Project 1: 2 days left

Prevention-Based Cyber Security: Reference Monitor

CS 459/559: Science of Cyber Security
12th Lecture

Instructor:

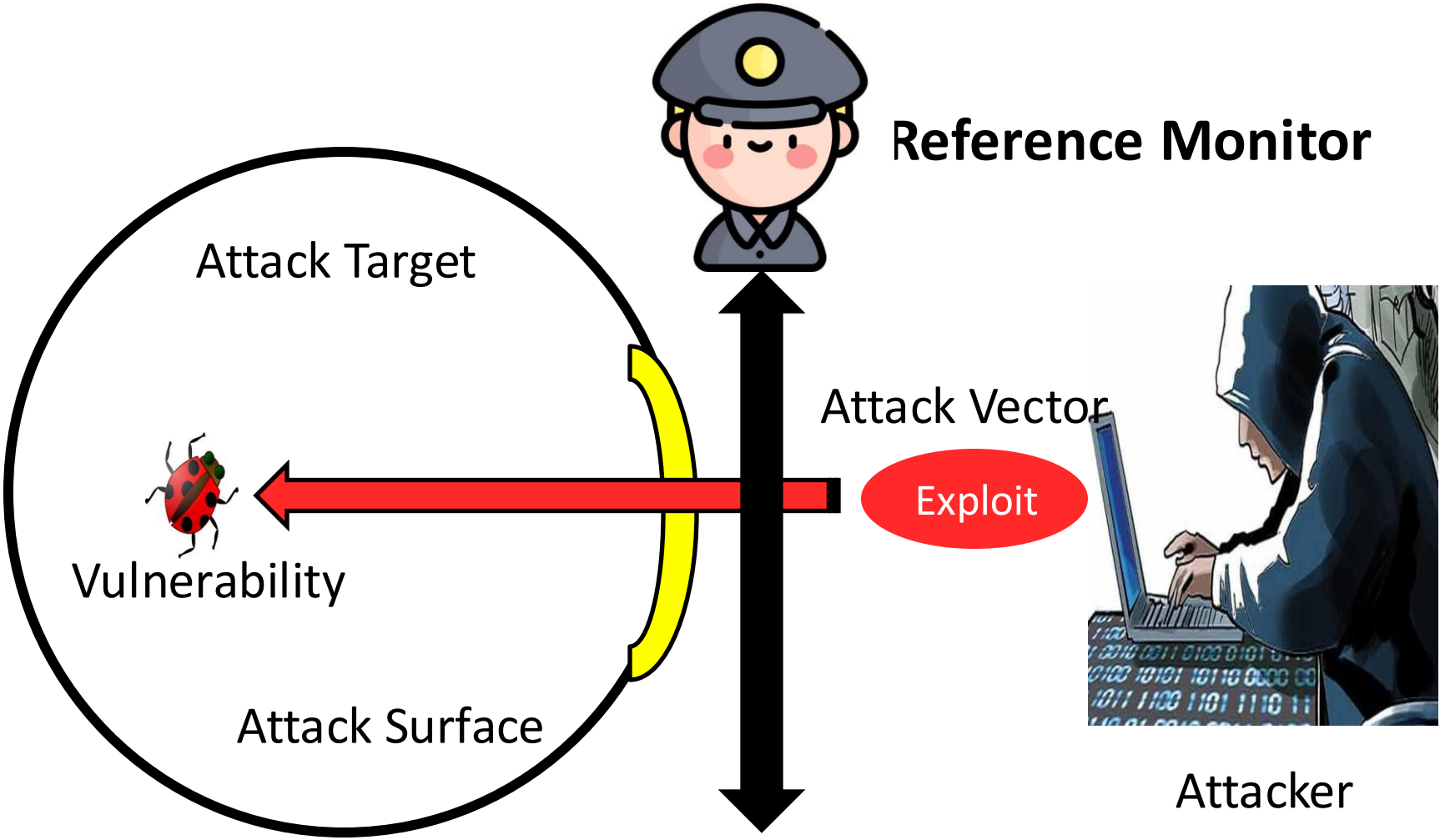
Guanhua Yan

Agenda

- ~~Quiz 1: September 29 (closed book)~~
- Project 1 (offense): October 10
- Project 2 (defense): December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final report: December 15

Goal of reference monitor

- Idea: enforce security policy at the gateway



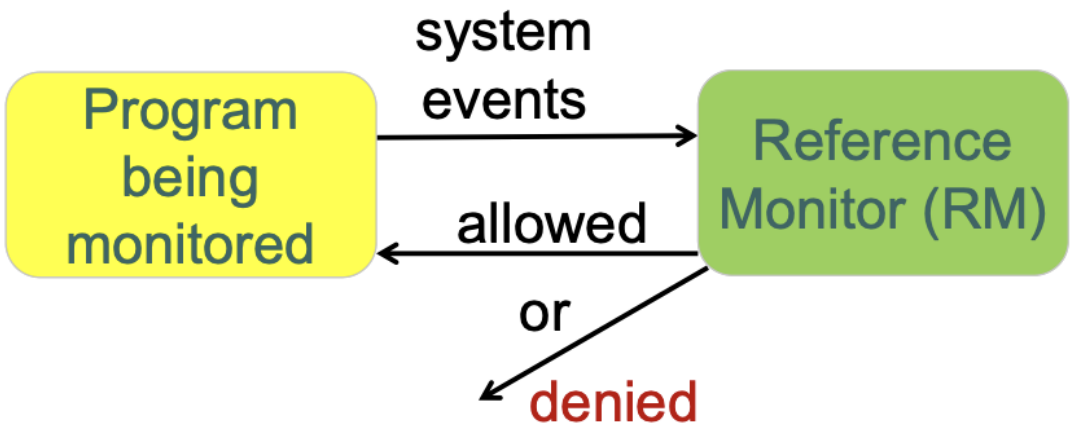
Outline

- **What is reference monitor**
- **Power of reference monitor**
- **Examples of reference monitor**
  - Hardware-based protection
  - Software fault isolation
  - Control flow integrity

Reference Monitors

Reference Monitor

- Observe the execution of a program and halt the program if it's going to violate the security policy.



Common Examples of RM

- Operating system monitors users applications
  - Monitor system calls by user apps
  - Kernel vs user mode
  - Hardware based
- Software-based: Interpreters, language virtual machines, software-based fault isolation
- Firewalls
- …
- Claim: majority of today's security enforcement mechanisms are instances of reference monitors

Requirements for a Monitor

- Must have (reliable) access to information about what the program is about to do
  - e.g., what syscall is it about to execute?
- Must have the ability to "stop" the program
  - can't stop a program running on another machine that you don't own
  - stopping isn't necessary; transitioning to a "good" state may be sufficient
- Must protect the monitor's state and code from tampering
  - key reason why a kernel's data structures and code aren't accessible by user code
- In practice, must have low overhead

Requirements for a Monitor

- Must have (reliable) access to information about what the program is about to do
  - e.g., what syscall is it about to execute?
- Must have the ability to "stop" the program
  - can't stop a program running on another machine that you don't own
  - stopping isn't necessary; transitioning to a "good" state may be sufficient
- Must protect the monitor's state and code from tampering
  - key reason why a kernel's data structures and code aren't accessible by user code
- In practice, must have low overhead

Requirements for a Monitor

- Must have (reliable) access to information about what the program is about to do
  - e.g., what syscall is it about to execute?
- Must have the ability to "stop" the program
  - can't stop a program running on another machine that you don't own
  - stopping isn't necessary; transitioning to a "good" state may be sufficient
- Must protect the monitor's state and code from tampering
  - key reason why a kernel's data structures and code aren't accessible by user code
- In practice, must have low overhead

Requirements for a Monitor

- Must have (reliable) access to information about what the program is about to do
  - e.g., what syscall is it about to execute?
- Must have the ability to "stop" the program
  - can't stop a program running on another machine that you don't own
  - stopping isn't necessary; transitioning to a "good" state may be sufficient
- Must protect the monitor's state and code from tampering
  - key reason why a kernel's data structures and code aren't accessible by user code
- In practice, must have low overhead

What Policies Can be Enforced?

- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can't guess the future – the decision to determine whether to halt a program must be computable
- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: **safety properties**
  - There are desirable policies that no reference monitor can enforce precisely

What Policies Can be Enforced?

- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can't guess the future – the decision to determine whether to halt a program must be computable
- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: **safety properties**
  - There are desirable policies that no reference monitor can enforce precisely

What Policies Can be Enforced?

- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can't guess the future – the decision to determine whether to halt a program must be computable
- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: **safety properties**
  - There are desirable policies that no reference monitor can enforce precisely

What Policies Can be Enforced?

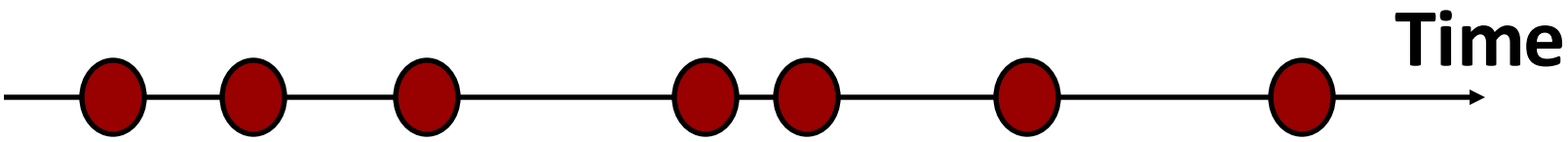
- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can't guess the future – the decision to determine whether to halt a program must be computable
- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: **safety properties**
  - There are desirable policies that no reference monitor can enforce precisely

Analysis of the Power and Limitations of Execution Monitoring

"Enforceable Security Policies" by Fred Schneider

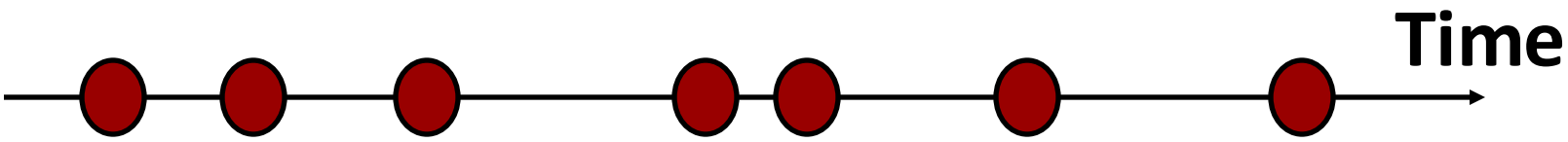
Execution Traces

- System behavior σ: a finite or infinite execution trace of system events
  - $\sigma = e_0 e_1 e_2 e_3 \ldots e_i \ldots$, where e_i is a system event
- Example: a trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)
- Example: a trace of system calls
  - System-call events: open(…); read(…); close(…); gettimeofday(…); fork(…); …
- Example: a system for access control
  - Oper(p, o, r): Principal p invoked an operation involving object o and requiring right r to that object
  - AddP(p, p'): Principal p invoked an operation to create a principal named p'
  - …



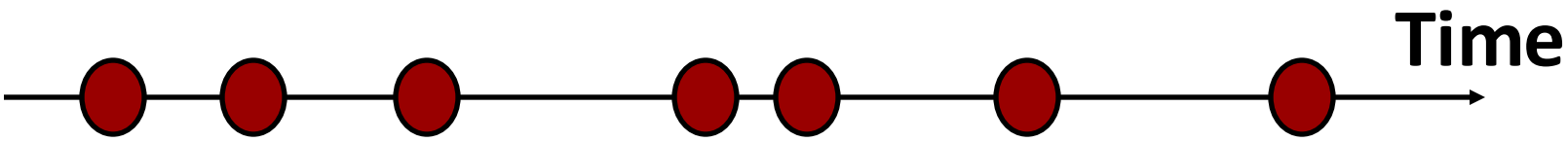
Execution Traces

- System behavior σ: a finite or infinite execution trace of system events
  - $\sigma = e_0 e_1 e_2 e_3 \ldots e_i \ldots$, where e_i is a system event
- Example: a trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)
- Example: a trace of system calls
  - System-call events: open(…); read(…); close(…); gettimeofday(…); fork(…); …
- Example: a system for access control
  - Oper(p, o, r): Principal p invoked an operation involving object o and requiring right r to that object
  - AddP(p, p'): Principal p invoked an operation to create a principal named p'
  - …



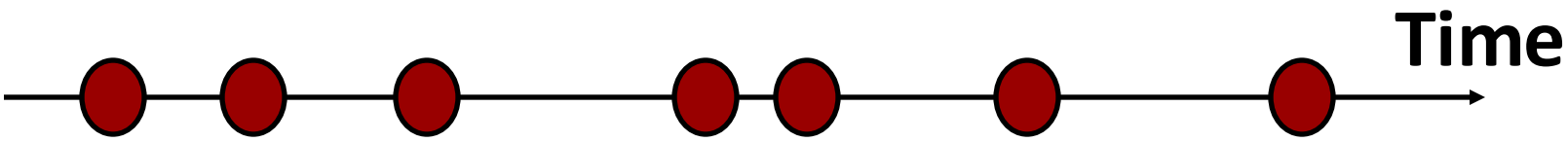
Execution Traces

- System behavior σ: a finite or infinite execution trace of system events
  - $\sigma = e_0 e_1 e_2 e_3 \ldots e_i \ldots$, where e_i is a system event
- Example: a trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)
- Example: a trace of system calls
  - System-call events: open(…); read(…); close(…); gettimeofday(…); fork(…); …
- Example: a system for access control
  - Oper(p, o, r): Principal p invoked an operation involving object o and requiring right r to that object
  - AddP(p, p'): Principal p invoked an operation to create a principal named p'
  - …



Execution Traces

- System behavior σ: a finite or infinite execution trace of system events
  - $\sigma = e_0 e_1 e_2 e_3 \ldots e_i \ldots$, where e_i is a system event
- Example: a trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)
- Example: a trace of system calls
  - System-call events: open(…); read(…); close(…); gettimeofday(…); fork(…); …
- Example: a system for access control
  - Oper(p, o, r): Principal p invoked an operation involving object o and requiring right r to that object
  - AddP(p, p'): Principal p invoked an operation to create a principal named p'
  - …



Modeling a System

- A **system** modeled as a set of execution traces (its behaviors)
  - S= $\{\sigma_1, \sigma_2, \ldots, \sigma_k, \ldots\}$
  - Each trace corresponds to the execution for a possible input
- For example
  - Sets of traces of reads and writes
  - Sets of traces of system calls

Definition of Security Policy

- A **security policy** P(S): a logical predicate on sets of execution traces.
  - A target system S satisfies security policy P if and only if P(S) holds.
- For example
  - A program cannot write to addresses outside of [0, 1000]
    - $P(S) = \forall \sigma \in S. \forall e_i \in \sigma.$
      $e_i = write(addr,v) \rightarrow addr \in [0, 1000]$
  - A program cannot send a network packet after reading from a local file
    - $P(S) = \forall \sigma \in S. \forall e_i \in \sigma.$
      $e_i = fileRead(...) \rightarrow \forall k>i.\ e_k \neq networkSend(...)$

Constraint on Monitors: Property

- Can a reference monitor see more than one trace at a time?
  - A reference monitor only sees one execution trace of a program
- So we can only enforce policies P s.t.:
  - **(1)** $P(S) = \forall \sigma \in S. \mathcal{P}(\sigma)$
  - where $\mathcal{P}$ is a predicate on individual traces
- A security policy is a **property** if its predicate specifies whether an individual trace is legal
  - The membership is determined solely by the trace and not by the other traces

Enforceability

What is a Non-Property?

- A policy that may depend on multiple execution traces
- Information flow polices
  - Sensitive information should not flow to unauthorized person explicitly or **implicitly**
  - Example: a system protected by passwords
    - Suppose the password checking time correlates closely to the length of the prefix that matches the true password
      - Timing channel
    - To rule this out, a policy should say: no matter what the input is, the password checking time should be the same **in all traces**
    - Not a property

More Constraints on Monitors

Shouldn't be able to "see" the future.

