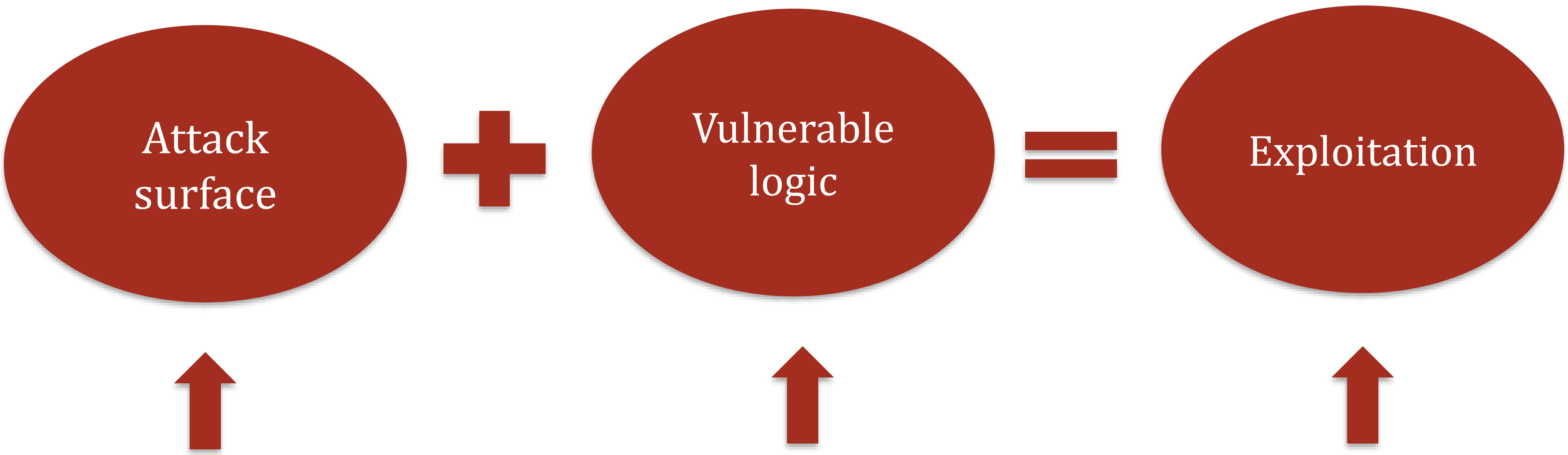
CS165 – Computer Security

Control Flow Integrity and Software Fault Isolation
Nov 16, 2021

Our story so far…

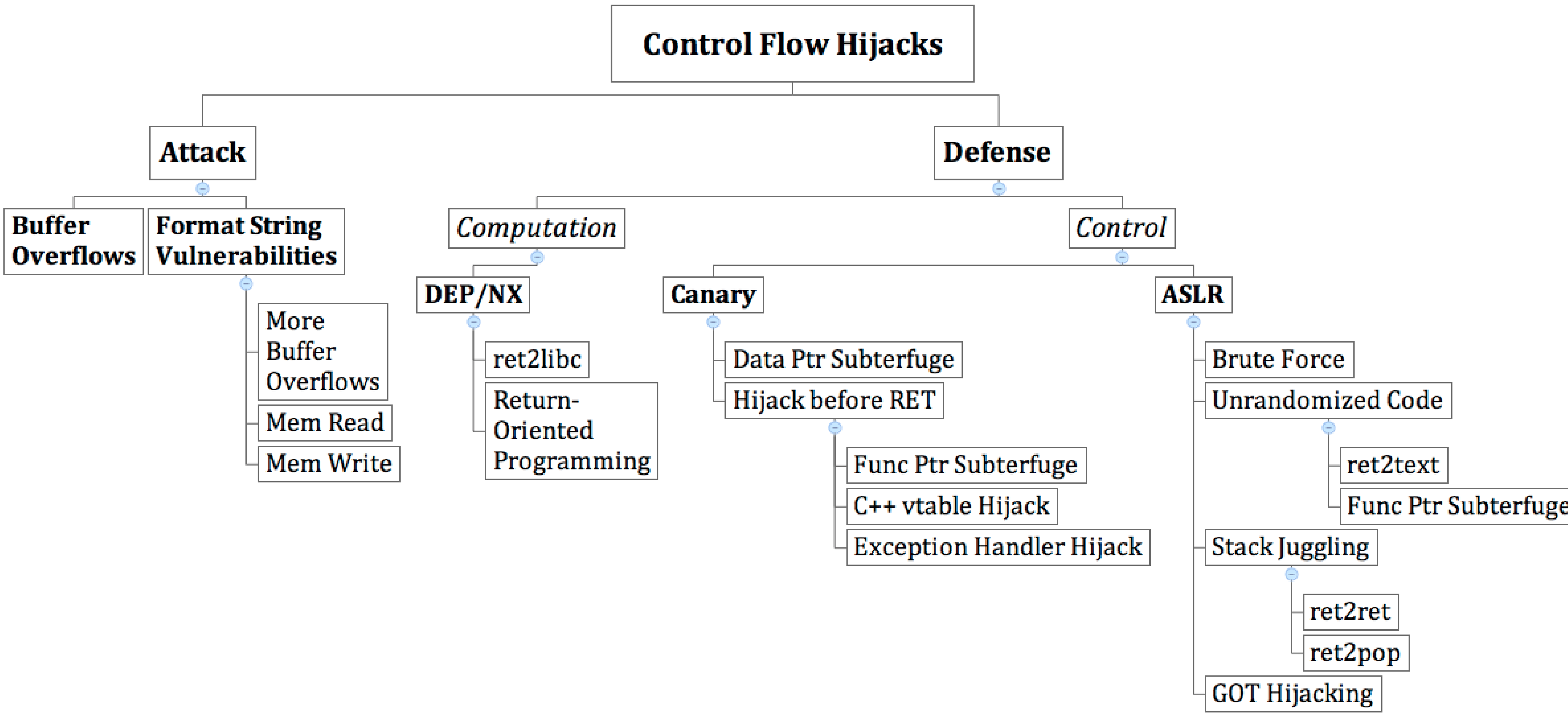


Access control

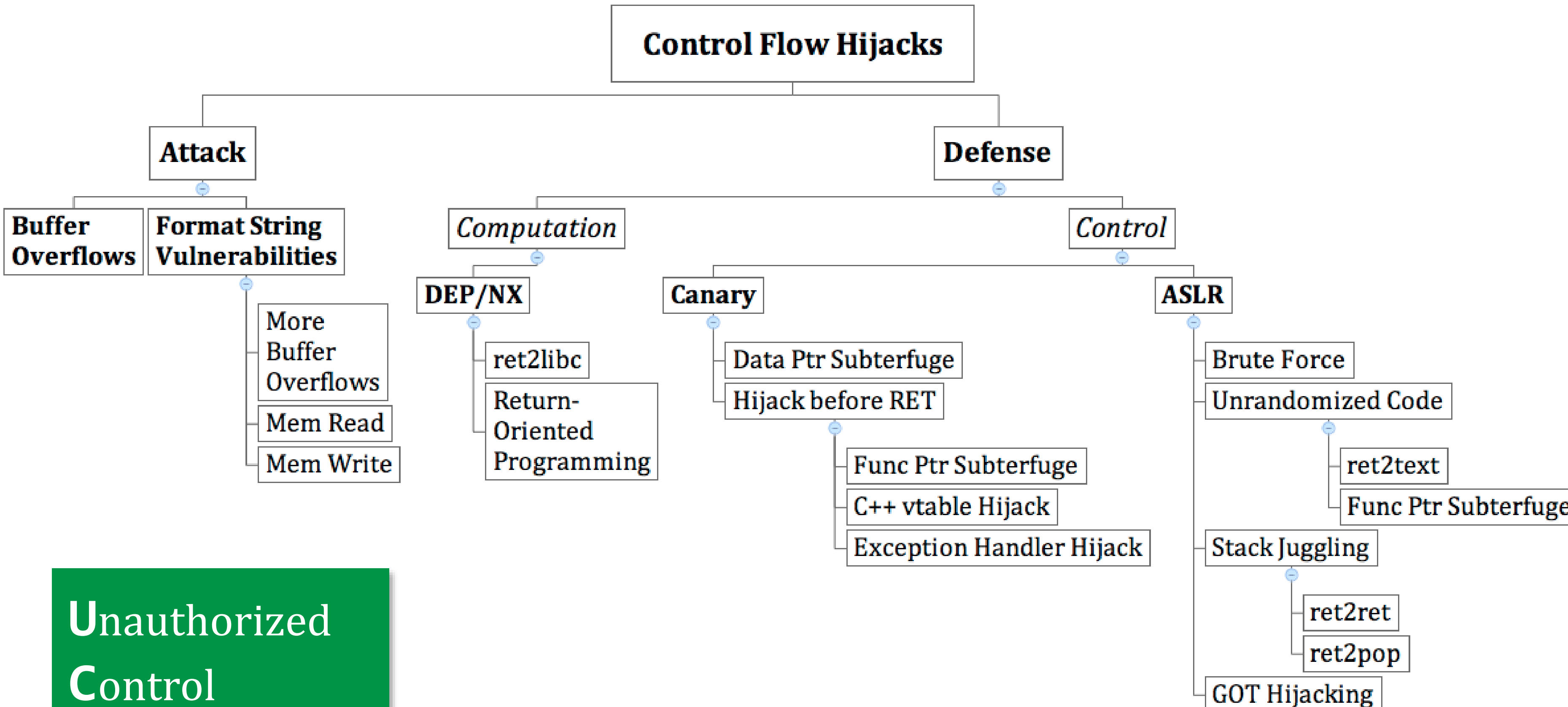
Bug finding via
(1) Static analysis
(2) Dynamic analysis

Prevention/mitigation techniques?

Our story so far…



Our story so far…



Unauthorized
Control
Information
Tampering

Control Flow Hijack: Always control + computation

shellcode (aka payload)	padding	&buf

computation + ***control***

Stack buffer overflow, return-to-libc
ROP, functional pointer subterfuge:
Hijacking the control

Can we prevent control manipulation?

Control Flow Integrity!

