CS155: Computer Security

Isolation

The confinement principle

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and Zakir Durumeric 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:
  - mobile apps, Javascript, browser extensions
- exposed applications: browser, pdf viewer, outlook
- legacy daemons: sendmail, bind
- honeypots

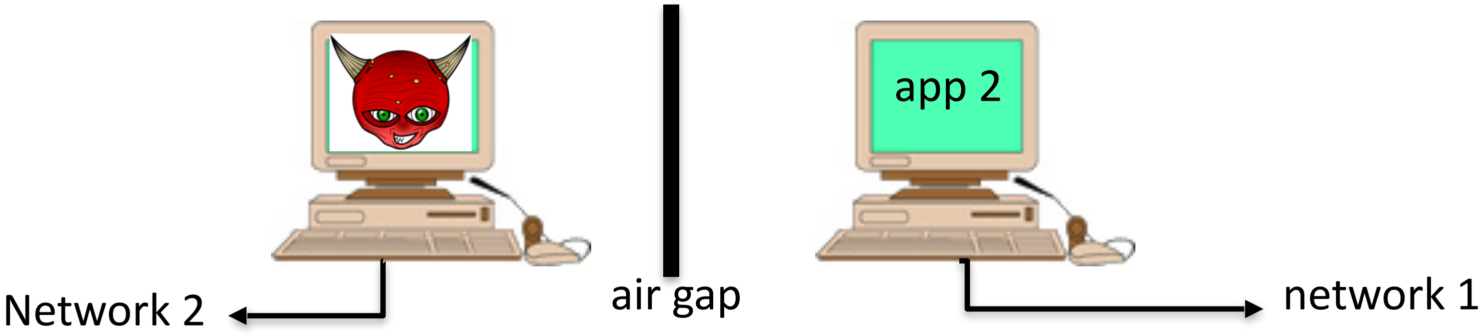
<u>Goal</u>: if application "misbehaves" ⇒ kill it

Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

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



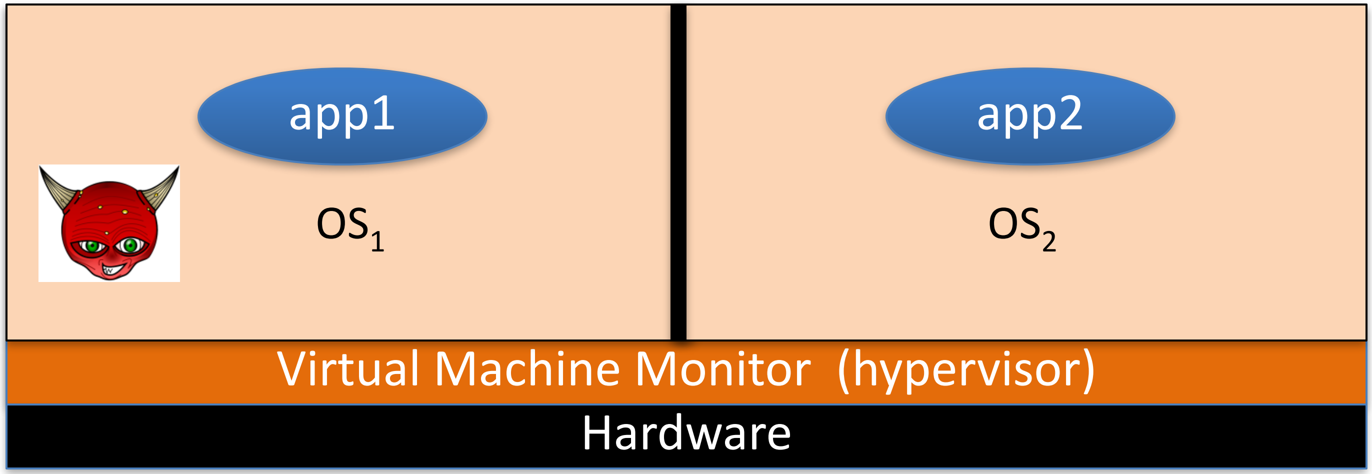
⇒ difficult to manage

Approach: confinement

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

Can be implemented at many levels:

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



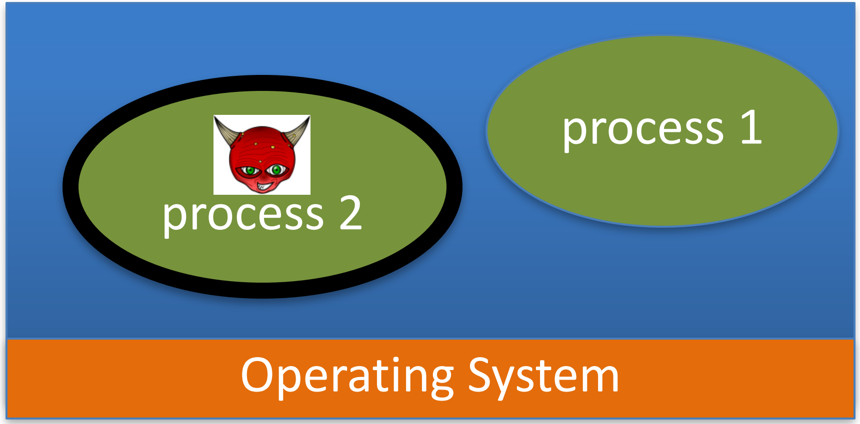
Approach: confinement

Confinement: ensure misbehaving app cannot harm rest of system

Can be implemented at many levels:

