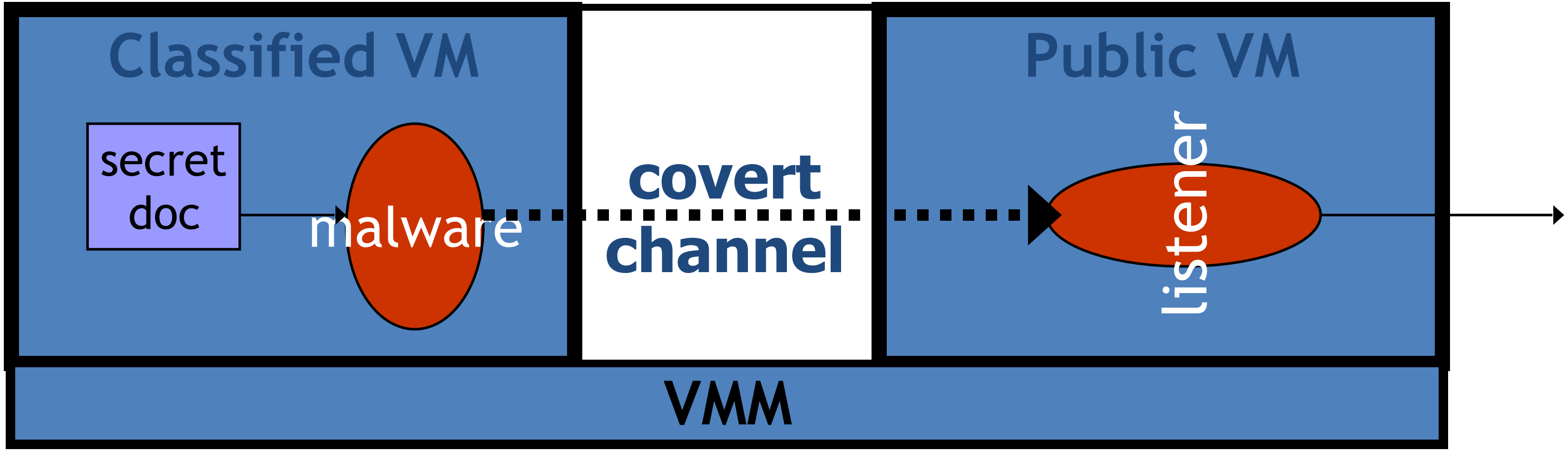
- Cannot infect host OS
- Cannot infect other VMs on the same hardware

Requires that hypervisor protect itself and is not buggy

- (some) hypervisors are much simpler than a full 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 1:00am do CPU intensive calculation
- $b= 0$: at 1:00am do nothing

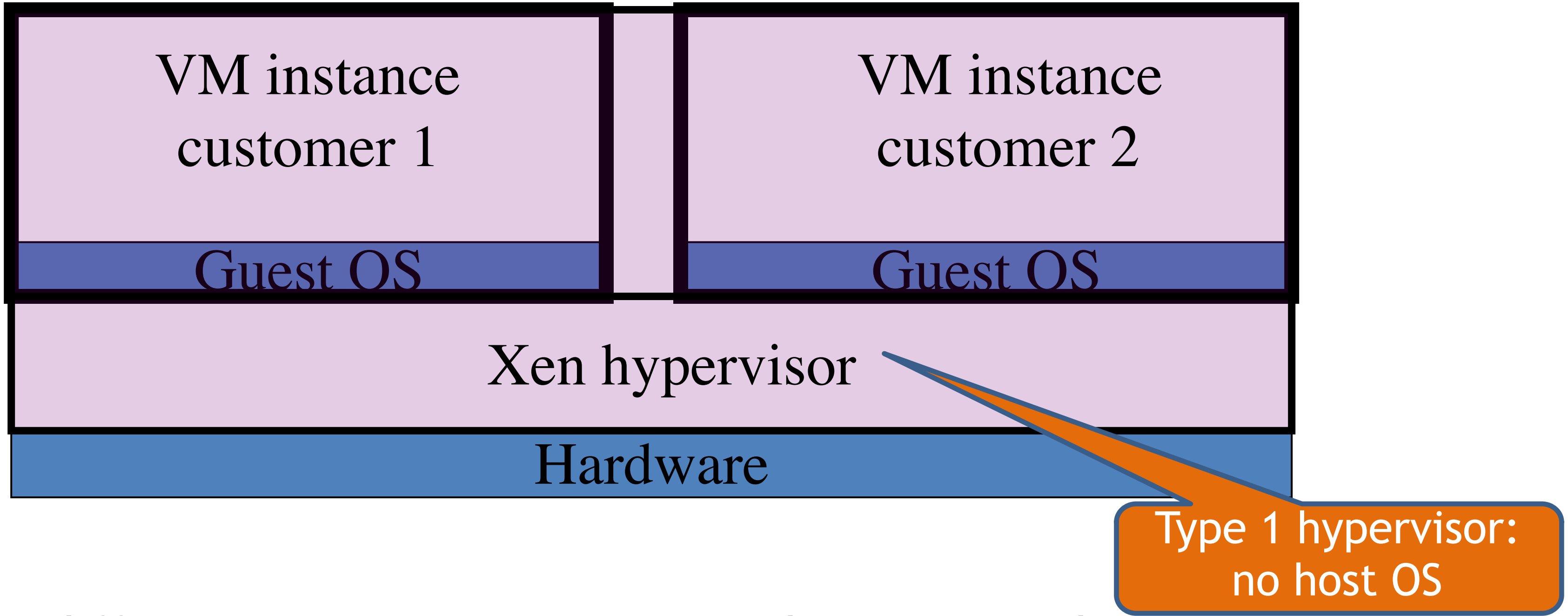
At 1:00am listener does CPU intensive calc. and measures completion time

