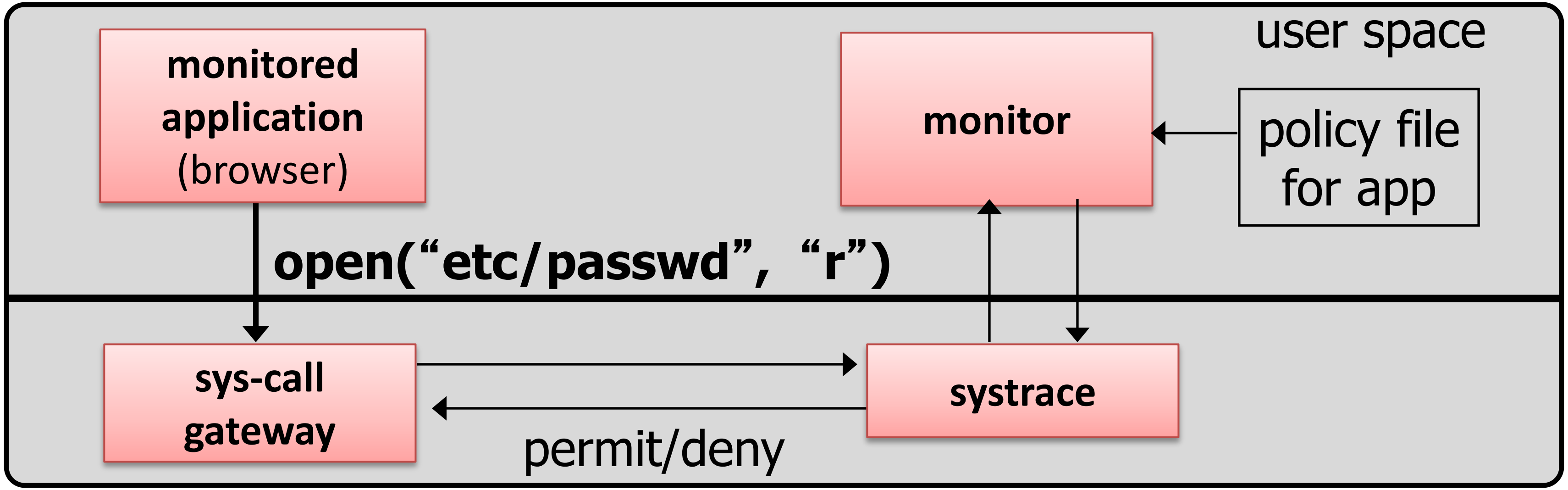
System Call Interposition

Slides credit: Dan Boneh

Administrative Issues

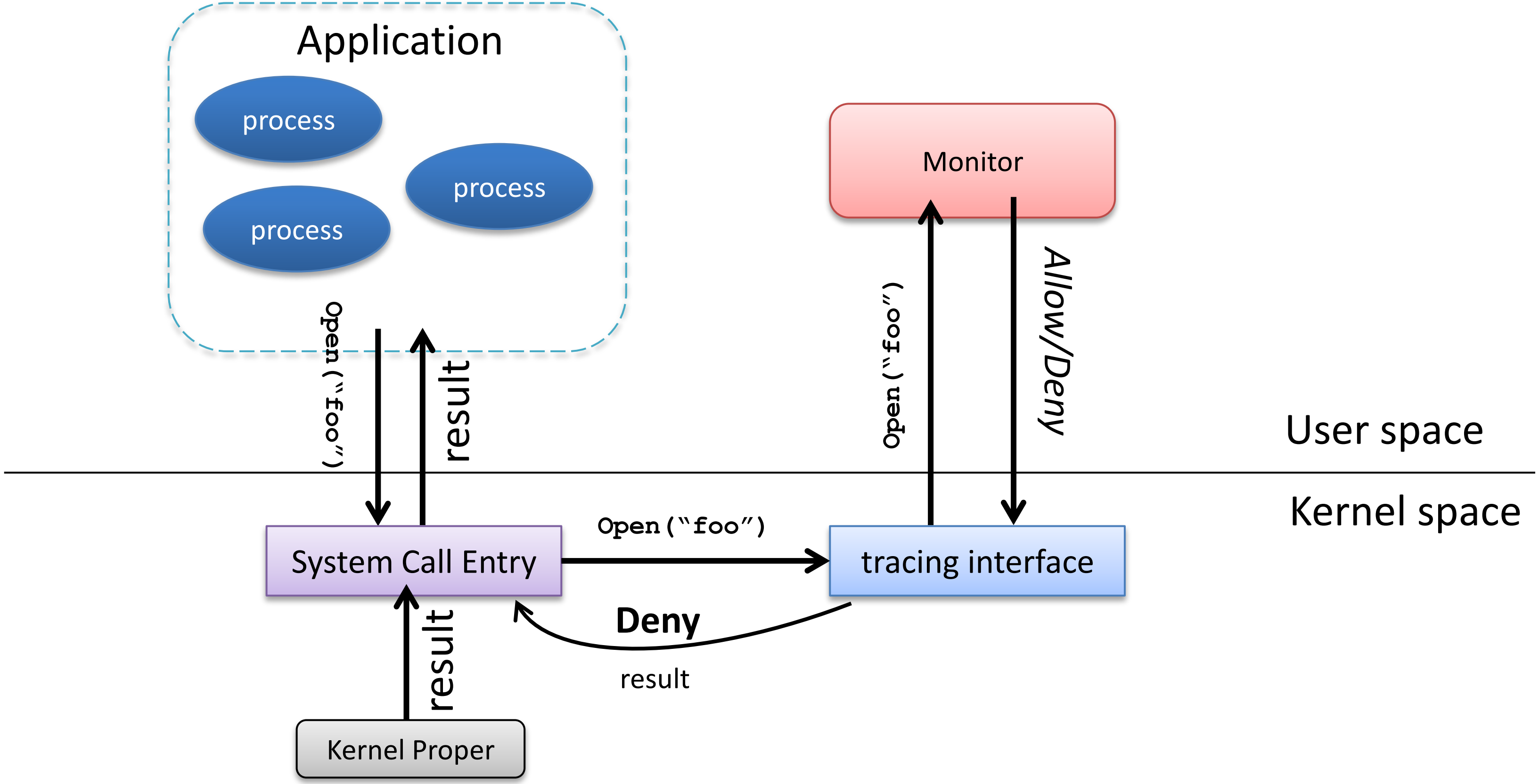
- Optional reading
- Practice questions for midterm
- Study guide for midterm
- Class survey