– **Process:** System Call Interposition (containers)

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 level confinement**:
  e.g. browser sandbox for Javascript and WebAssembly

Implementing confinement

Key component: **reference monitor**

- **Mediates requests** from applications
  - Enforces confinement
  - Implements a specified protection policy
- 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 old example: chroot

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 every file system accesses:

fopen("/etc/passwd", "r") ⇒

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

⇒ application (e.g., web server) cannot access files outside of jail

Escaping from jails

Early escapes: relative paths

fopen("../../etc/passwd", "r") ⇒

fopen("/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

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 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: sanboxing a process

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:

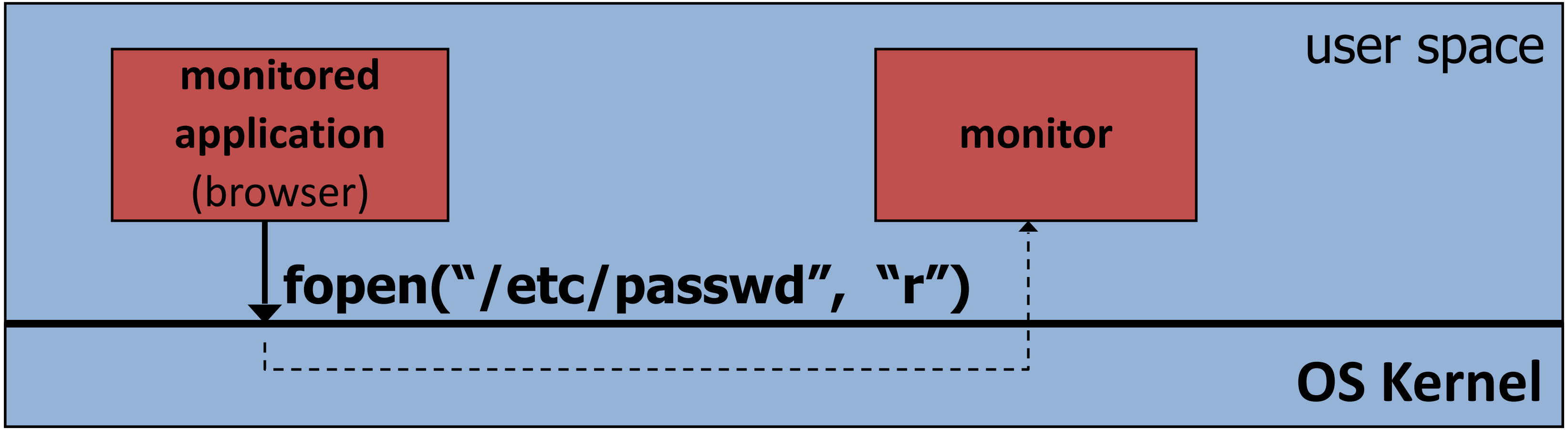
- Completely kernel space (e.g., Linux seccomp)
- Completely user space (e.g., program shepherding)
- Hybrid (e.g., Systrace)

Early 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

- 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

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

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

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**

- Example: symlink: me ⟶ mydata.dat

time ↓

proc 1: open("me")

monitor checks and authorizes

proc 2: me ⟶ /etc/passwd

OS executes open("me")