$$b = 1 \quad \Rightarrow \quad \text{completion-time} > \text{threshold}$$

Many covert channels exist in running system:

- File lock status, cache contents, interrupts, …
- Difficult to eliminate all

VM isolation in practice: cloud



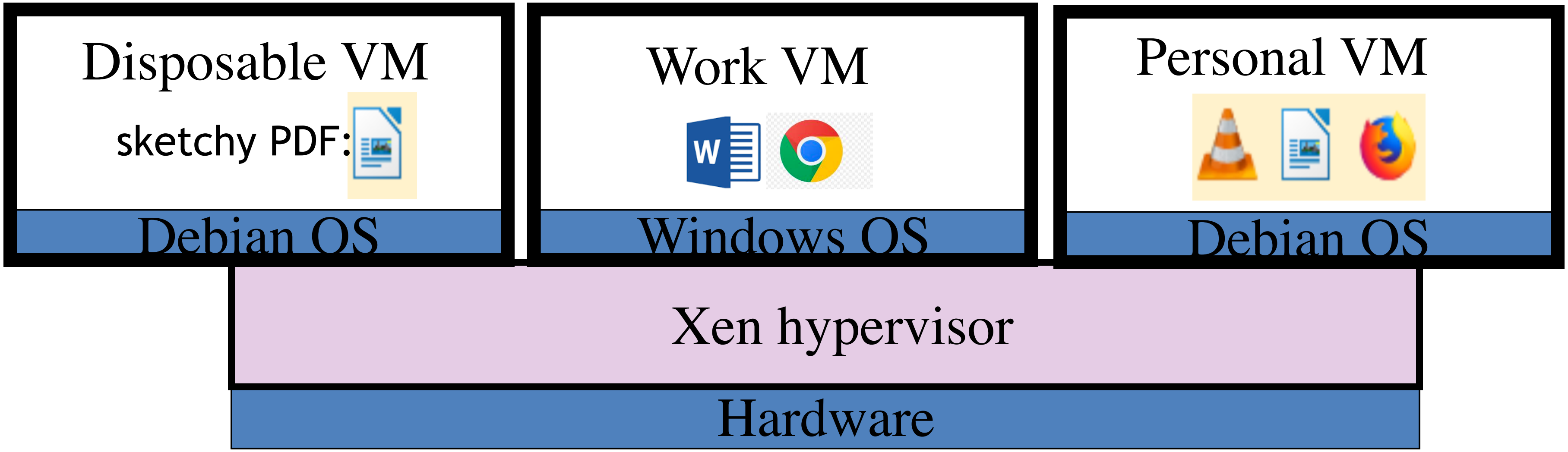
VMs from different customers may run on the same machine