- Assumption: must make decisions in finite time.
- Suppose $\mathcal{P}(\sigma)$ is false, then it must be rejected at some finite time i; that is, $\mathcal{P}(\sigma[..i])$ is false

(2) $\forall \sigma.\ \neg \mathcal{P}(\sigma) \rightarrow (\exists i.\ \neg \mathcal{P}(\sigma[..i]))$

Once a trace has been rejected by the monitor, then any further events from the system cannot make the monitor to revoke that decision

(3) $\forall \sigma.\ \neg \mathcal{P}(\sigma) \rightarrow (\forall \sigma'.\ \neg \mathcal{P}(\sigma\sigma'))$

Prefix Closure

More Constraints on Monitors

Shouldn't be able to "see" the future.

- Assumption: must make decisions in finite time.
- Suppose $\mathcal{P}(\sigma)$ is false, then it must be rejected at some finite time i; that is, $\mathcal{P}(\sigma[..i])$ is false

(2) $\forall \sigma.\ \neg \mathcal{P}(\sigma) \rightarrow (\exists i.\ \neg \mathcal{P}(\sigma[..i]))$

Once a trace has been rejected by the monitor, then any further events from the system cannot make the monitor to revoke that decision

(3) $\forall \sigma.\ \neg \mathcal{P}(\sigma) \rightarrow (\forall \sigma'.\ \neg \mathcal{P}(\sigma\sigma'))$

Finite refutability

Reference Monitors Enforce Safety Properties

A predicate P on sets of sequences s.t.

(1) $P(S) = \forall \sigma \in S. \mathcal{P}(\sigma)$

(2) $\forall \sigma. \neg \mathcal{P}(\sigma) \rightarrow (\exists i. \neg \mathcal{P}(\sigma[..i]))$

(3) $\forall \sigma. \neg \mathcal{P}(\sigma) \rightarrow (\forall \sigma'. \neg \mathcal{P}(\sigma\sigma'))$

is a ***safety property***: "no bad thing will happen."

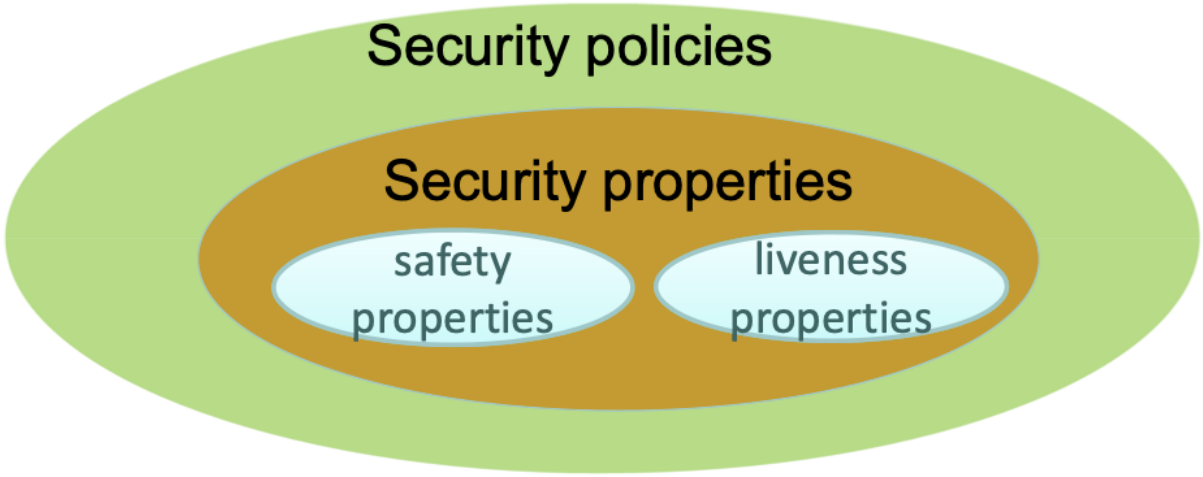
Conclusion: a reference monitor can't enforce a policy P unless it's a safety property.

Safety and Liveness Properties
[Alpern & Schneider 85,87]

- Safety: Some "bad thing" doesn't happen.
  - Proscribes traces that contain some "bad" prefix
  - Example: the program won't read memory outside of range [0,1000]
- Liveness: Some "good thing" does happen
  - Example: program will terminate
  - Example: program will eventually release the lock
- Theorem: Every security property can be decomposed into a safety property and a liveness property

Classification of Policies

- "Enforceable Security Policies" [Schneider 00]



Policies Enforceable by Reference Monitors

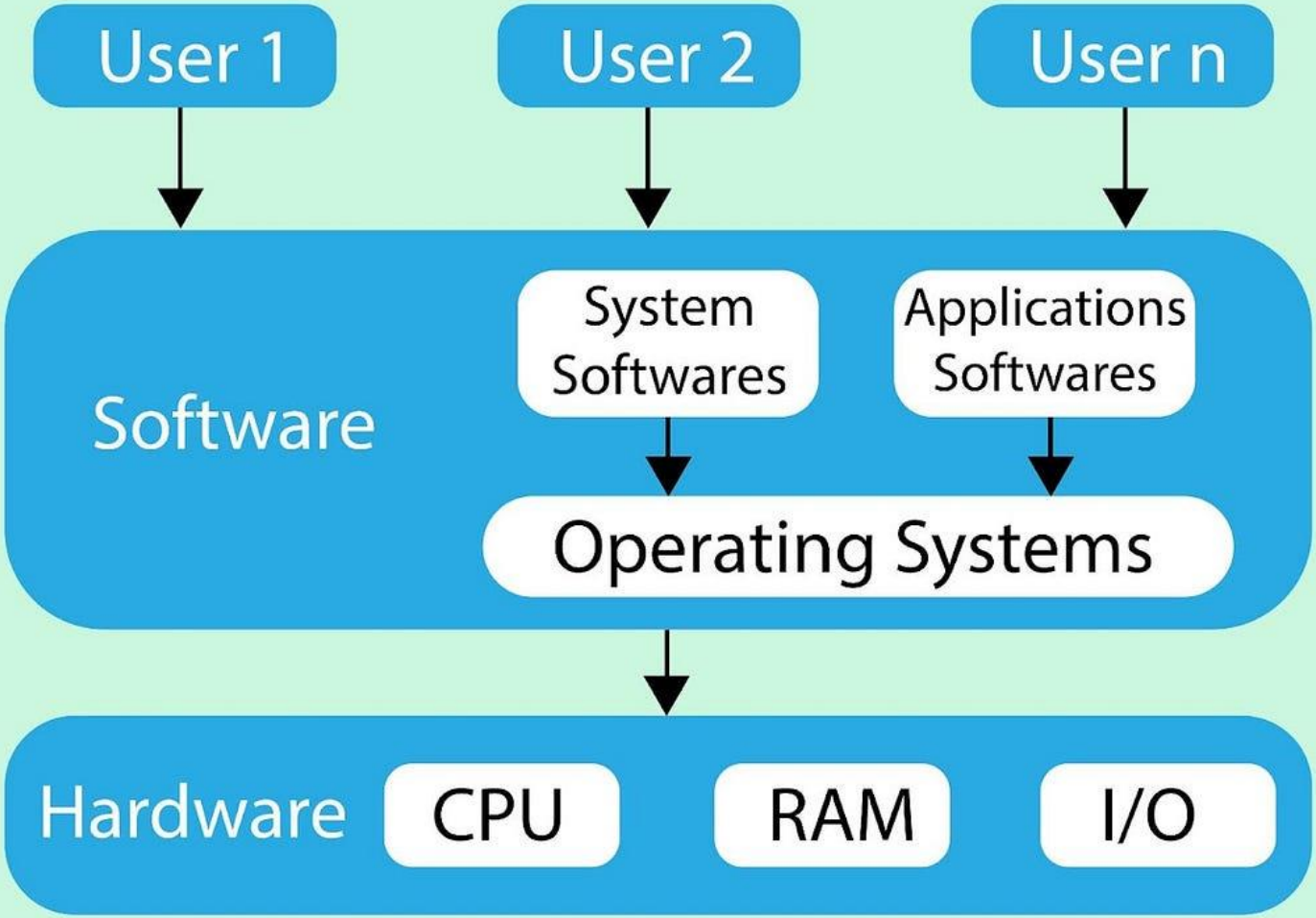
- Reference monitor **can** enforce any safety property
  - Intuitively, the monitor can inspect the history of computation and prevent bad things from happening
- Reference monitor **cannot** enforce liveness properties
  - The monitor cannot predict the future of computation
- Reference monitor **cannot** enforce non-properties
  - The monitor cannot inspect multiple traces simultaneously

Safety Is Nice

Safety has its benefits:

- They compose: if P and Q are safety properties, then P & Q is a safety property (just the intersection of allowed traces.)
- Safety properties can approximate liveness by setting limits. e.g., we can determine that a program terminates within k steps.
- We can also approximate many other security policies (e.g., info. flow) by simply choosing a stronger safety property.