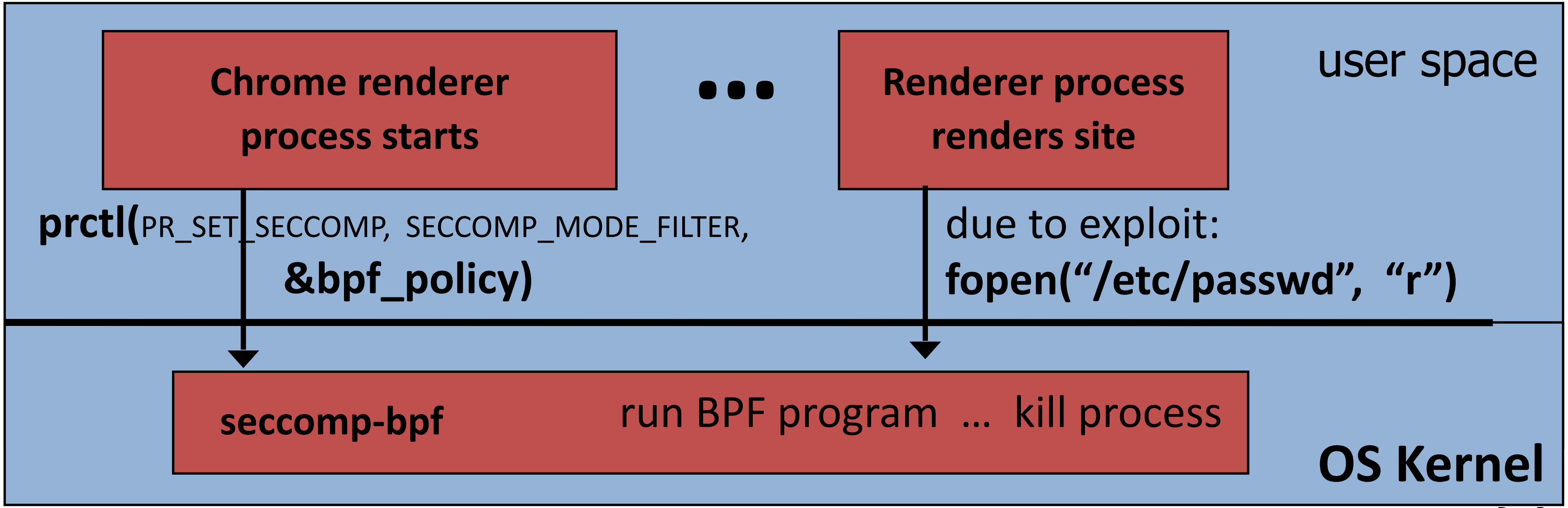
not atomic

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

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" );
}
```

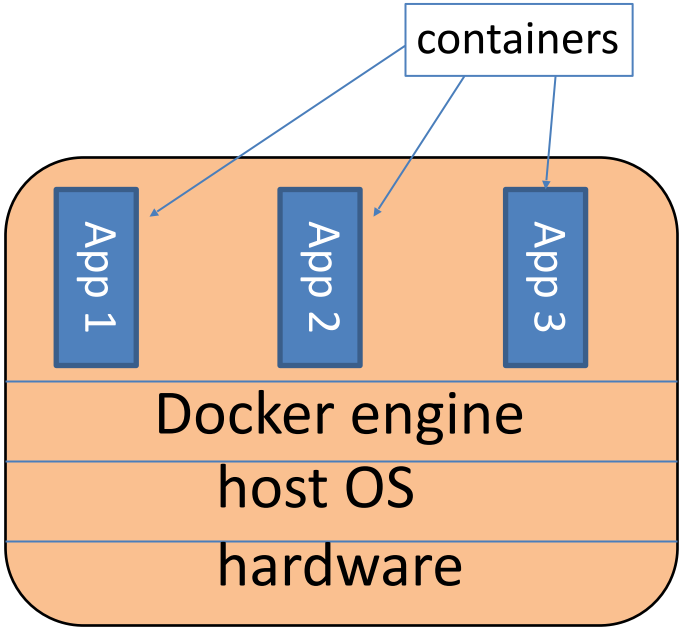
Kill if call open() for write

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

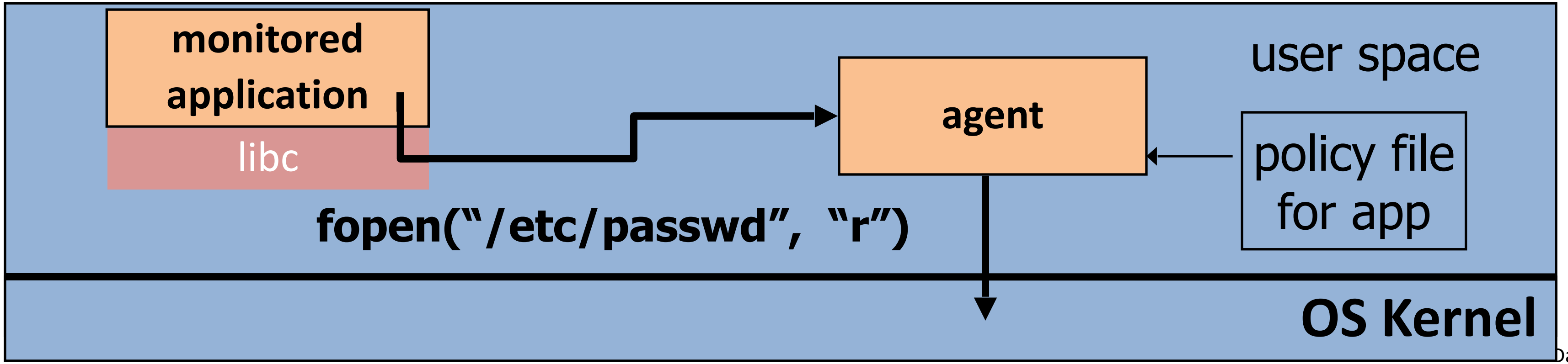
Monitored app disallowed from making monitored sys calls

- 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

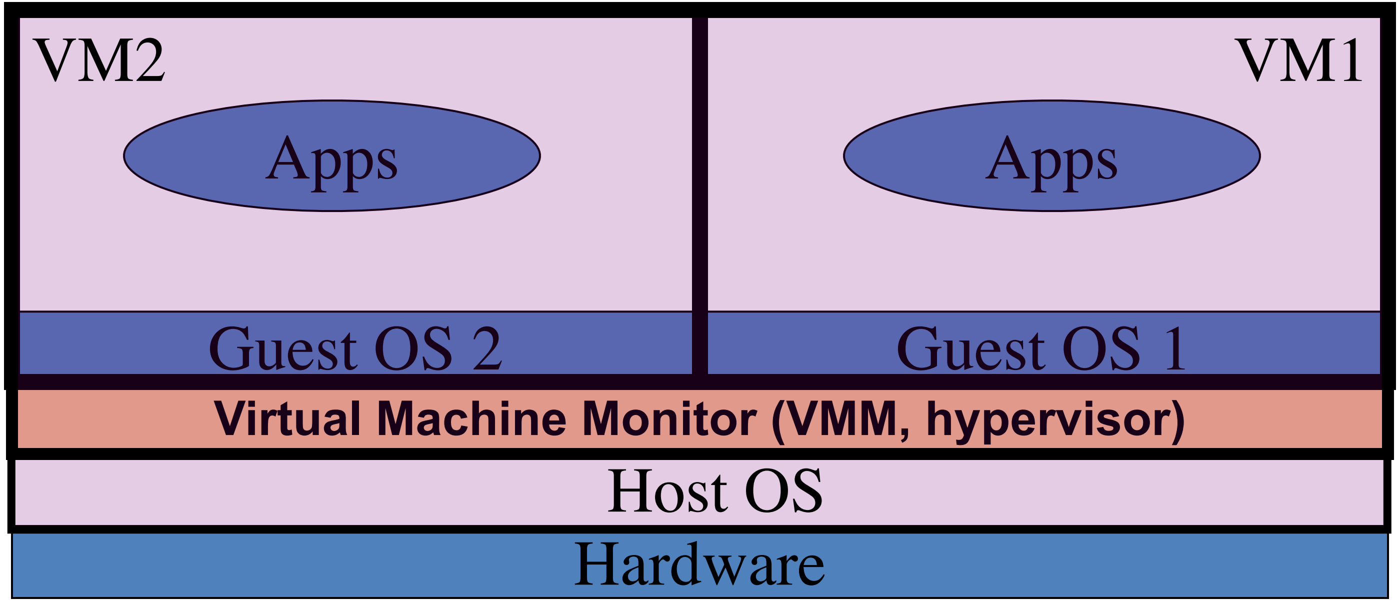
⇒ Incorrect state syncing will not result in policy violation



Isolation

Isolation via Virtual Machines

Virtual Machines