Agenda

Reference Monitors

Control Flow Integrity

Software Fault Isolation

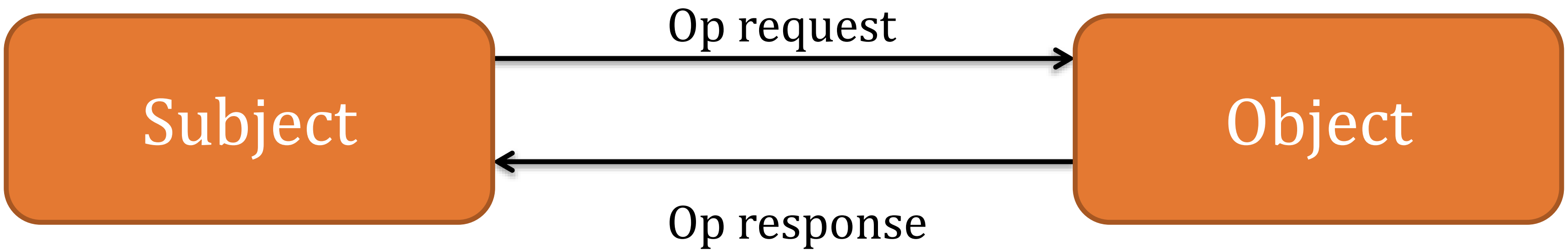
Agenda

Reference Monitors

NEXT

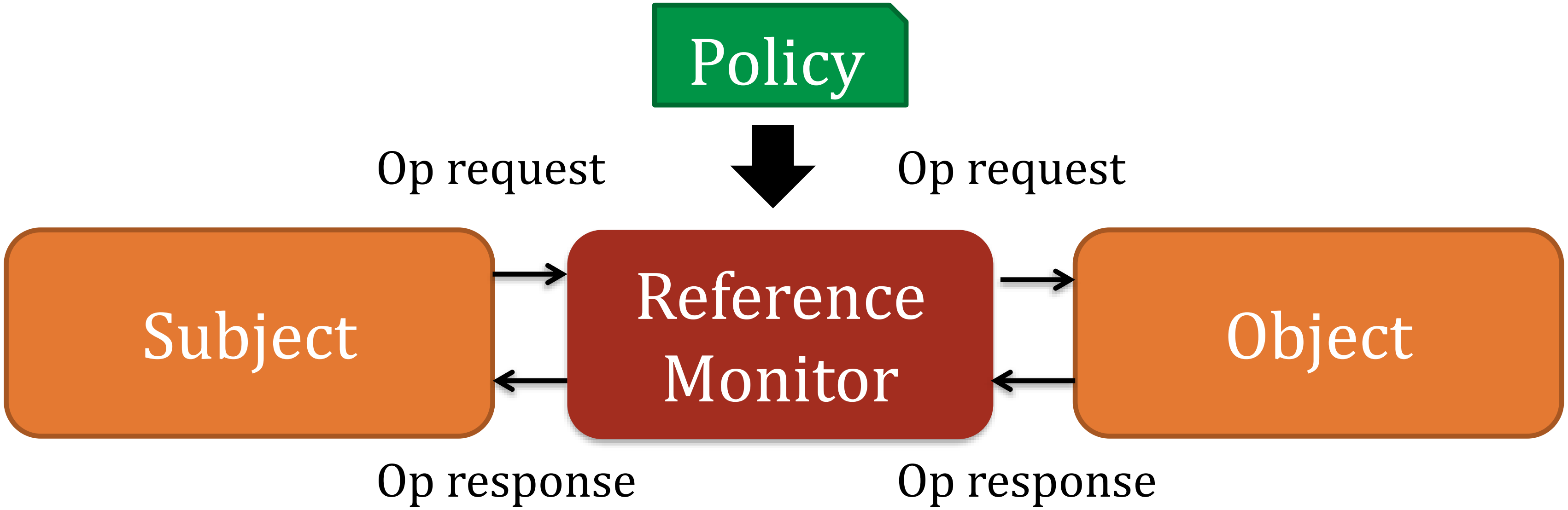
Control Flow Integrity

Software Fault Isolation



People
Processes
Computer Operations

Files
Sockets
Computer Operations



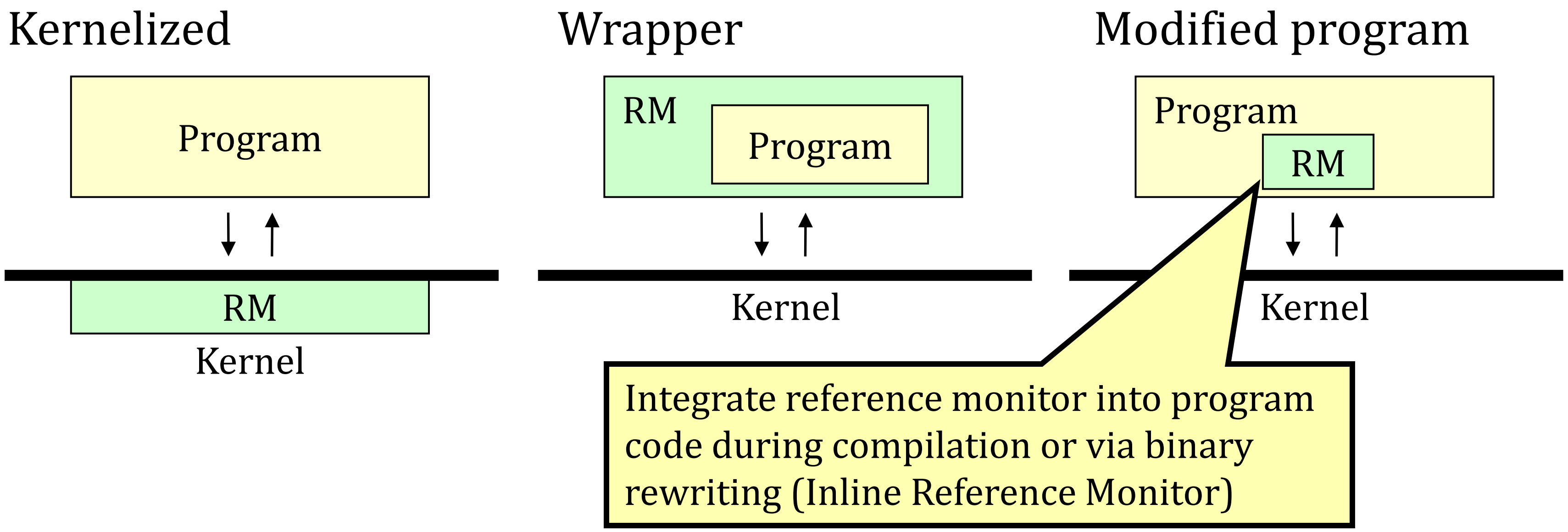
Principles:

1. Complete Mediation: The reference monitor must always be invoked
2. Tamper-proof: The reference monitor cannot be changed by unauthorized subjects or objects
3. Verifiable: The reference monitor is small enough to thoroughly understand, test, and ultimately, verify.