Reference Monitor Example 1: Hardware-Based Protection


User 1
User 2
User n
Software
System Softwares
Applications Softwares
Operating Systems
Hardware
CPU
RAM
I/O

Operating Systems circa '75

Simple Model: system is a collection of running processes and files.

- processes perform actions on behalf of a user.
  - open, read, write files
  - read, write, execute memory, etc.
- files have access control lists dictating which users can read/write/execute/etc. the file.

(Some) High-Level Policy Goals:

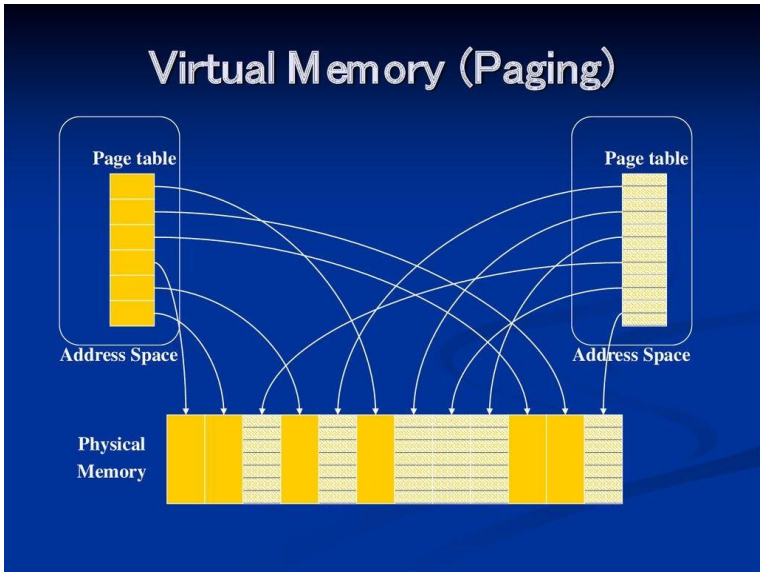
- Integrity: one user's processes shouldn't be able to corrupt the code, data, or files of another user.
- Availability: processes should eventually gain access to resources such as the CPU or disk.
- Confidentiality? Access control?

What Can go Wrong?

- read/write/execute or change ACL of a file for which process doesn't have proper access.
  - check file access against ACL
- process writes into memory of another process
  - isolate memory of each process (& the OS!)
- process pretends it is the OS and execute its code
  - maintain process ID and keep certain operations privileged --- need some way to transition.
- process never gives up the CPU
  - force process to yield in some finite time
- process uses up all the memory or disk
  - enforce quotas

Hardware-Based Protection

- Based on virtual memory
  - Protect one process from reading/writing to other processes' memory locations
  - Memory safety (fault isolation) at the process level



Virtual Memory

- Each process assumes a virtual address space
- A page table translates virtual pages to physical pages
  - Dedicated hardware for acceleration: TLB

virtual page	physical page	permission
0	0xABCD0000	rw
1	0xA0700000	r
⋮		

A **translation lookaside buffer** (**TLB**) is a memory cache that stores the recent translations of virtual memory to physical memory. It is used to reduce the time taken to access a user memory location.

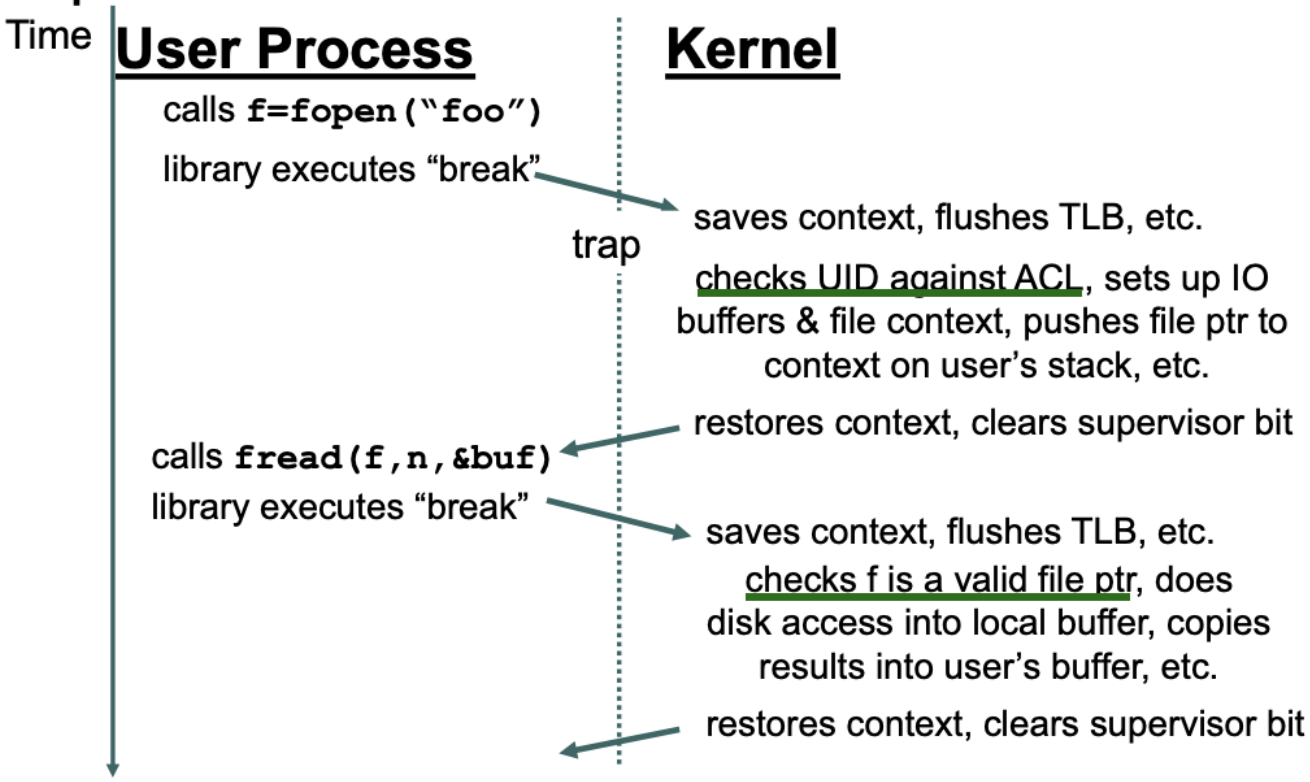
Hardware Privilege Modes

- Generally two: kernel mode & user mode (x86 has more)
- At any time, CPU is in some mode
- Dangerous ("privileged") instructions usable only in the kernel mode
  - E.g., direct access to any physical memory location, or privilege-mode change
  - Violations trap to a fixed address in the kernel

Moving Between Privilege Modes

- Downgrade privilege
  - Usually have a simple instruction for this
- Upgrade privilege
  - Usually have a special “system call” instruction (like function call), but
    - Go to a fixed address or through a fixed jumptable
    - Change into privileged (kernel) mode

Steps in a System Call



Pros and Cons of Hardware-Based Protection

- Pros
  - Low overhead; built into the hardware
  - Transparent for applications
- Cons
  - Inter-process communication is cumbersome and slow
    - Signals, remote procedural calls (RPC), pipes, sockets, shared memory, …
    - Involves context switch (changes to a new page table)
  - Granularity of protection is per-process
    - Doesn't protect a buffer of size, say, 100 bytes
    - Buffer overflow attacks apply

Reference Monitor Example 2: Software Fault Isolation

SFI Example: Google Chrome Sandbox

A new approach to browser security: the Google Chrome Sandbox

Thursday, October 2, 2008

Building a secure browser is a top priority for the Chromium team; it's why we spend a lot of time and effort keeping our code secure. But as you can imagine, code perfection is something almost impossible to achieve for a project of this size and complexity. To make things worse, a browser spends most of its time handling and executing untrusted and potentially malicious input data. In the event that something goes wrong, the team has developed a sandbox to help thwart any exploit in two of the most popular vectors of attack against browsers: HTML Rendering and JavaScript execution.