single HW platform with isolated components

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 , …
- VMs heavily used in private and public clouds

Hypervisor security assumption

Hypervisor 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

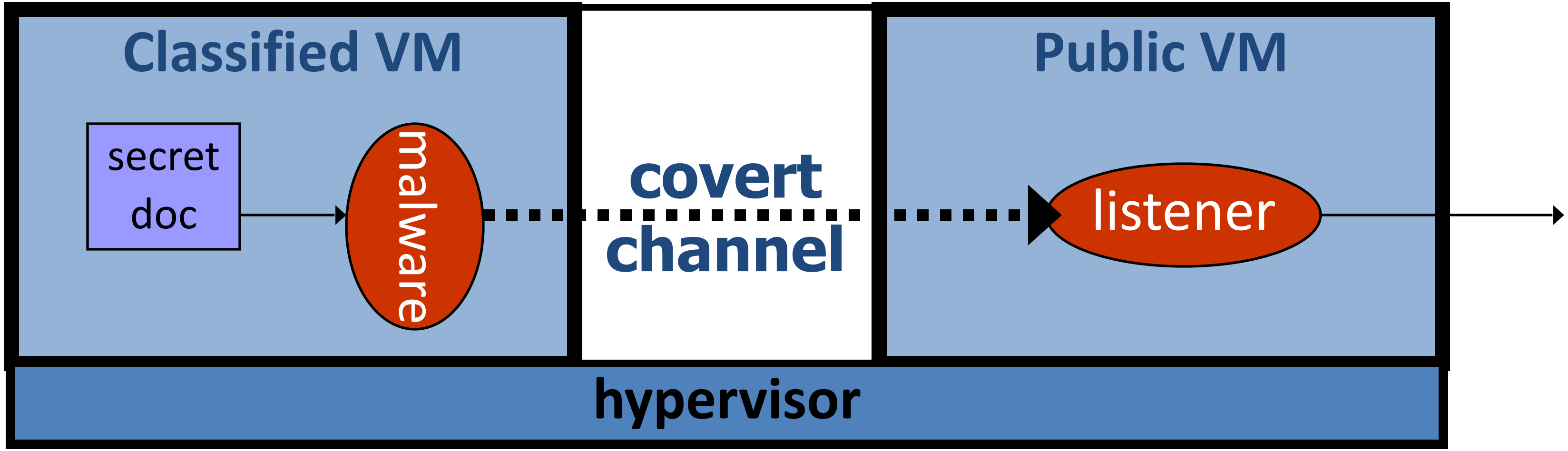
Requires that hypervisor protect itself and is not buggy

- (some) hypervisors are much simpler than a full OS

Problem: covert channels

Covert channel: unintended communication channel between isolated components

– Can 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

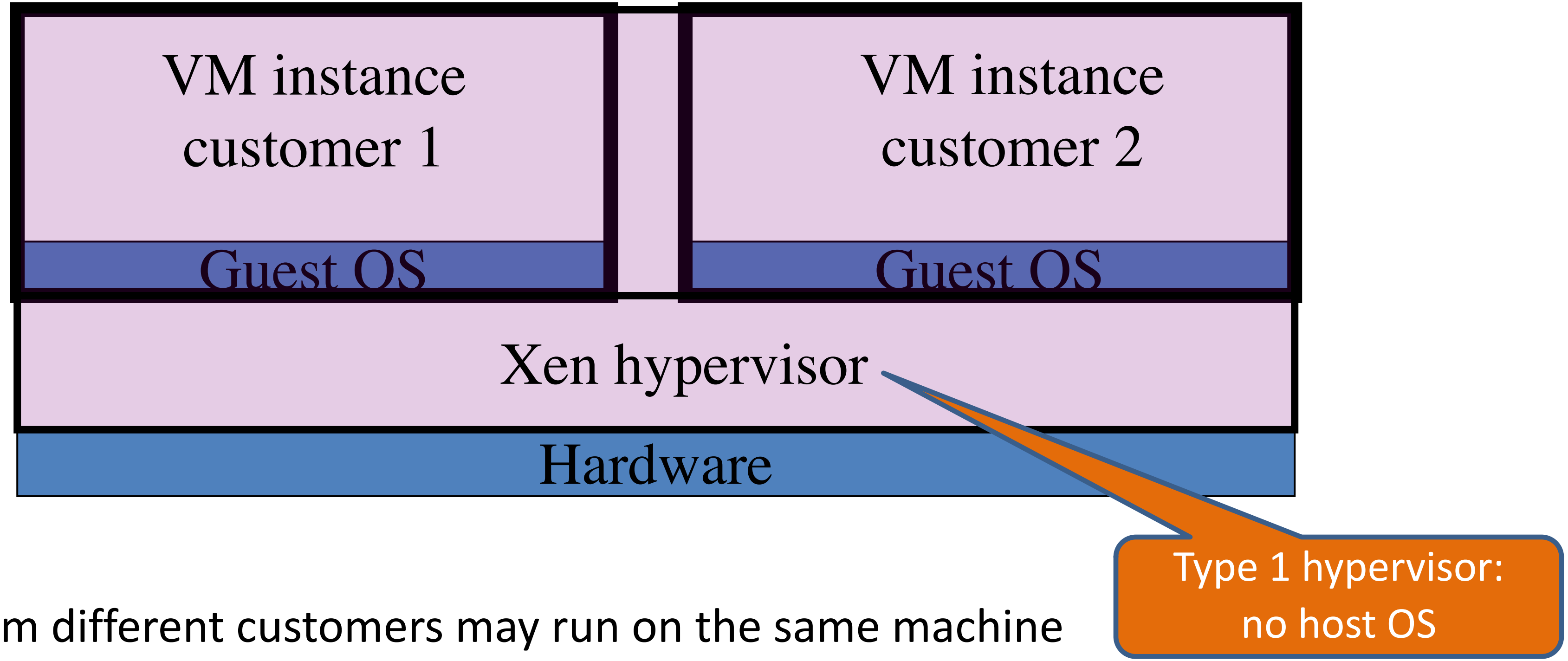
At 1:00am listener does CPU intensive calc. and measures completion time