OS As a Reference Monitor

- OS enforces a variety of policies
  - File accesses are checked against file's Access Control List (ACL)
  - Process cannot write into memory of another process
  - Some operations require superuser privileges
    - But may need to switch back and forth (e.g., setuid in Unix)
  - Enforce CPU sharing, disk quotas, etc.

Reference Monitor Implementation

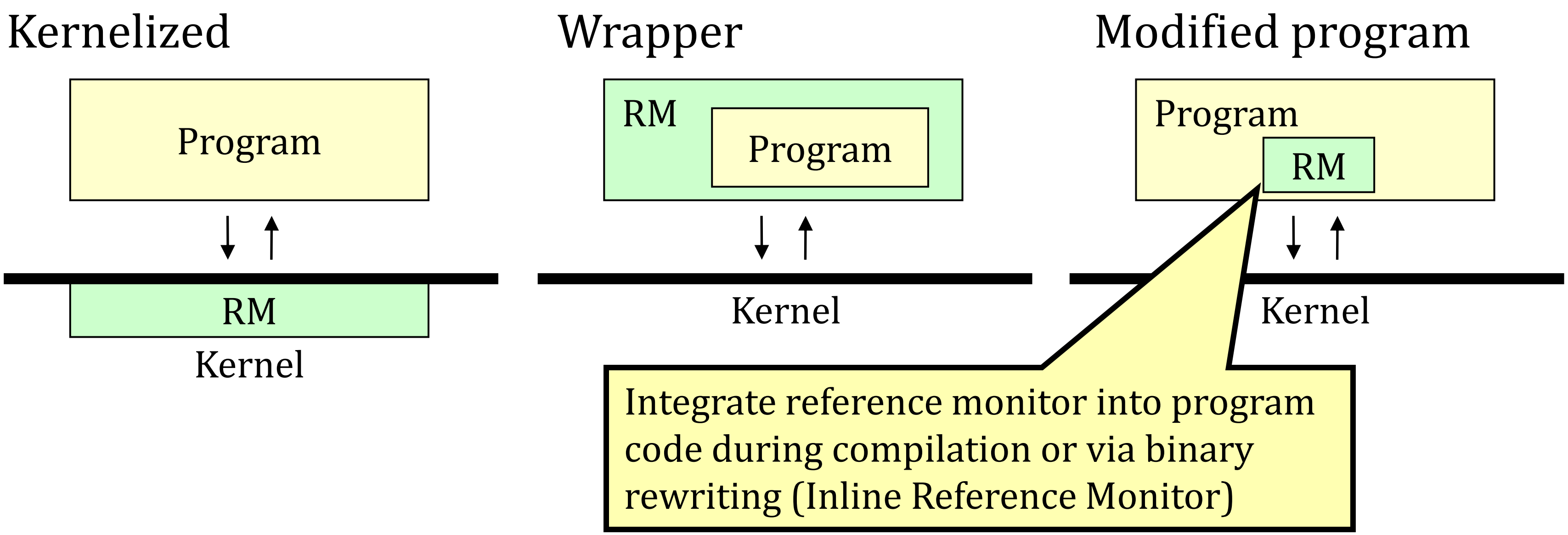


Today's Example: Inlining a control flow policy into a program

What Makes a Process itself Safe?

- Memory safety: all memory accesses are "correct"
  - Respect array bounds, separation of code and data
- Type safety: all function calls and operations have arguments of correct type
- Control-flow safety: all control transfers are envisioned by the original program
  - No arbitrary jumps, no calls to library routines that the original program did not call

Reference Monitor Implementation



- Policies can depend on application semantics
- Enforcement doesn't require context switches in the kernel
- Lower performance overhead

Agenda

Reference Monitors

Control Flow Integrity

Software Fault Isolation

Agenda

Reference Monitors

Control Flow Integrity NEXT

Software Fault Isolation

Control Flow Integrity

Assigned Reading:

Control-Flow Integrity: Principles, Implementation and Applications
by Abadi, Budiu, Erlingsson, and Ligatti

Control Flow Integrity

Control Flow Integrity

- **protects against powerful adversary**
  - with <u>full</u> control over <u>entire</u> data memory

Control Flow Integrity

- **protects against powerful adversary**
  – with full control over entire data memory
- **widely-applicable**
  – language-neutral; requires binary only

Control Flow Integrity

- **protects against powerful adversary**
  - with full control over entire data memory
- **widely-applicable**
  - language-neutral; requires binary only
- **provably-correct & trustworthy**
  - formal semantics; small verifier

Control Flow Integrity

- **protects against powerful adversary**
  - with full control over entire data memory
- **widely-applicable**
  - language-neutral; requires binary only
- **provably-correct & trustworthy**
  - formal semantics; small verifier
- **efficient**
  - hmm… 0-45% in experiments; average 16%

Control Flow Integrity

- **protects against powerful adversary**
  - with full control over entire data memory
- **widely-applicable**
  - language-neutral; requires binary only
- **provably-correct & trustworthy**
  - formal semantics; small verifier
- **efficient**
  - hmm… 0-45% in experiments; average 16%

CFI Adversary Model

CAN

- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

CFI Adversary Model

CAN

- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

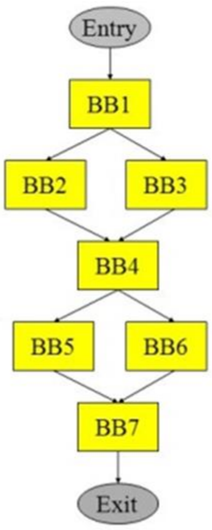
CANNOT

- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs

CFI Overview

Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

“static”