Alternate design: systrace [P'02]



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

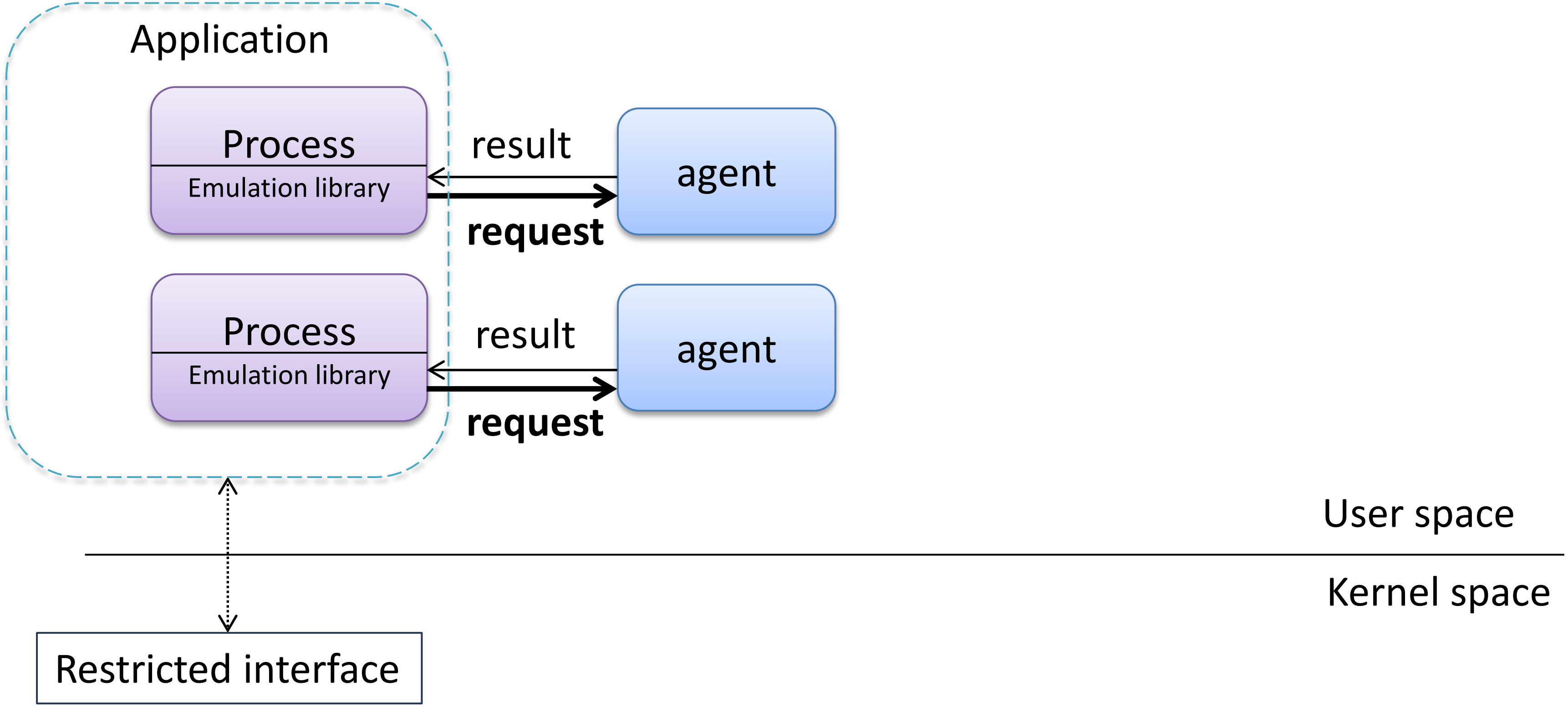
Filtering Architecture



Issues with Filtering Architecture

- Filter examines sys-calls and decides whether to block
- Difficulty with syncing state between app and monitor (CWD, UID, ..)
  - Incorrect syncing results in security vulnerabilities (e.g. disallowed file opened)

Ostia: a Delegation Architecture [GBR04]



Application

Process

Emulation library

result

request

agent

Process

Emulation library

result

request

agent

User space

Kernel space

Restricted interface

Ostia: a delegation architecture [GPR'04]

- 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

    (requires an emulation layer in monitored process)
- Incorrect state syncing will not result in policy violation
- What should agent do when app calls **execve?**
  - Process can make the call directly. Agent loads new policy file.

Policy

Sample policy file:

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

Manually specifying policy for an app can be difficult:

- Systrace can auto-generate policy by learning how app behaves on “good” inputs
- If policy does not cover a specific sys-call, ask user