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

Many covert channels exist in running system:

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

VM isolation in practice: cloud



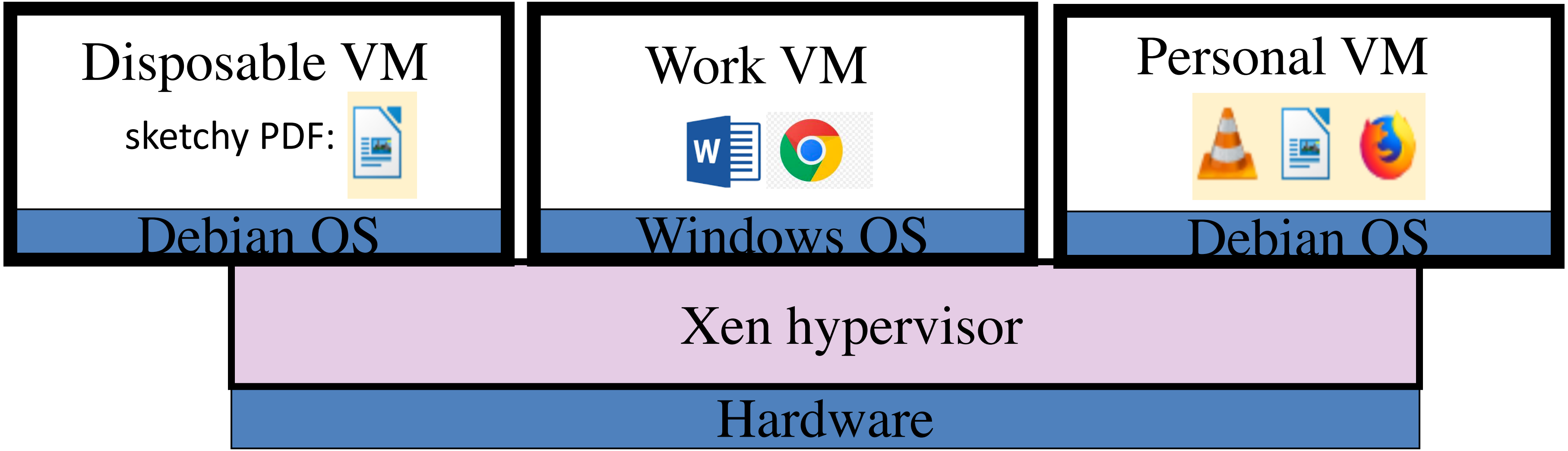
VMs from different customers may run on the same machine

- 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

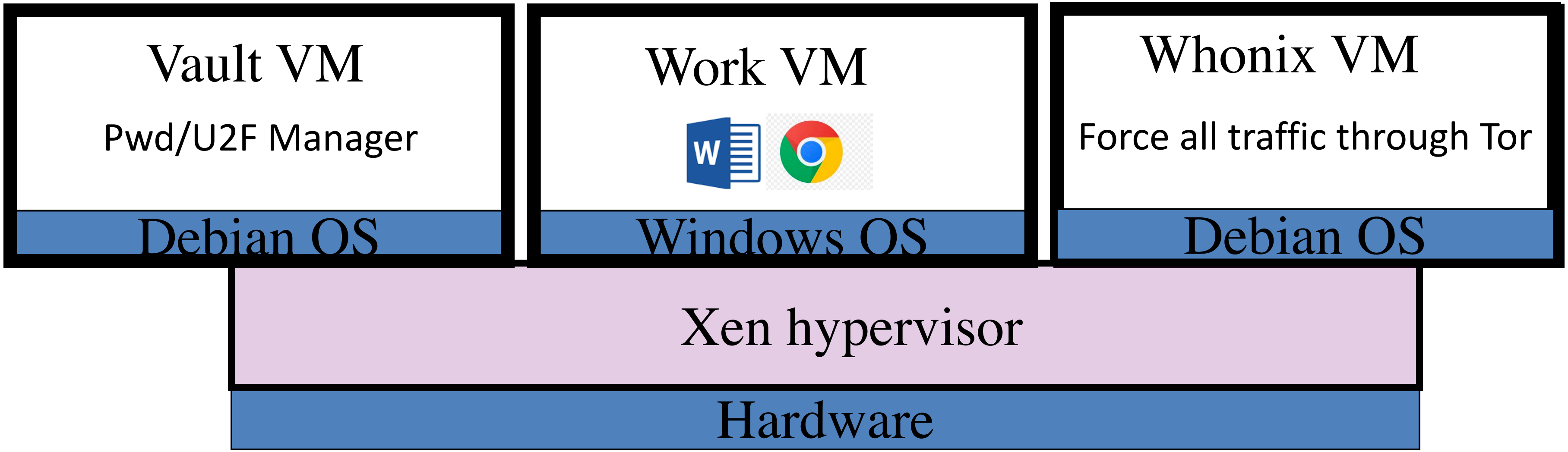
- Runs on top of the Xen hypervisor
- Access to peripherals (mic, camera, usb, …) controlled by VMs



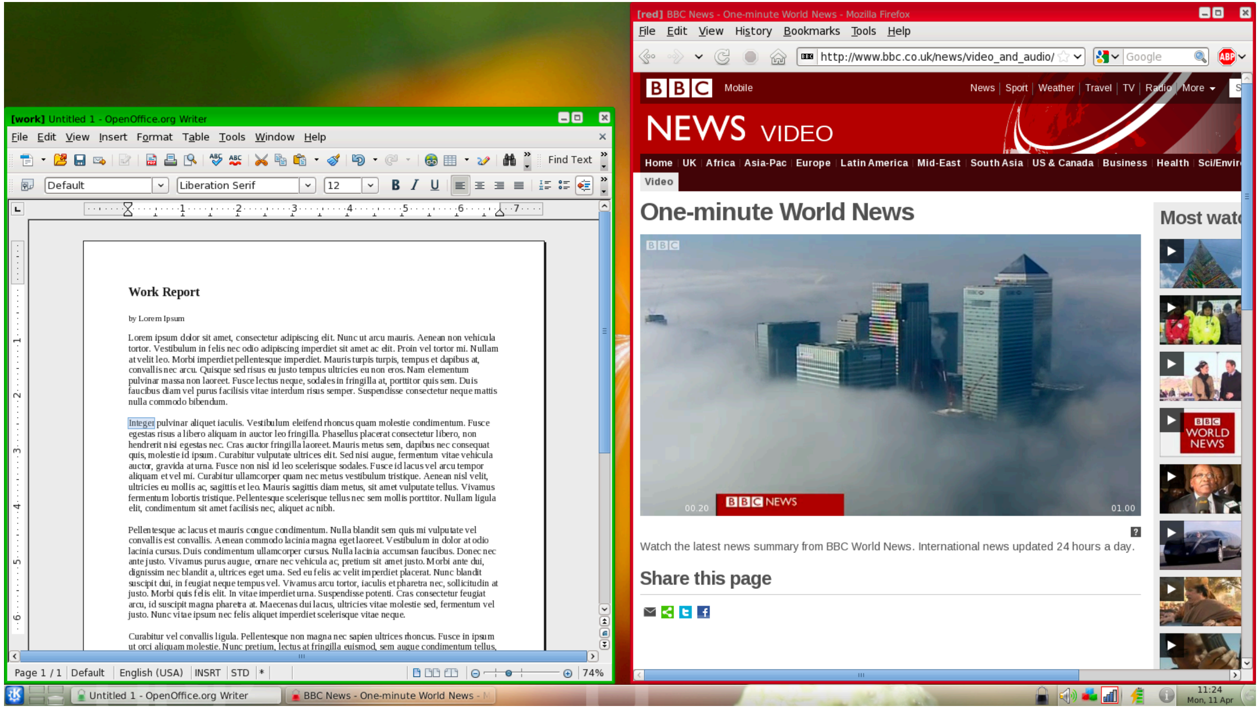
VM isolation in practice: end-user

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