Most control flow transfer targets are hard-coded

jnz	**short loc_18002C19E**	**0F 85 B4 00 00 00**
call	**__scrt_initialize_crt**	**EB 12 0B 00 00**

But what about indirect jumps and ret?

e.g., func pointer, ret address

CFI Overview

Method to check indirect control transfers:

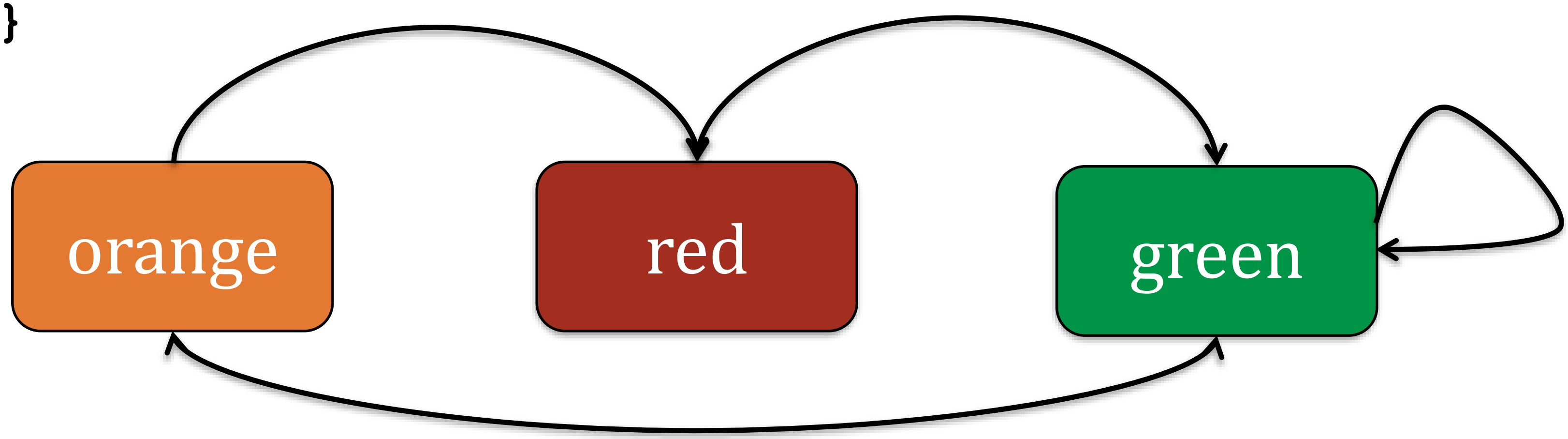
- build CG and CFG statically, e.g., at compile time
  - **call, jmp, ret** instructions
- instrument (rewrite) binary, e.g., at install time
  - add IDs and ID checks; maintain ID uniqueness
- verify CFI instrumentation at load time
  - indirect jump targets, presence of IDs and ID checks, ID uniqueness
- perform ID checks at run time
  - indirect jumps have matching IDs

Call Graph – Checking **Return Address**

```
void orange()
{
1.  red(1);
2.  red(2);
3.  green();
}
```

```
void red(int x)
{
green();
...
}
```

```
void green()
{
   green();
   orange();
}
```



- Upon return (e.g., red)

ret →

```
mov ecx, [esp]  //check ret addr
cmp [ecx], AABBCCDDh
jne error_label
jmp ecx+4
```

- At the caller (e.g., orange)

call
...
→

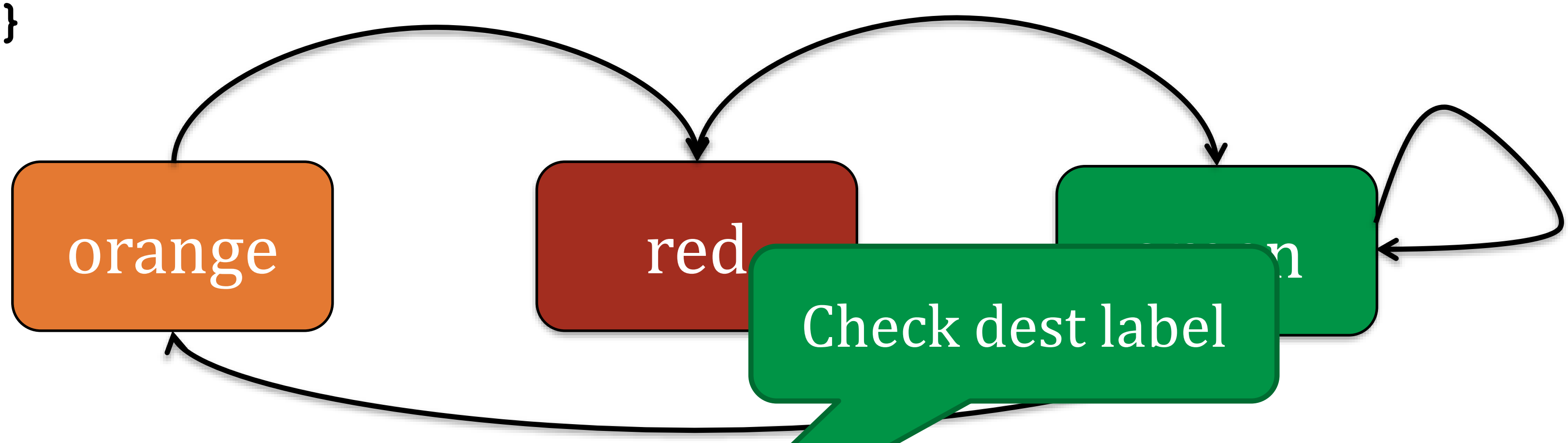
```
0x00   call red
0x04   AABBCCDDh #read-only data
0x08   ...
```