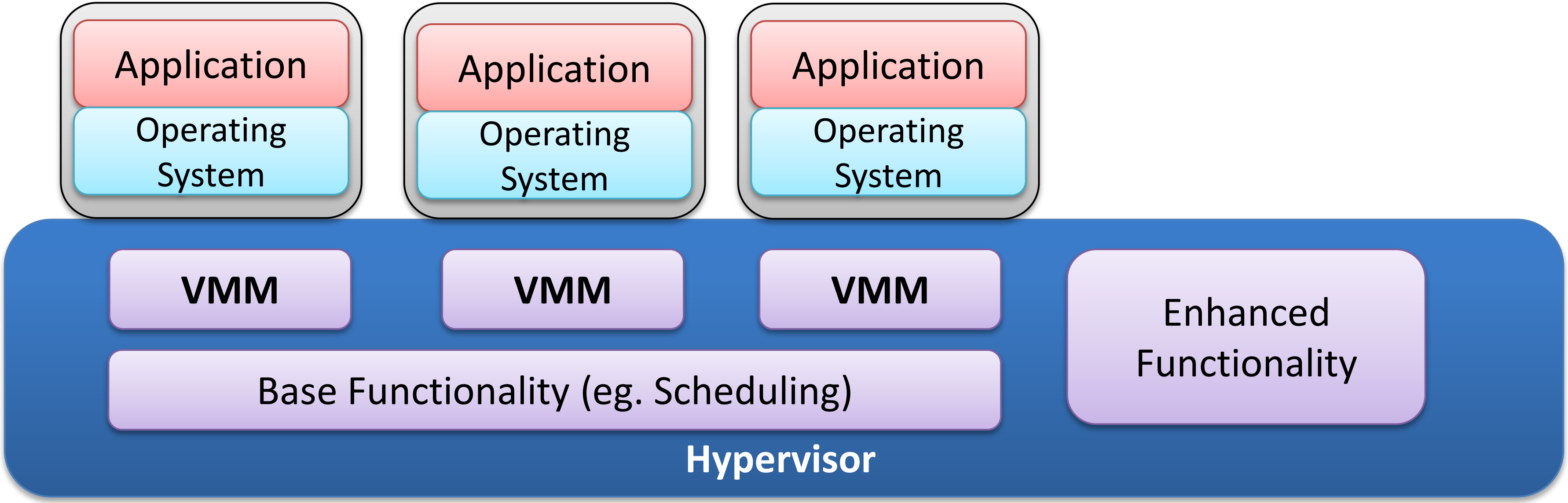
  … but user has no way to decide

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

Virtual Machine Monitor

Slides credit: Dan Boneh

Virtualization



Application

Operating System

Application

Operating System

Application

Operating System

VMM

VMM

VMM

Enhanced Functionality

Base Functionality (eg. Scheduling)

Hypervisor

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

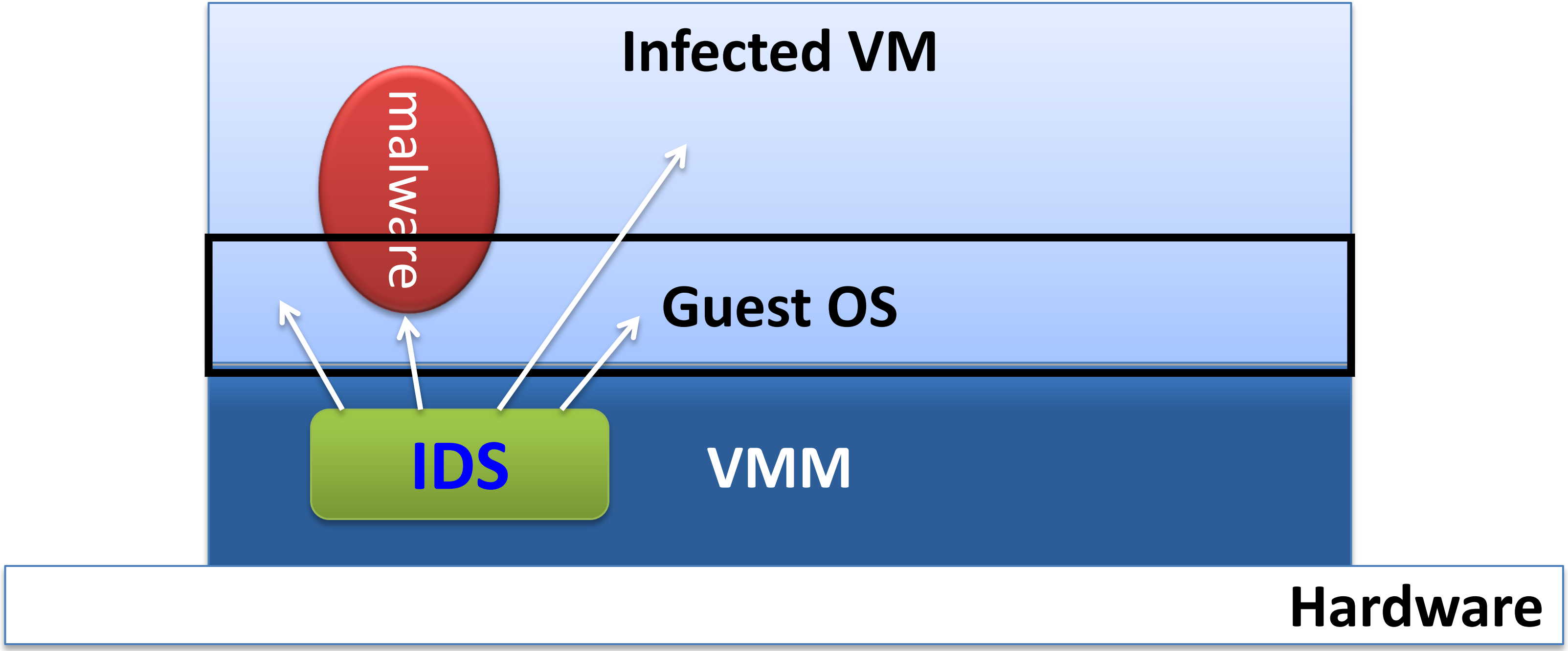
Standard solution: **run IDS system in the network**