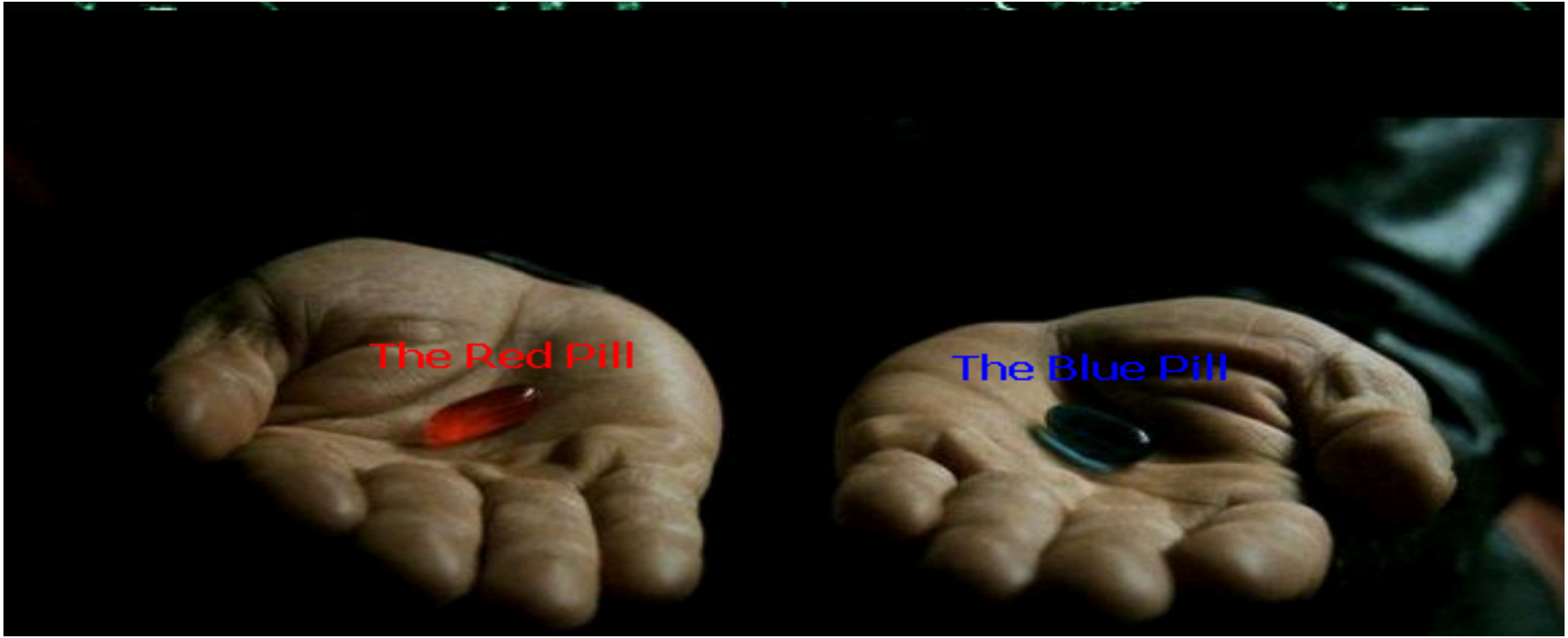
Hypervisor detection

Can an OS detect it is running on top of a hypervisor?

Applications:

- Malware can detect hypervisor
  - refuse to run to avoid reverse engineering
- Software that binds to hardware can refuse to run in VM
- DRM systems may refuse to run on top of hypervisor

Hypervisor detection



Hypervisor detection (red pill techniques)

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

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

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

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