Call Graph – Checking **Return Address**

```
void orange()
{
1.  red(1);
2.  red(2);
3.  green();
}
```

```
void red(int x)
{
green();
...
}
```

```
void green()
{
  green();
  orange();
}
```



- Upon return (e.g., red)

ret →

```
mov ecx, [esp]  //check ret addr
cmp [ecx], AABBCCDDh
jne error_label
jmp ecx+4
```

- At the caller (e.g., orange)

call
…
→

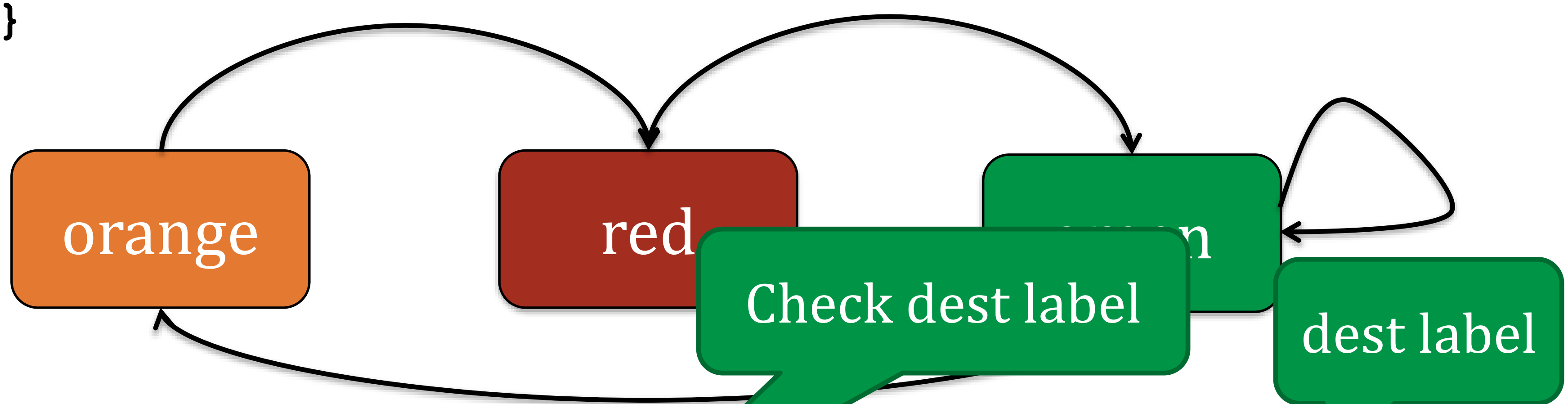
```
0x00   call red
0x04   AABBCCDDh #read-only data
0x08   ...
```

Call Graph – Checking **Return Address**

```
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
green();
...
}
```

```
void green()
{
  green();
  orange();
}
```



- Upon return

ret →

```
mov ecx, [esp]  //check ret addr
cmp [ecx], AABBCCDDh
jne error_label
jmp ecx+4
```

- At the caller (e.g., orange)

call
...
→

```
0x00   call red
0x04   AABBCCDDh #read-only data
0x08   ...
```

Checking **Function Pointer** Deference

```
FF 53 08                   call   [ebx+8]            ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 43 08                   mov   eax, [ebx+8]        ; load pointer into register
3E 81 78 04 78 56 34 12    cmp   [eax+4], 12345678h  ; compare opcodes at destination
75 13                      jne   error_label         ; if not ID value, then fail
FF D0                      call eax                  ; call function pointer
3E 0F 18 05 DD CC BB AA    prefetchnta [AABBCCDDh]   ; label ID, used upon the return
```

Checking **Function Pointer** Deference

```
FF 53 08                   call   [ebx+8]           ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 43 08                   mov  eax, [ebx+8]        ; load pointer into register
3E 81 78 04 78 56 34 12    cmp  [eax+4], 12345678h  ; compare opcodes at destination
75 13                      jne  error_label         ; if not ID value, then fail
FF D0                      call eax                 ; call function pointer
3E 0F 18 05 DD CC BB AA    prefetchnta [AABBCCDDh]  ; label ID, used upon the return
```