- 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

VMM-based IDS



Sample checks

Stealth root-kit 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

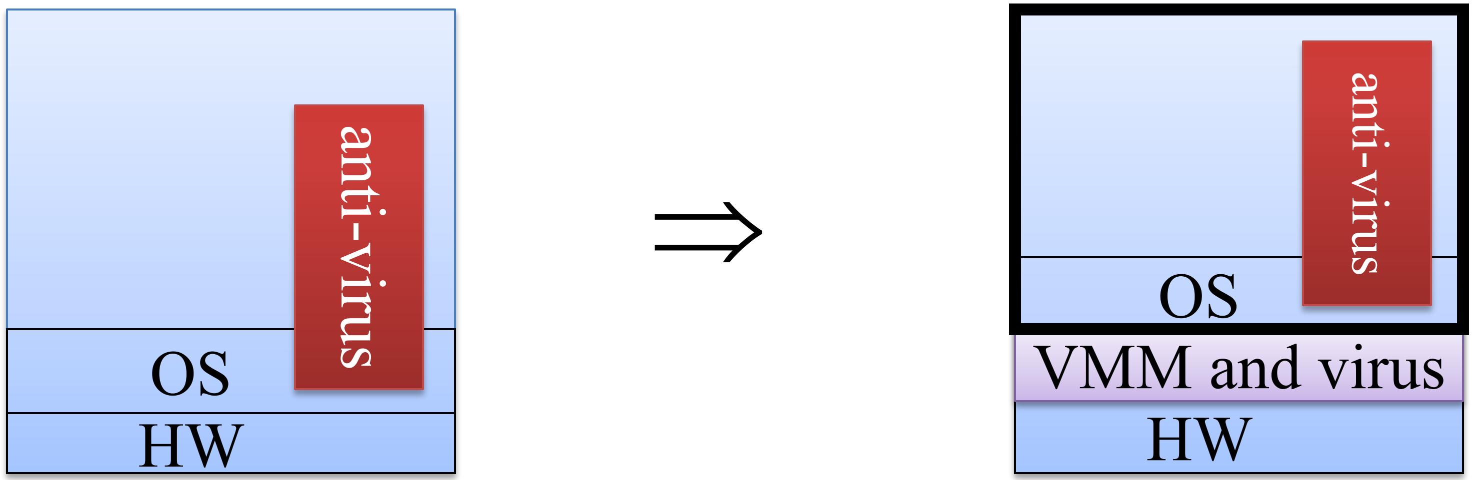
3. **Ensure GuestOS kernel integrity**
   - example: detect changes to sys_call_table

4. **Virus signature detector**
   - Run virus signature detector on GuestOS memory

VM-based Malware: 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

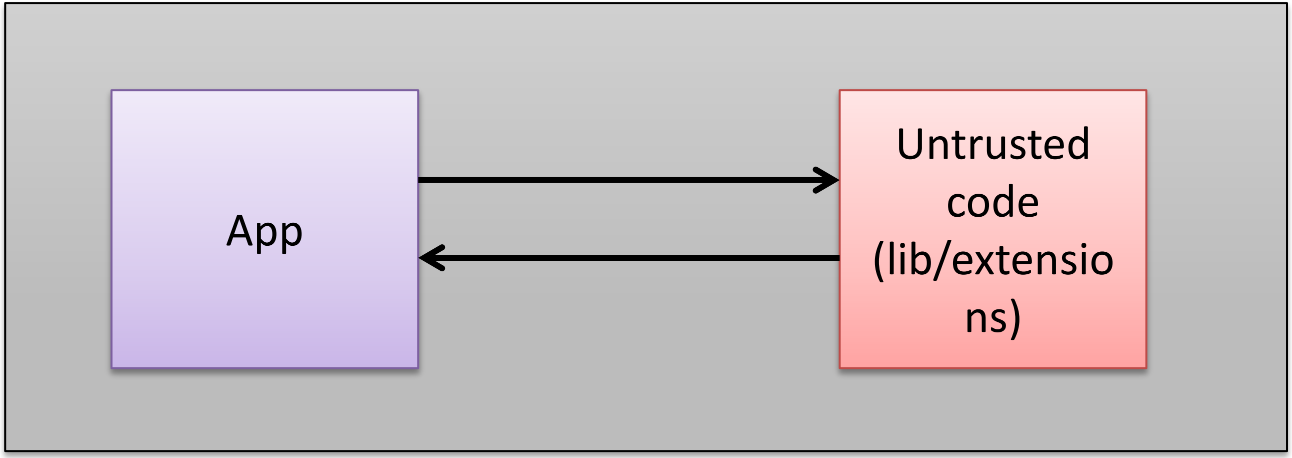


Software Fault Isolation

Slides credit: Dan Boneh, Stephen McCamant

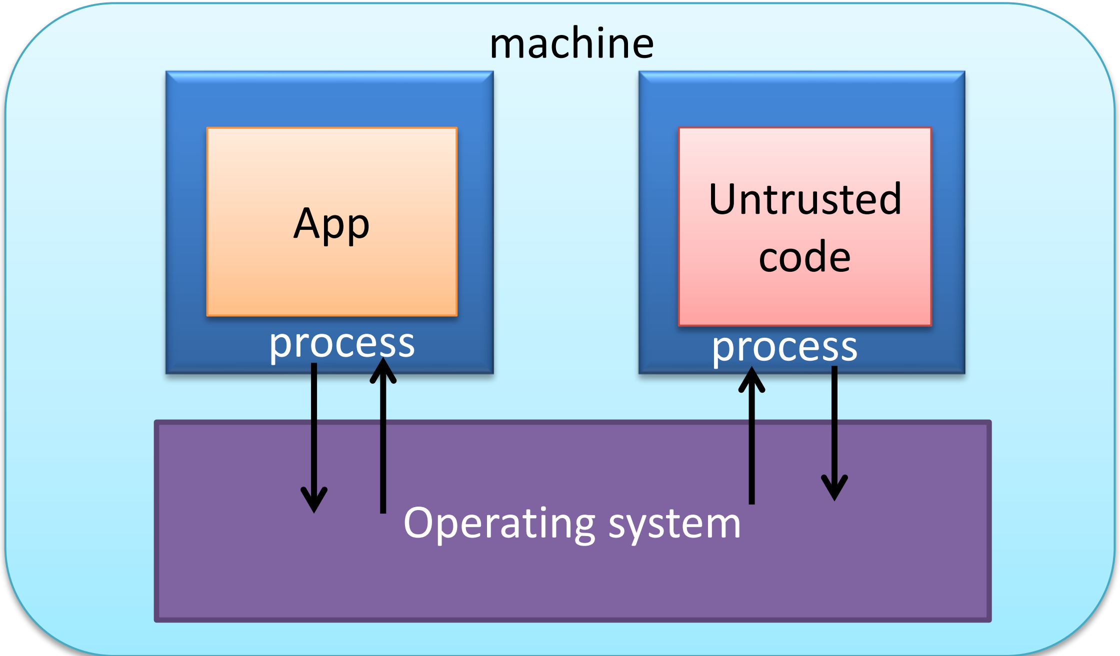
Goal

- Protect app from untrusted code it has to interact with
  - E.g., 3rd party libraries, modules, extensions, device drivers