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

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

Hypervisor detection

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

Hypervisors today focus on:

Compatibility: ensure off the shelf software works

Performance: minimize virtualization overhead

- VMMs do not provide **transparency**
  - **Anomalies reveal existence of hypervisor**

Isolation

Software Fault Isolation:

isolating threads

Software Fault Isolation [Whabe et al., 1993]

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

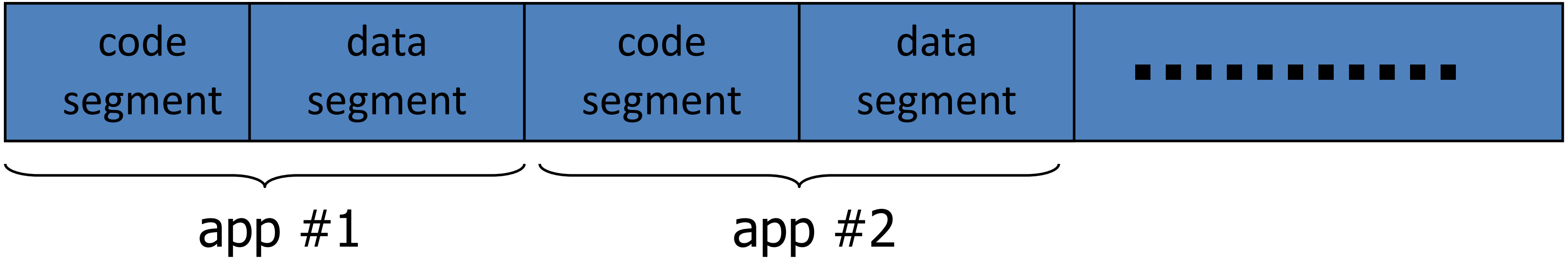
- Kernel module should not corrupt kernel