Check dest label

Performance

Size: increase 8% avg

Time: increase 0-45%; 16% avg

– I/O latency helps hide overhead

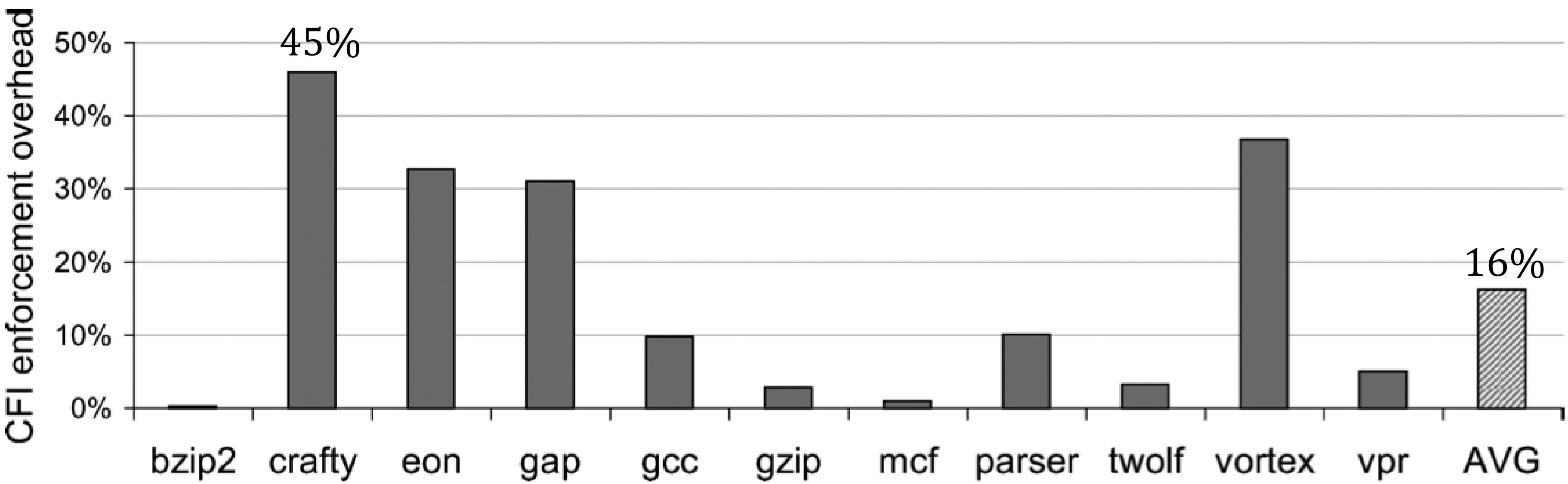
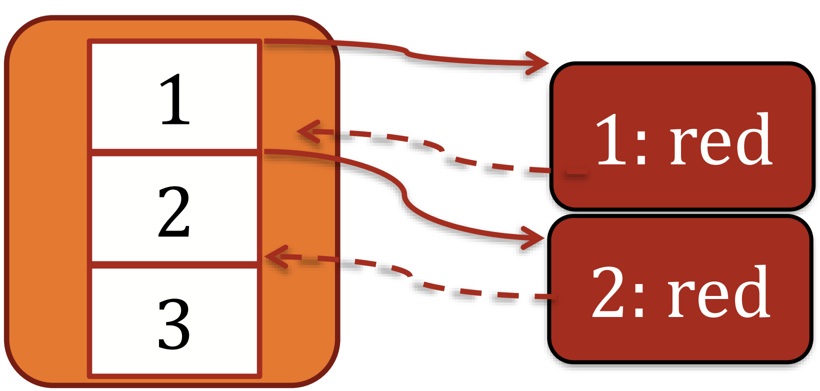


Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Context-Sensitive CFI

- Previous assumption: destination is fixed (a single target or a group)

```
void orange()      void red(int x)      void green()
{                  {                    {
1. red(1);         ..                     green();
2. red(2);         }                      orange();
3. green();                             }
}
```



A *more precise* CFI for orange lines 1 and 2.

Context-Sensitive CFI

Whether different calling contexts are distinguished

```
void orange()
{
1.  red(1);
2.  red(2);
3.  green();
}
```

```
void red(int x)
{
..
}
```

```
void green()
{
  green();
  yellow();
}
```

Context sensitive distinguishes 2 different calls to red(-)

Context Sensitive Example

```
a = id(4);
                void id(int z)
                { return z; }
b = id(5);
```

Context-Sensitive
(color denotes matching call/ret)

Context sensitive can tell one call returns 4, the other 5

```
a = id(4);
                void id(int z)
                { return z; }
b = id(5);
```

Context-Insensitive
(note merging)

Context insensitive will say both calls return {4,5}

Context Sensitivity Problems

Suppose A and B both call C.

- CFI uses same return label in A and B.

How to prevent C from returning to B when it was called from A?

- **Solultion: Shadow Call Stack**
  - a protected memory region for call stack
  - each call/ret instrumented to update shadow
  - CFI ensures instrumented checks will be run

Security Guarantees

Effective against attacks based on illegitimate control-flow transfer

- buffer overflow, ret2libc, ROP, pointer subterfuge, etc.

Any check becomes non-circumventable.

Security Guarantees

Effective against attacks based on illegitimate control-flow transfer

- buffer overflow, ret2libc, ROP, pointer subterfuge, etc.

Any check becomes non-circumventable.

Allow data-only attacks since they respect CFG!

- incorrect usage (e.g. printf can still dump mem)
- substitution of data (e.g. replace file names)

CFI an active area of research

CCS 2015:

- Per-Input Control-Flow Integrity
- Practical Context-Sensitive CFI
- CCFI: Cryptographically Enforced Control Flow Integrity
- Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks

CFI Summary

Control Flow Integrity ensures that control flow follows a path in CFG

- Accuracy of CFG determines level of enforcement
- Can build other security policies on top of CFI
- Simple version now deployed in Windows 10 (a slow but continuing trend)

Agenda

Reference Monitors ✔

Control Flow Integrity ✔

Software Fault Isolation

Agenda

Reference Monitors

Control Flow Integrity

Software Fault Isolation

NEXT

Software Fault Isolation

Optional Reading:
Efficient Software-Based Fault Isolation
by Wahbe, Lucco, Anderson, Graham

Motivation: 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

Isolation Mechanisms

Isolation Mechanisms

- Hardware
  - Memory Protection (virtual address translation, x86 segmentation)

Isolation Mechanisms

- Hardware
  - Memory Protection (virtual address translation, x86 segmentation)
- Software
  - Sandboxing
  - Language-Based

Isolation Mechanisms

- Hardware
  - Memory Protection (virtual address translation, x86 segmentation)
- Software
  - Sandboxing
  - Language-Based
- Hardware + Software
  - Virtual machines

Isolation Mechanisms

- Hardware
  - Memory Protection (virtual address translation, x86 segmentation)
- Software
  - Sandboxing
  - Language-Based
- Hardware + Software
  - Virtual machines