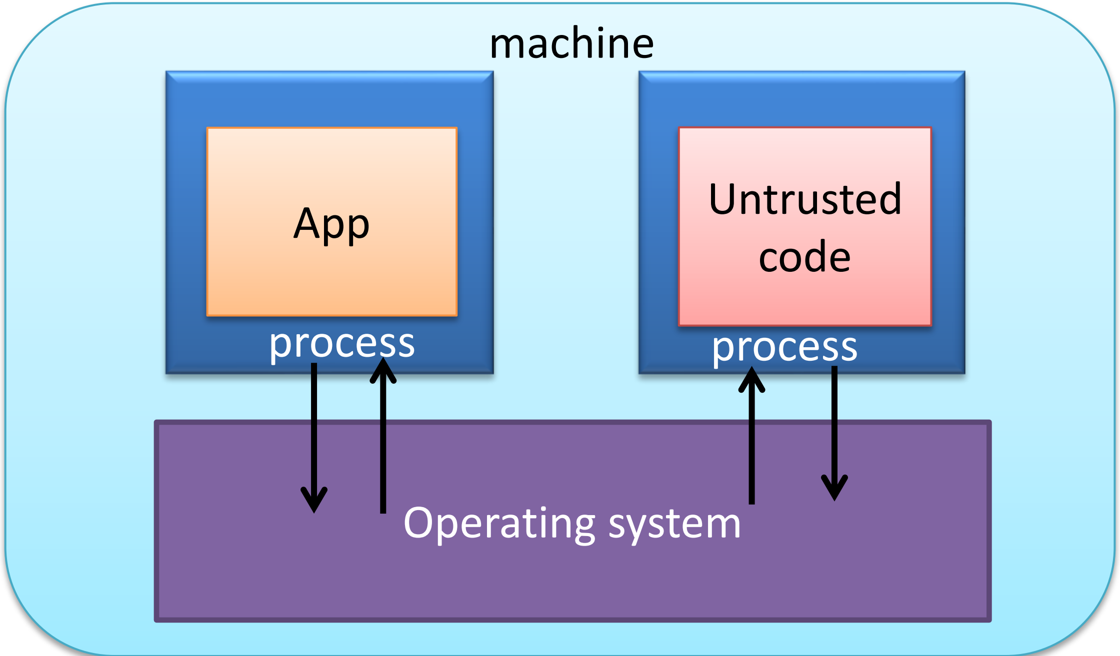
Solution I: Process Isolation

- Running in different processes
- Communicate with inter-process communication

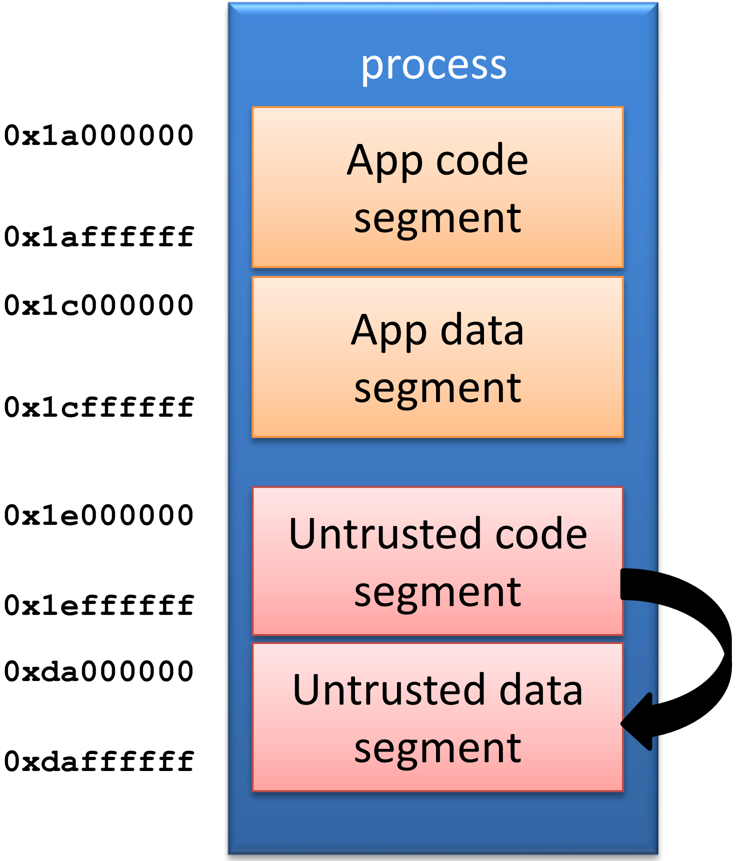


Issues with Process Isolation

- Inefficient for frequent IPC



Solution II: Software Fault Isolation



- App & untrusted code runs in same process
- Security enforcement: untrusted code can only read and write untrusted data segment
- [Wahbe et al. SOSP'93]

Untrusted code can read/write untrusted data

SFI: basic idea

```
f00:  nop
f04:  nop
f08:  nop
f0c:  nop
f10:  nop
f14:  sw $t3, 0($t4)
f18:  nop
```