In a nutshell, a sandbox is security mechanism used to run an application in a restricted environment. If an attacker is able to exploit the browser in a way that lets him run arbitrary code on the machine, the sandbox would help prevent this code from causing damage to the system. The sandbox would also help prevent this exploit from modifying and even reading your files or any information on the system.

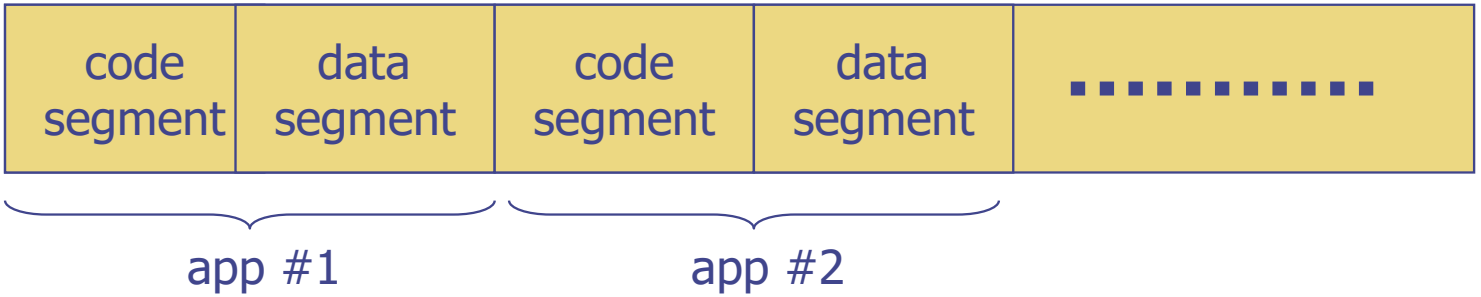
https://blog.chromium.org/2008/10/new-approach-to-browser-security-google.html