- Native libraries should not corrupt JVM

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

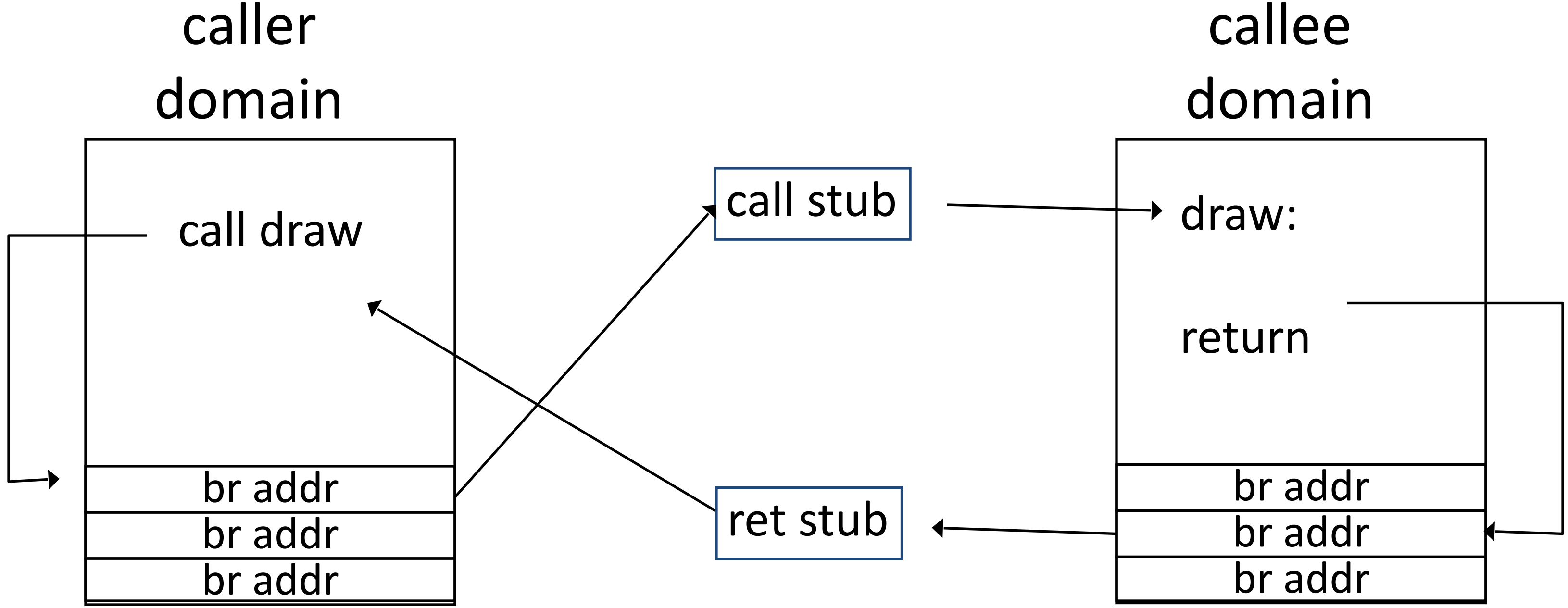
Problem: what if **jmp [addr]** jumps directly into indirect load?

(bypassing guard)

Solution:

This is why **jmp** instructions need a guard:
jmp guard ensures **[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

- 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

Isolation: summary

- Many sandboxing techniques:

    Physical air gap,   Virtual air gap (hypervisor),

    System call interposition (SCI), Software Fault isolation (SFI)

    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