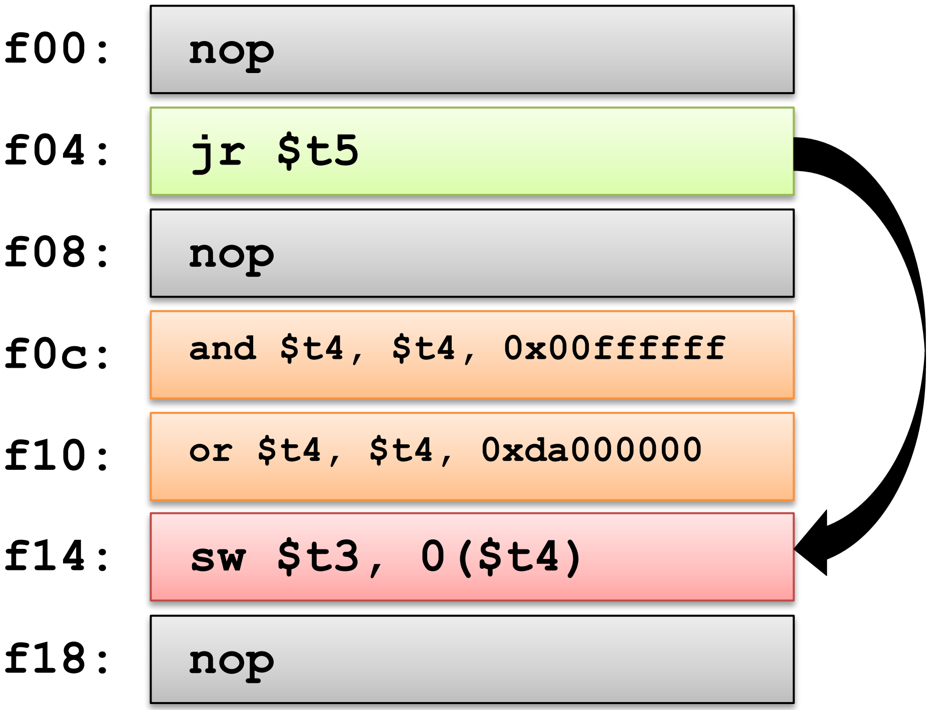
SFI: basic idea

```
f00:  nop
f04:  nop
f08:  nop
f0c:  nop
f10:  sandbox $t4
f14:  sw $t3, 0($t4)
f18:  nop
```

SFI: basic idea

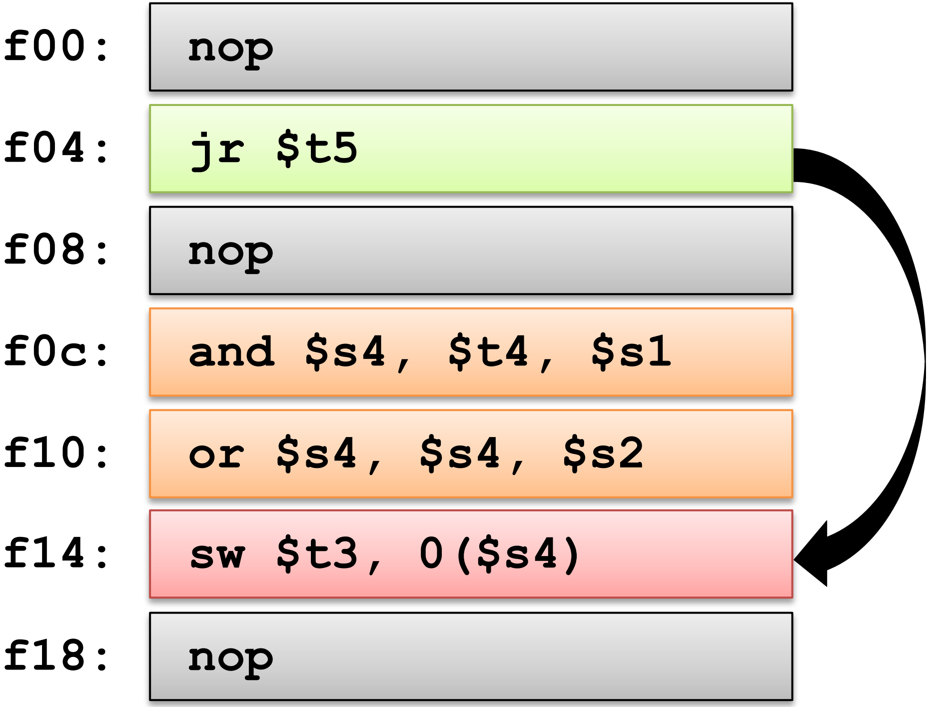
```
f00: nop
f04: nop
f08: nop
f0c: and $t4, $t4, 0x00ffffff
f10: or $t4, $t4, 0xda000000
f14: sw $t3, 0($t4)
f18: nop
```

SFI: basic idea



```
f00: nop
f04: jr $t5
f08: nop
f0c: and $t4, $t4, 0x00ffffff
f10: or $t4, $t4, 0xda000000
f14: sw $t3, 0($t4)
f18: nop
```