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

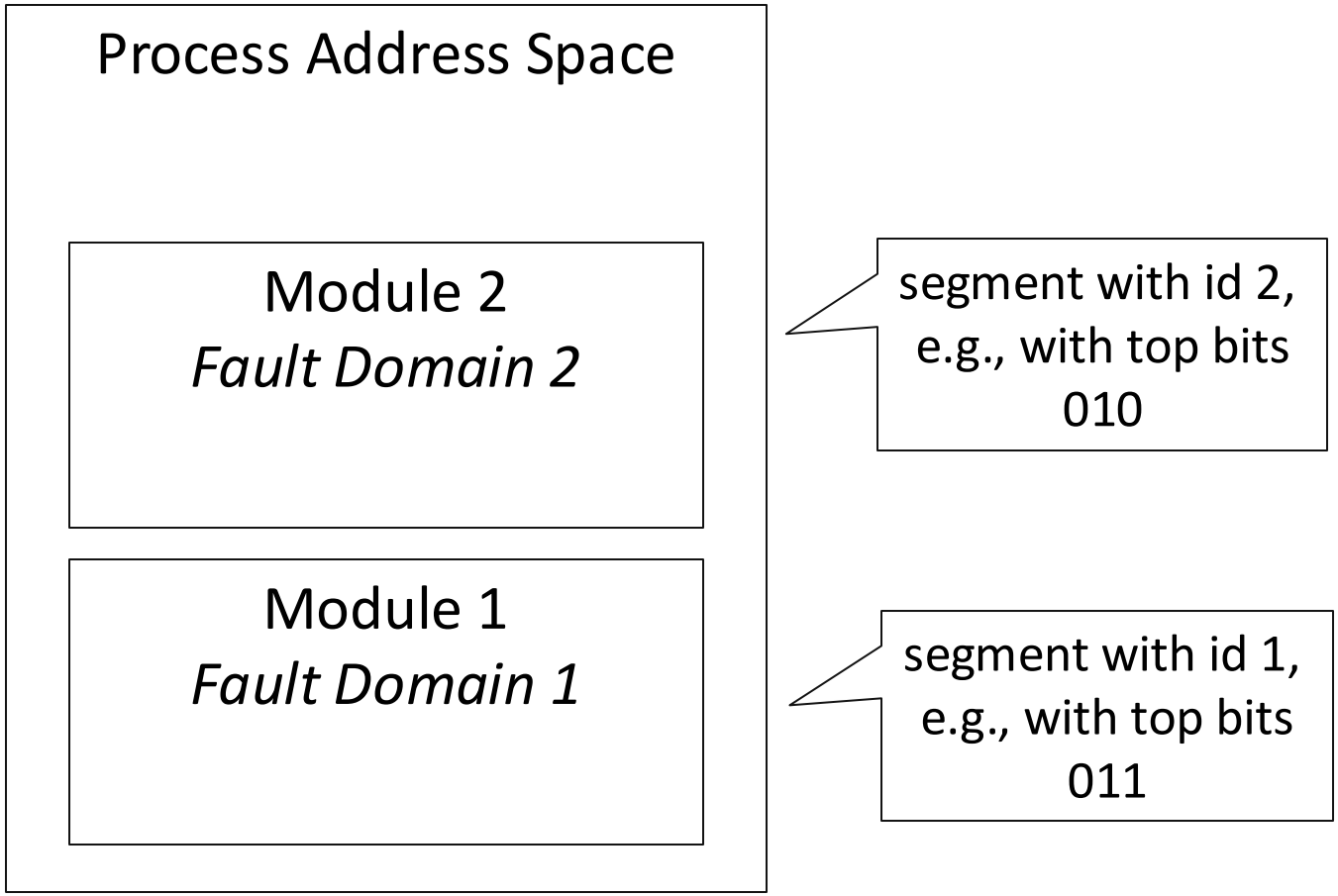
Software Fault Isolation

- SFI approach:
  - Partition process memory into segments



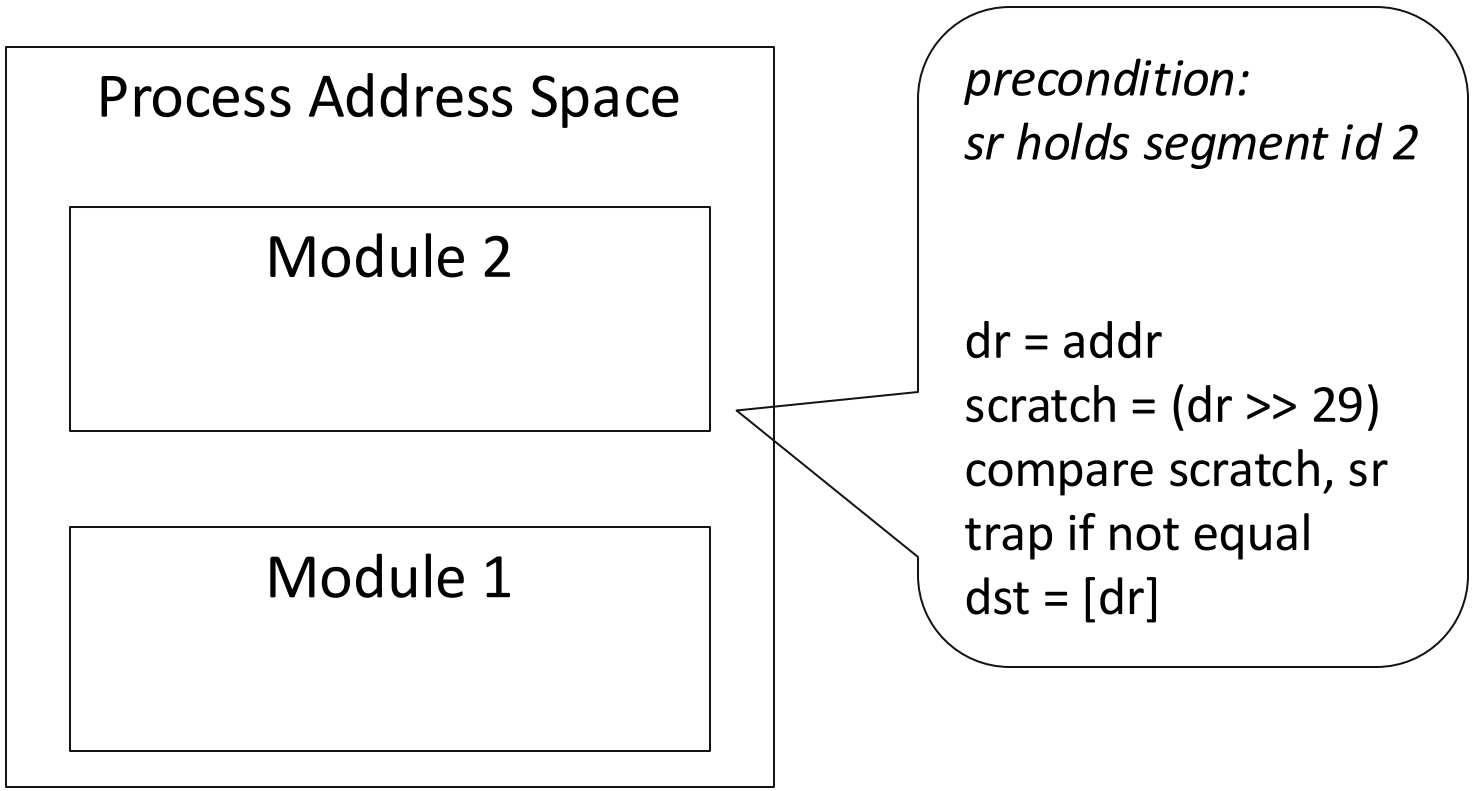
  - Locate unsafe instructions: jmp, load, store
    - At compile time, add guards before unsafe instructions
    - When loading code, ensure all guard are present

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

Scheme 1: Segment Matching

- *Check* every memory access for matching segment id
- Assume dedicated registers segment register (sr) and data register (dr)
  - not available to the program

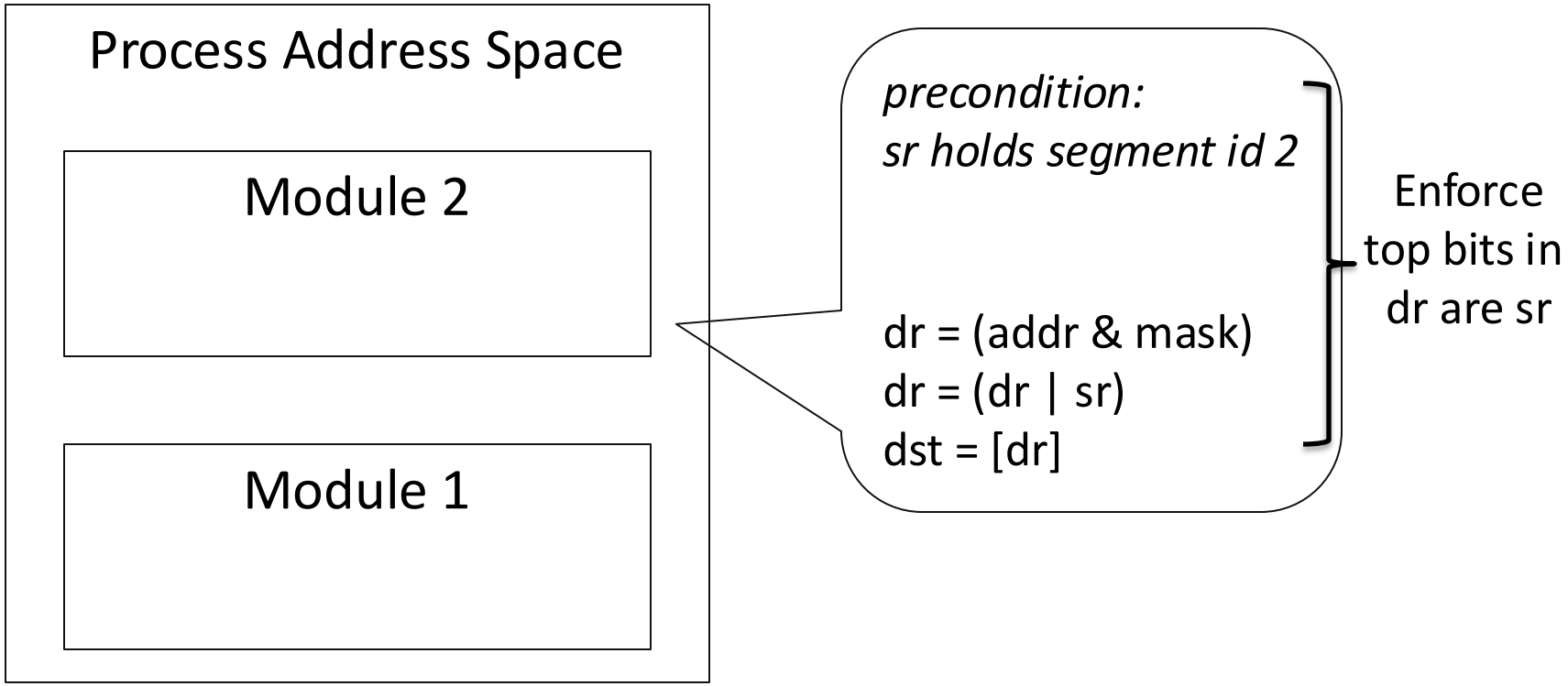


Safety

- Segment matching code must always be run to ensure safety.
- Dedicated registers must not be writeable by module.

Scheme 2: Sandboxing

- **Force top bits to match seg id and continue**
- **No comparison is made**



Segment Matching vs. Sandboxing

Segment Matching

- more instructions
- can pinpoint exact point of fault where segment id doesn't match

Sandboxing

- fewer instructions
- just ensures memory access stays in region (crash is ok)