- Hypervisor must isolate VMs … but some info leaks

VM isolation in practice: end-user

Qubes OS: a desktop/laptop OS where everything is a VM

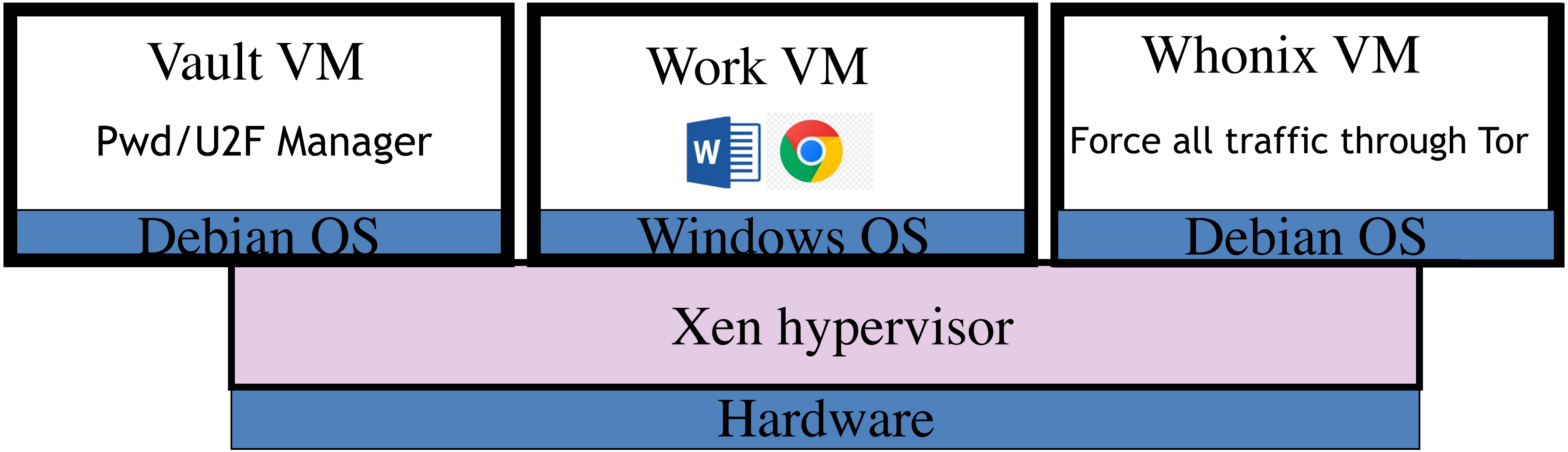
- Runs on top of the Xen hypervisor
- Access to peripherals (mic, camera, usb, …) controlled by VMs