SFI: basic idea

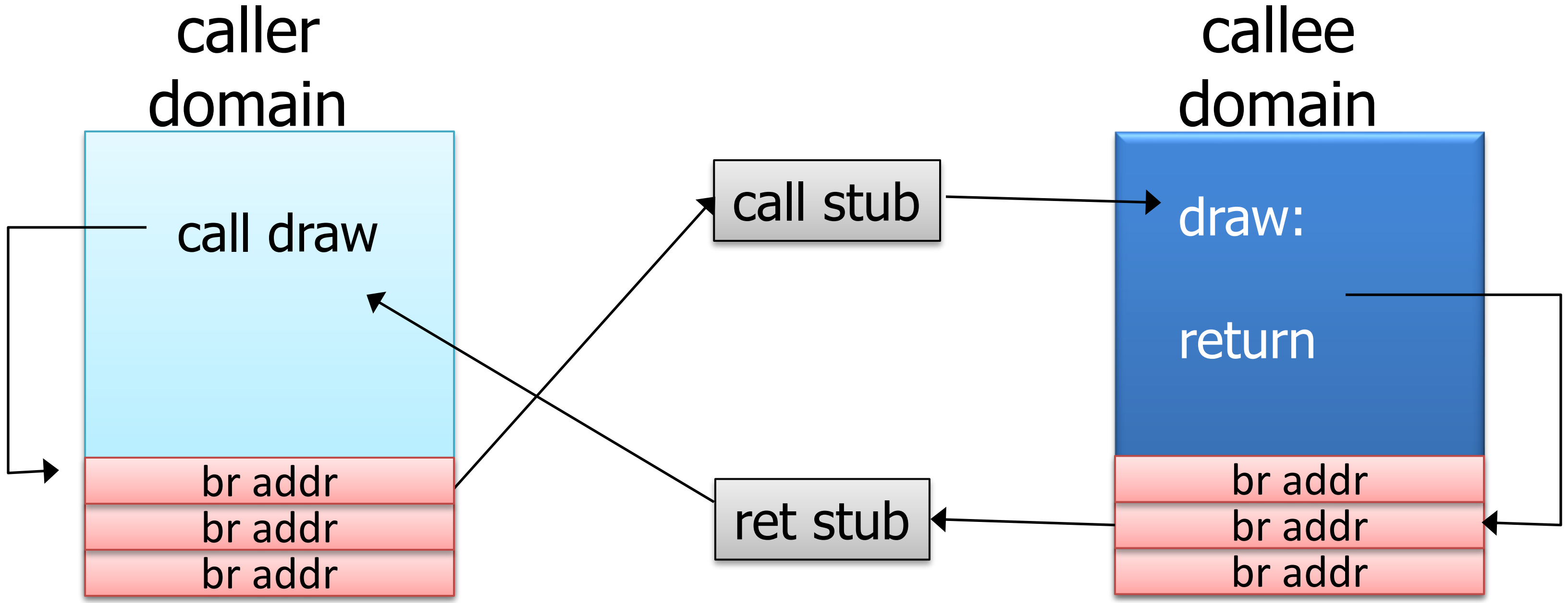


```
f00:  nop
f04:  jr $t5
f08:  nop
f0c:  and $s4, $t4, $s1
f10:  or $s4, $s4, $s2
f14:  sw $t3, 0($s4)
f18:  nop
```

Invariants:

```
$s1 = 0x00ffffff
$s2 = 0xda000000
$s4 = 0xda******
```

Cross domain calls

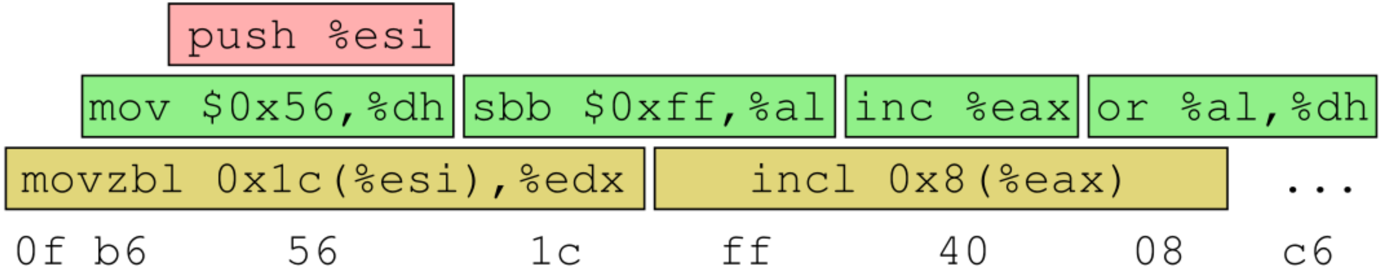


- Only stubs allowed to make cross-domain jumps
- Jump table contains allowed exit points
  - Addresses are hard coded, read-only segment

SFI and CISC

- The classic SFI approach only works for RISC-style aligned instructions
- Inapplicable to important CISC architectures like x86(-64)

CISC challenge: overlapping instructions



push %esi

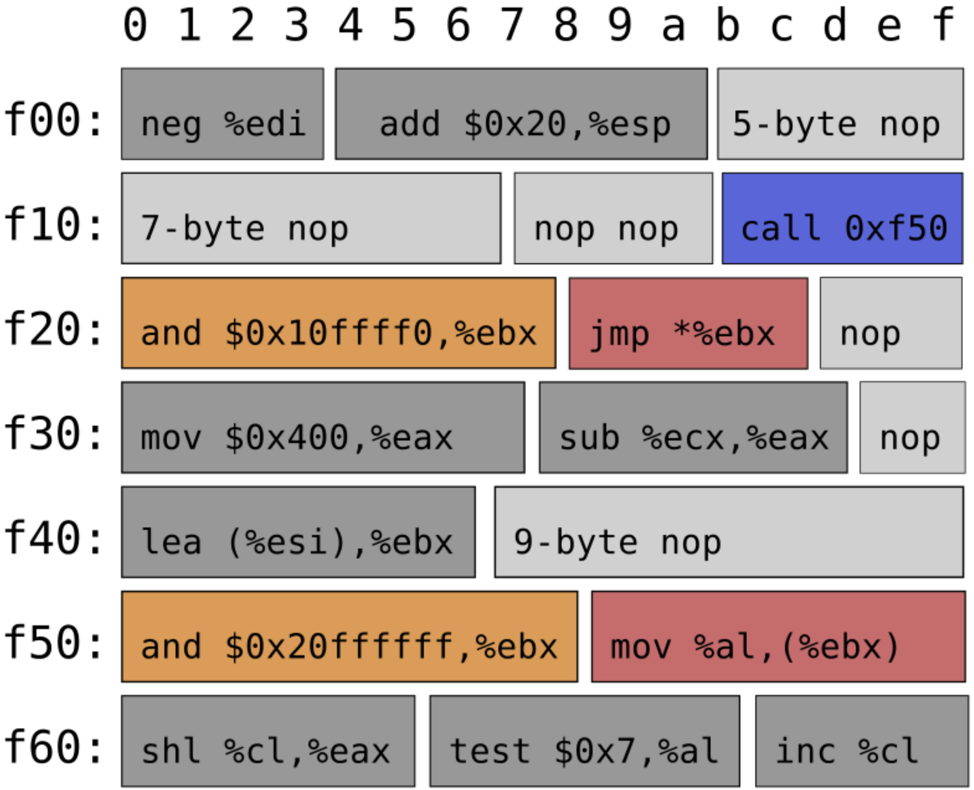
mov $0x56,%dh sbb $0xff,%al inc %eax or %al,%dh

movzbl 0x1c(%esi),%edx incl 0x8(%eax) ...