Reference Monitor Example 3: Control Flow Integrity

Microsoft Control Flow Guard

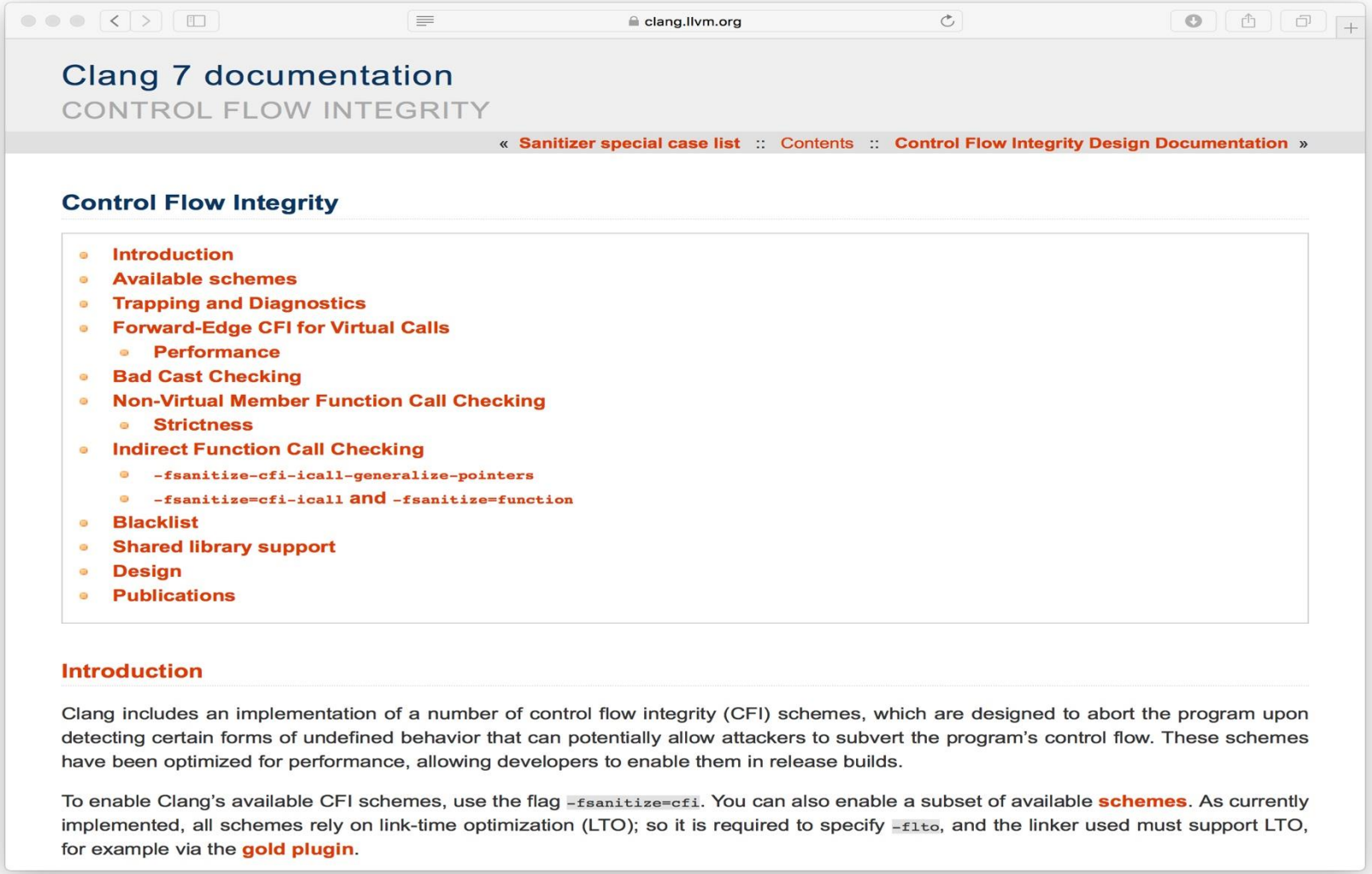
What is Control Flow Guard?

Control Flow Guard (CFG) is a highly-optimized platform security feature that was created to combat memory corruption vulnerabilities. By placing tight restrictions on where an application can execute code from, it makes it much harder for exploits to execute arbitrary code through vulnerabilities such as buffer overflows. CFG extends previous exploit mitigation technologies such as /GS, DEP, and ASLR.

This feature is available in Microsoft Visual Studio 2015, and runs on "CFG-Aware" versions of Windows—the x86 and x64 releases for Desktop and Server of Windows 10 and Windows 8.1 Update (KB3000850).

https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

LLVM Clang



clang.llvm.org

Clang 7 documentation

CONTROL FLOW INTEGRITY

« Sanitizer special case list :: Contents :: Control Flow Integrity Design Documentation »

Control Flow Integrity

Introduction

Clang includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow. These schemes have been optimized for performance, allowing developers to enable them in release builds.

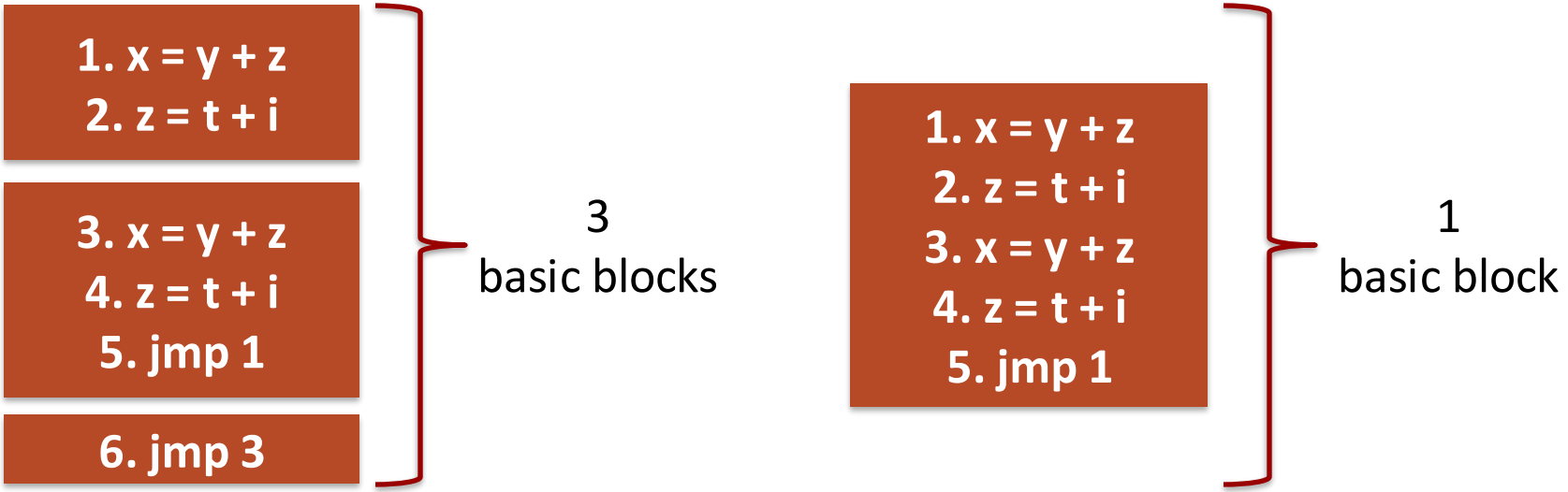
To enable Clang's available CFI schemes, use the flag `-fsanitize=cfi`. You can also enable a subset of available schemes. As currently implemented, all schemes rely on link-time optimization (LTO); so it is required to specify `-flto`, and the linker used must support LTO, for example via the gold plugin.

https://clang.llvm.org/docs/ControlFlowIntegrity.html

Basic Block

Definition of Basic Block: A consecutive sequence of instructions / code such that

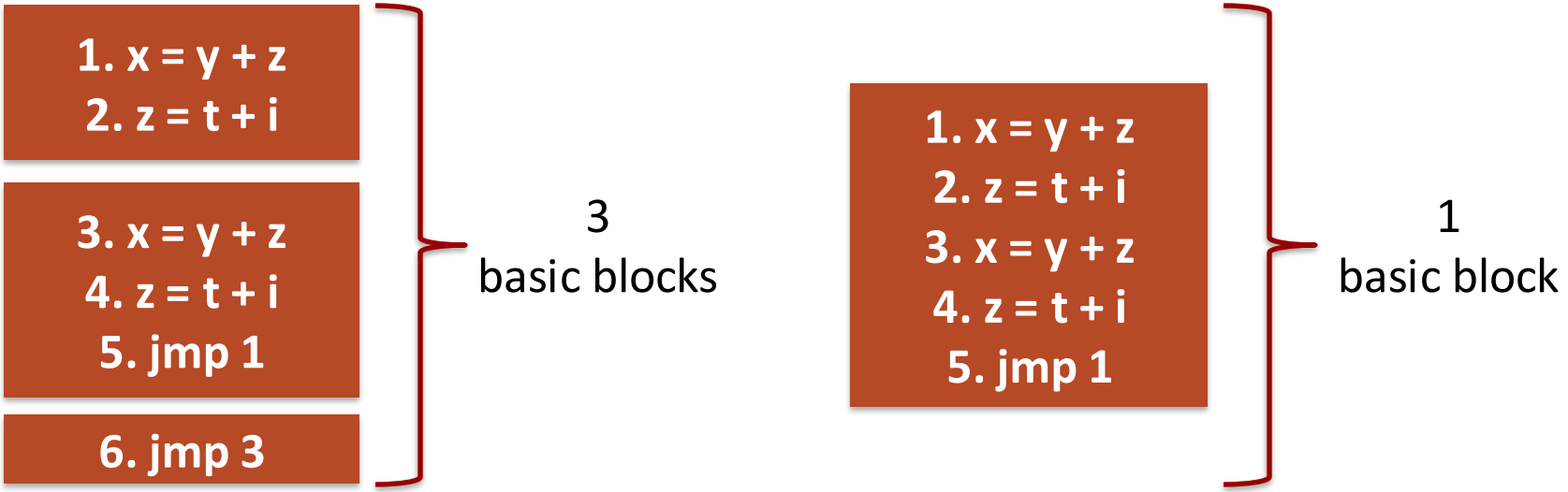
- the instruction in each position always executes before (dominates) all those in later positions, and
- no outside instruction can execute between two instructions in the sequence