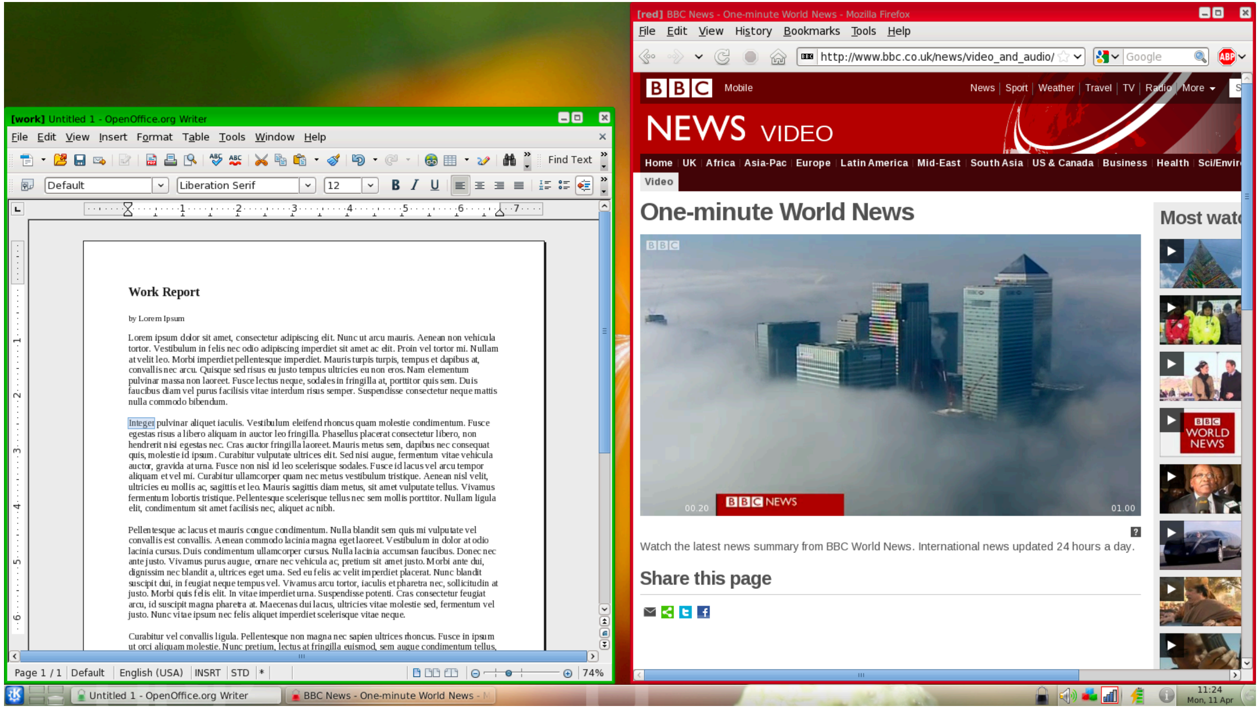
VM isolation in practice: end-user

Qubes OS: a desktop/laptop OS where everything is a VM

- Runs on top of the Xen hypervisor
- Access to peripherals (mic, camera, usb, …) controlled by VMs



Every window frame identifies VM source

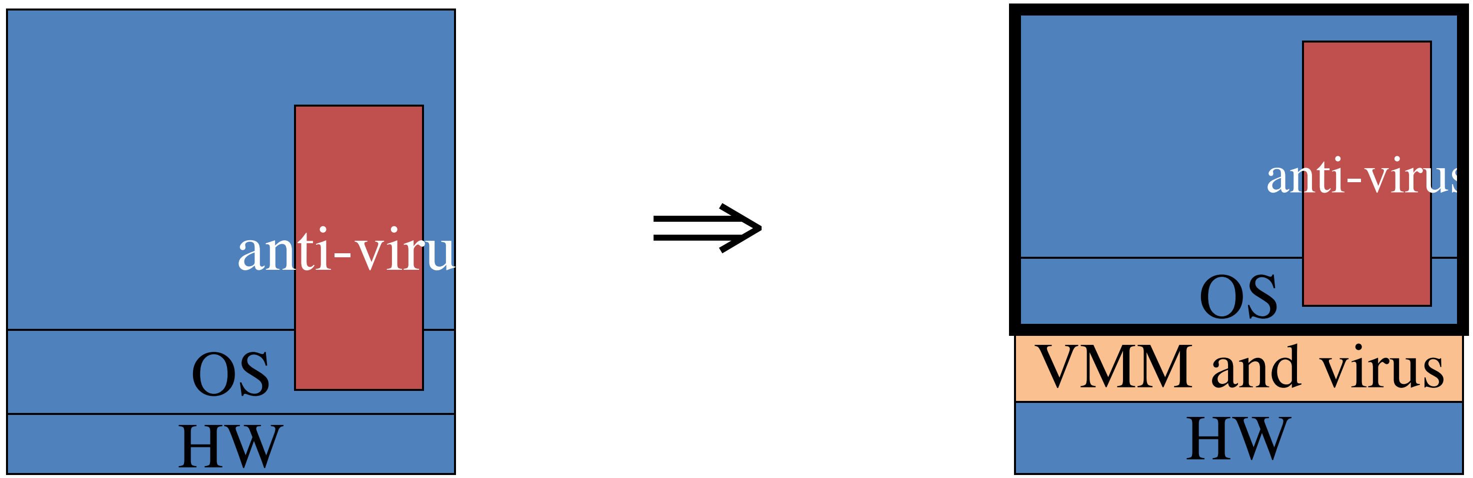


GUI VM ensures frames are drawn correctly

Subvirt [King et al. 2006]