0f b6 56 1c ff 40 08 c6

- Processor can jump to any byte

CISC challenge: overlapping instructions



0 1 2 3 4 5 6 7 8 9 a b c d e f

f00: `neg %edi` `add $0x20,%esp` `5-byte nop`

f10: `7-byte nop` `nop nop` `call 0xf50`

f20: `and $0x10ffff0,%ebx` `jmp *%ebx` `nop`

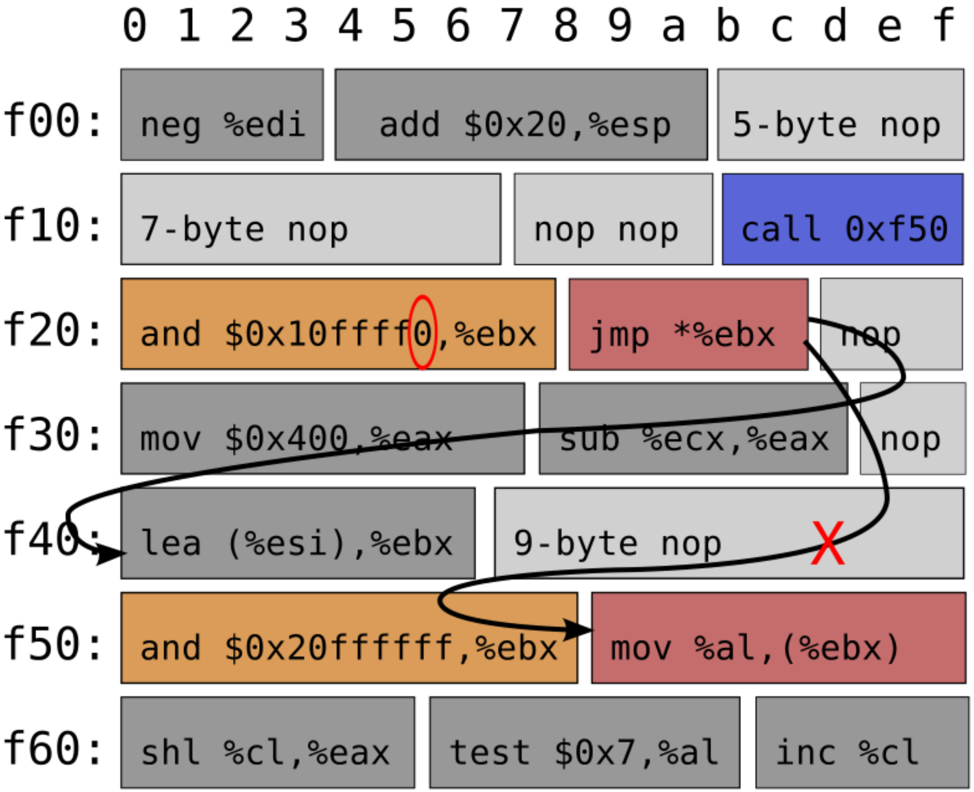
f30: `mov $0x400,%eax` `sub %ecx,%eax` `nop`

f40: `lea (%esi),%ebx` `9-byte nop`

f50: `and $0x20ffffff,%ebx` `mov %al,(%ebx)`

f60: `shl %cl,%eax` `test $0x7,%al` `inc %cl`

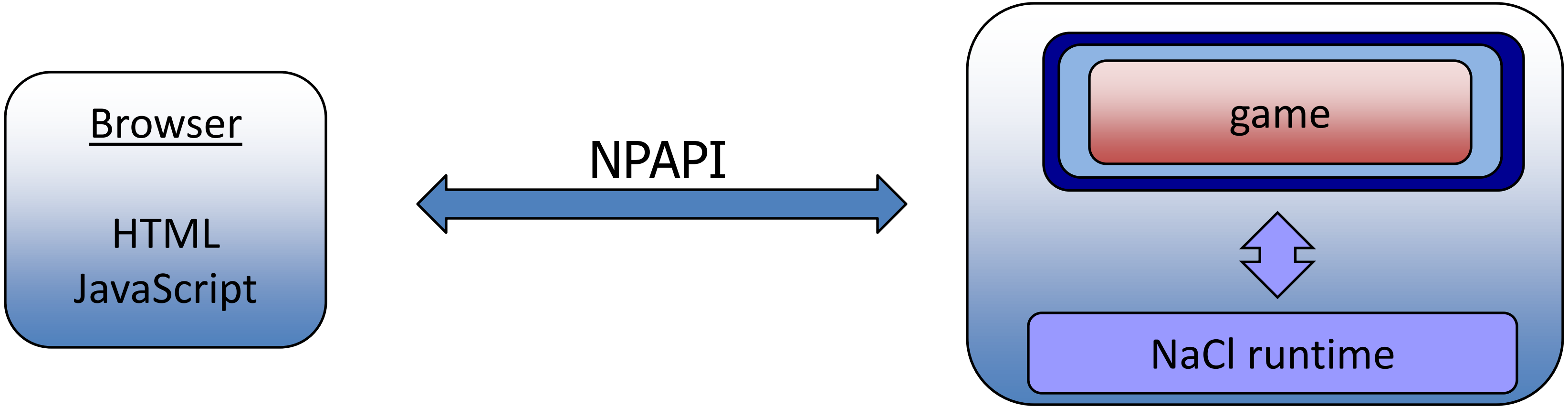
CISC challenge: overlapping instructions



More recently: Google Native Client

- Goal: make a web browser plugins as safe as JavaScript
  - But with the speed of machine code
- Uses SFI alignment approach
  - With variations for x86, ARM, x86-64
- Shipped in Google Chrome browser

NaCl: a modern day example



Browser

HTML

JavaScript

NPAPI

game

NaCl runtime

- game: untrusted x86 code
- Two sandboxes:
  - outer sandbox: restricts capabilities using system call interposition
  - Inner sandbox: uses x86 memory segmentation to isolate application memory among apps

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