Software Fault Isolation
≈
Memory Protection
in Software

Software Fault Isolation

- SFI ensures that a module only accesses memory within its region by adding ***checks*** (also a type of Inline Reference Monitor)
  - e.g., a plugin can accesses only its own memory

```
if(module_lower < x < module_upper)
    z = load[x];
```

SFI Check

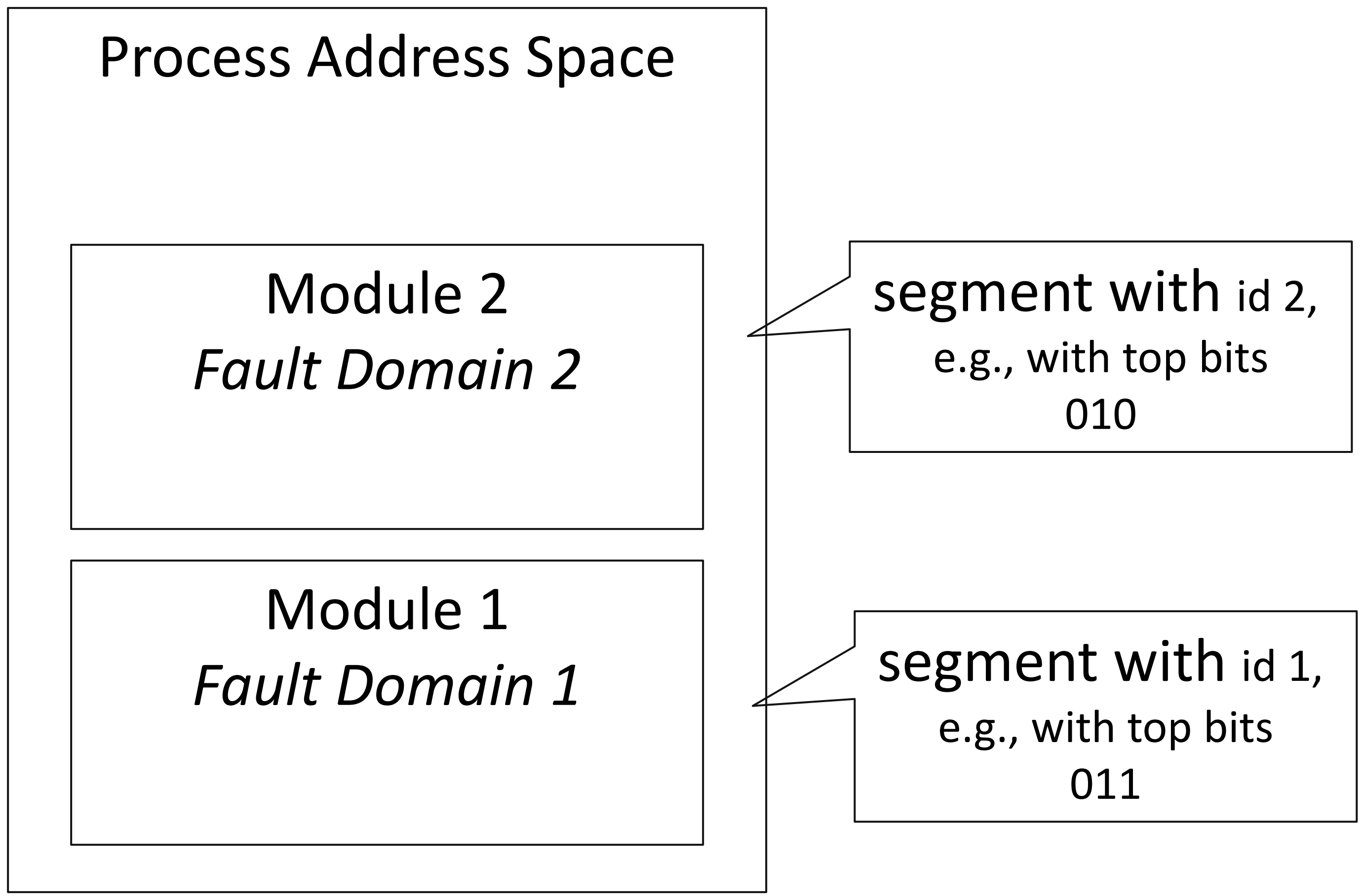
- CFI as a building block can ensure inserted memory checks are executed

SFI Goals

- Confine faults inside distrusted extensions
  - codec shouldn't compromise media player
  - device driver shouldn't compromise kernel
  - plugin shouldn't compromise web browser

- Allow for efficient cross-domain calls
  - numerous calls between media player and codec
  - numerous calls between device driver and kernel

Main Idea


Process Address Space
Module 2
Fault Domain 2
Module 1
Fault Domain 1
segment with id 2, e.g., with top bits 010
segment with id 1, e.g., with top bits 011

SFI Example

```
int compute_sum( int a[], int len )
{
    int sum = 0;
    for(int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

```
      ...
      mov  ecx, 0h              ; int i = 0
      mov  esi, [esp+8]         ; a[] base ptr
LOOP: and  esi, 20FFFFFFh       ; SFI masking
      add  eax, [esi+ecx*4]     ; sum += a[i]
      inc  ecx                  ; ++i
      cmp  ecx, edx             ; i < len
      jl   LOOP
```

Optimizing SFI using CFI

```c
int compute_sum( int a[], int len )
{
    int sum = 0;
    for(int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

```
      ...
      mov  ecx, 0h             ; int i = 0
      mov  esi, [esp+8]        ; a[] base ptr
      and  esi, 20FFFFFFh      ; SFI masking
LOOP: add  eax, [esi+ecx*4]    ; sum += a[i]
      inc  ecx                 ; ++i
      cmp  ecx, edx            ; i < len
      jl   LOOP
```

Agenda

Reference Monitors

Control Flow Integrity

Software Fault Isolation

Questions