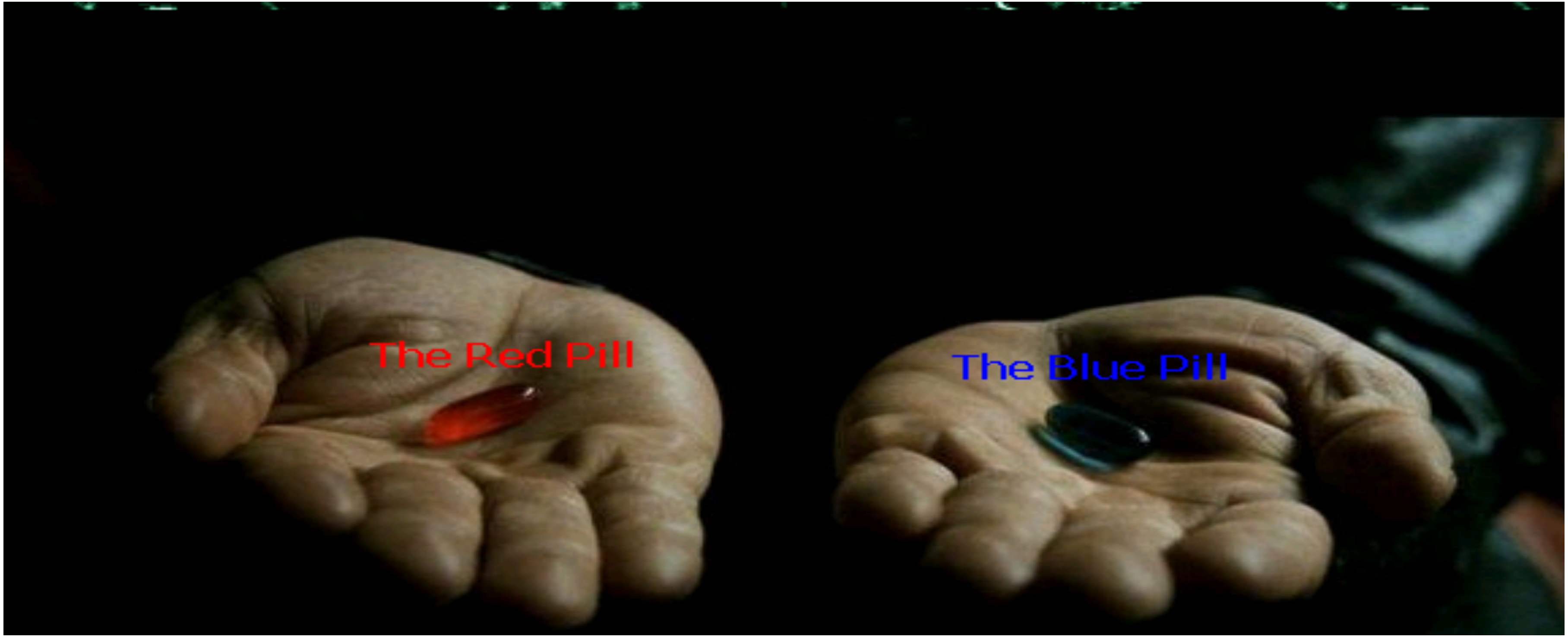
Virus idea:

– Once on victim machine, install a malicious VMM
– Virus hides in VMM
– Invisible to virus detector running inside VM



The MATRIX

Hypervisor detection



VM Based Malware (blue pill virus)

- **VMBR**: a virus that installs a malicious VMM (hypervisor)

- **Microsoft Security Bulletin:**
  - 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 (red pill techniques)

- VM platforms often emulate simple hardware
  - VMWare emulates an ancient i440bx chipset

    … but report 8GB RAM, dual CPUs, etc.

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

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

- … and many more methods **[GAWF'07]**

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

Hypervisor detection in the browser [HBBP'14]

Can we identify malware web sites?

- Approach: crawl web,
  load pages in a browser running in a VM,
  look for pages that damage VM

- The problem: Web page can detect it is running in a VM
  How? Using timing variations in writing to screen

- Malware in web page becomes benign when in a VM
  ⇒ evade detection

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**

Isolation

Software Fault Isolation:
isolating threads

Software Fault Isolation [Whabe et al., 1993]

Goal: confine apps running in <u>same address space</u>

- Codec code should not interfere with media player
- Device drivers should not corrupt kernel

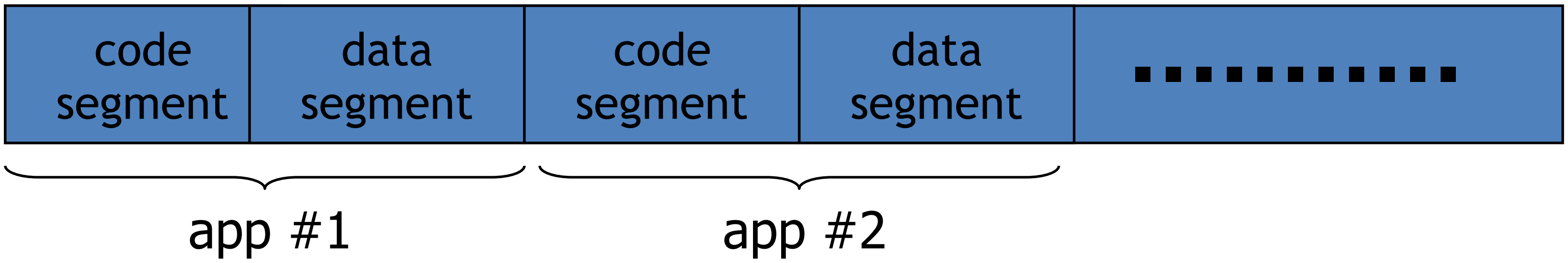
Simple solution: runs apps in separate address spaces

- 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 guards are present

Segment matching technique

- Designed for MIPS processor. Many registers available.
- **dr1, dr2**: dedicated registers not used by the binary.
  - compiler pretend these registers don't exist
  - dr2 contains segment id
- Indirect load instruction **R12<— R[34]** becomes:

```
dr1 <— R34
scratch-reg <— (dr1 >> 20)        :get segment ID
compare scratch-reg and dr2  : validate seg. ID
trap if not equal
R12 <— [dr1]                           : do load
```

Guard ensures code does not load data from another segment

Address sandboxing technique

- **dr2** holds segment ID
- indirect load instruction **R12<— R[34]** becomes:

```
dr1 <-- R34 & 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
- Similar guards places on all unsafe instructions

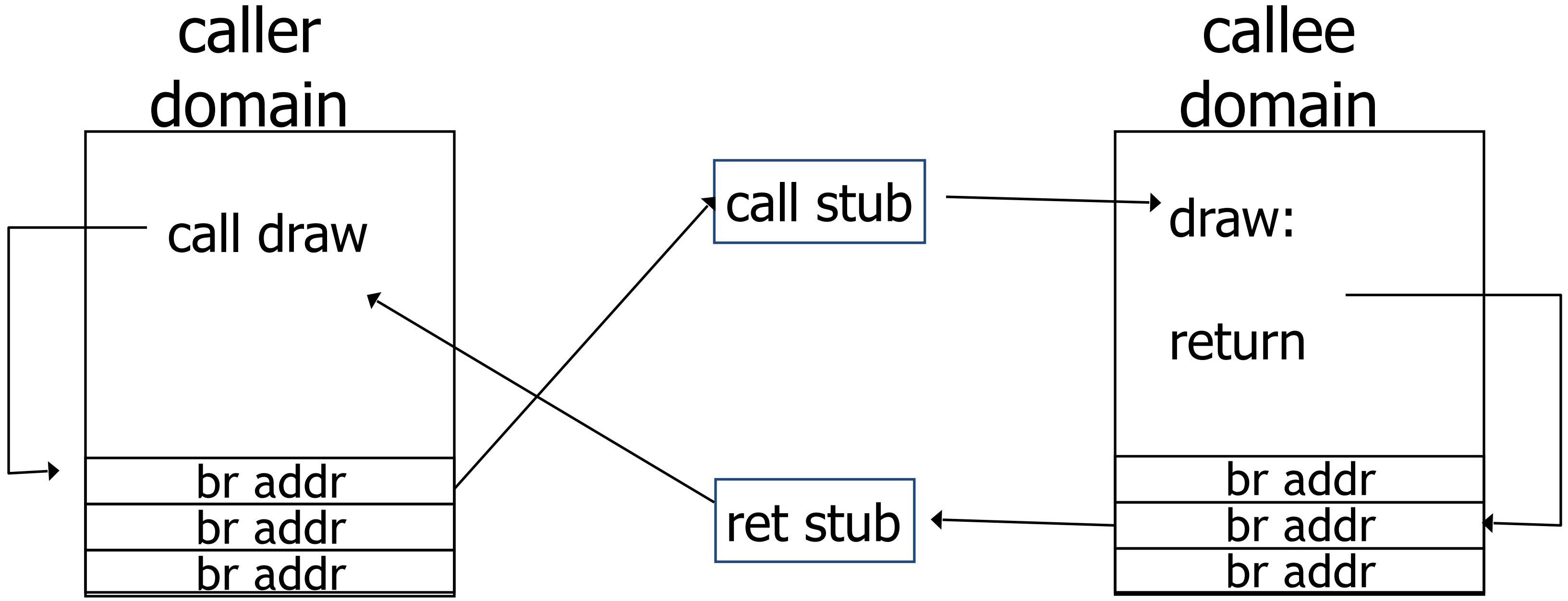
Problem: what if **jmp [addr]** jumps directly into indirect load?

(bypassing guard)

Solution:

jmp guard must ensure **[addr]** does not bypass load guard

Cross domain calls



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

SFI Summary

- Performance
  - Usually good: mpeg_play, 4% slowdown

- <u>Limitations of SFI</u>: 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

Isolation: 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 aspects of sandboxing:
  - Specifying policy: what can apps do and not do
  - Preventing covert channels

THE END