Basic Block in CFG

Definition of Basic Block: A consecutive sequence of instructions / code such that

- control is "straight"
- (no jump targets except at the beginning,
- no jumps except at the end)



CFG Definition

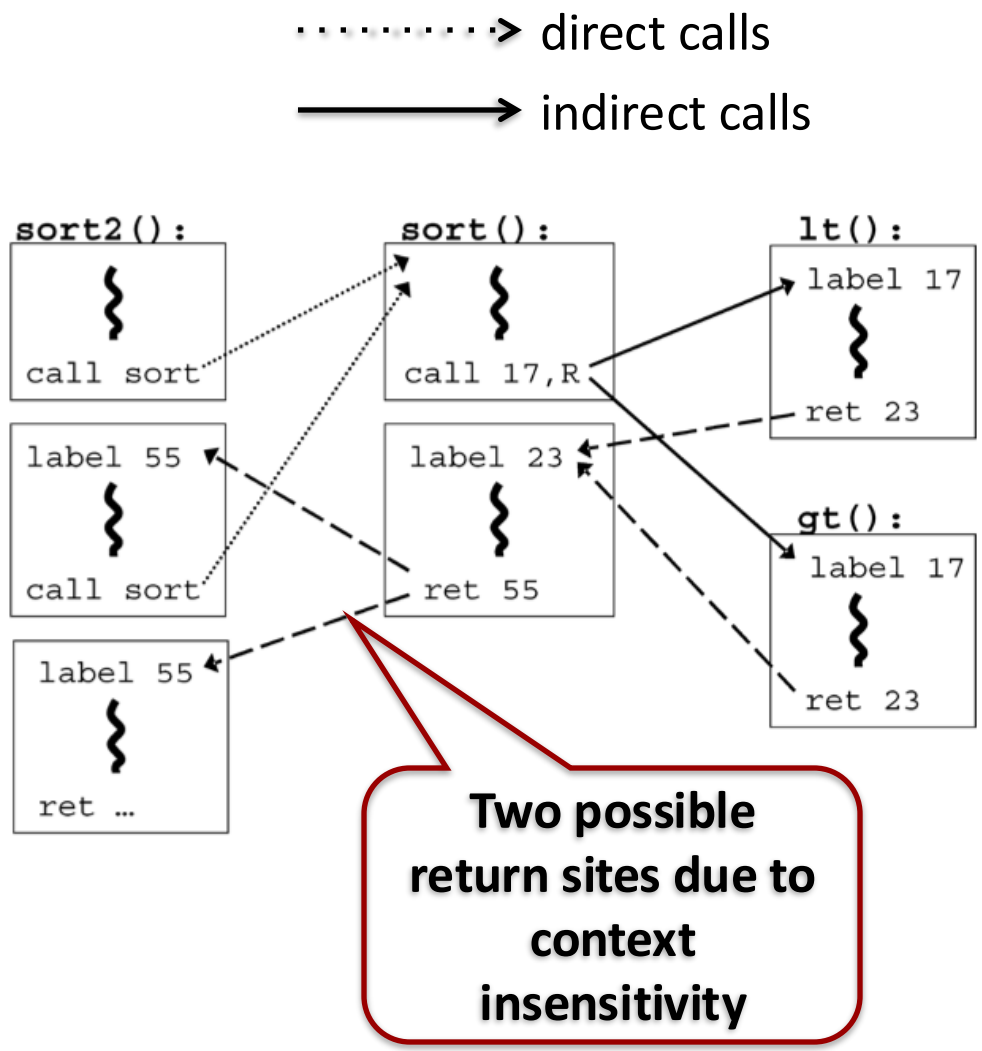
A static ***Control Flow Graph*** is a graph where

- each vertex v_i is a basic block, and
- there is an edge (v_i, v_j) if there ***may*** be a transfer of control from block v_i to block v_j.

Historically, the scope of a "CFG" is limited to a function or procedure, i.e., *intra*-procedural.

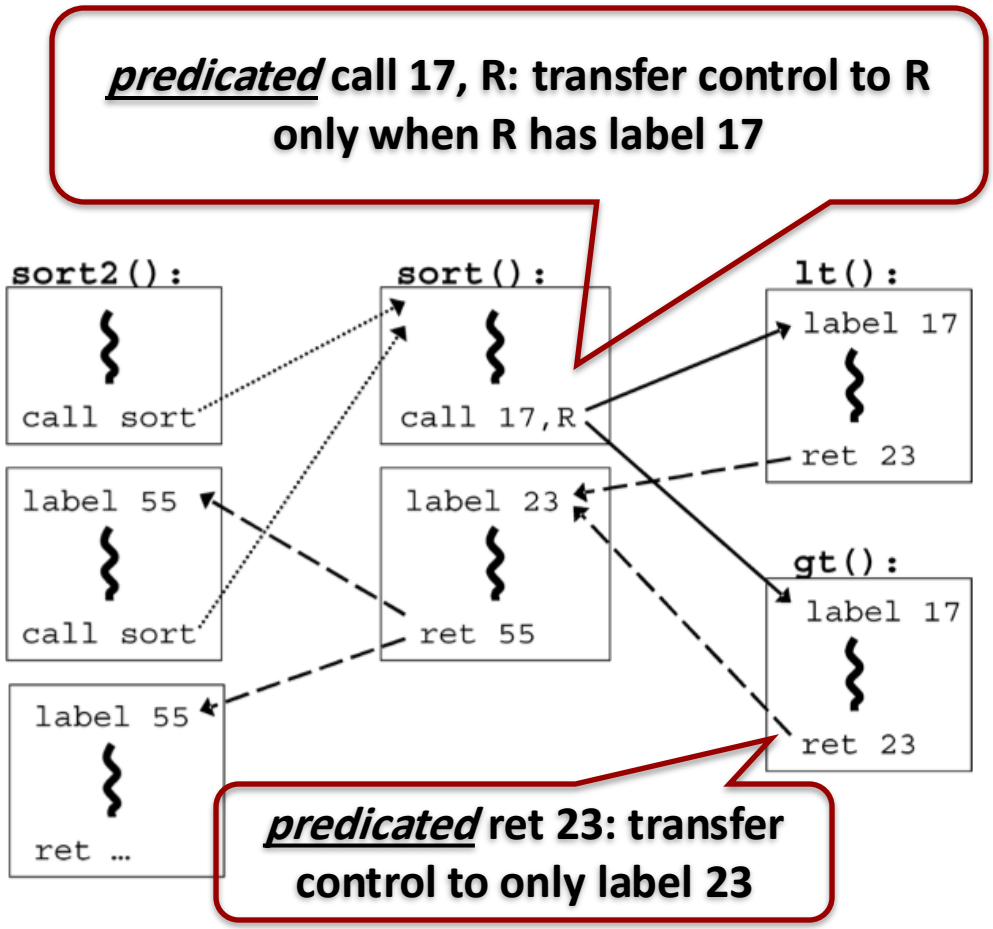
Build CFG

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Instrument Binary

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

Example of instrumentation

Function Call

Opcode bytes	Instructions	
FF 53 08	call [ebx+8]	; call fptr

Function Return

Opcode bytes	Instructions	
C2 10 00	ret 10h	; return

are instrumented using `prefetchnta` destination IDs, to become

```
8B 43 08                mov  eax, [ebx+8]       ; load fptr
3E 81 78 04 78 56 34 12 cmp  [eax+4], 12345678h ; comp w/ID
75 13                   jne  error_label        ; if != fail
FF D0                   call eax                ; call fptr
3E 0F 18 05 DD CC BB AA prefetchnta [AABBCCDDh] ; label ID
```

```
8B 0C 24       mov  ecx, [esp]  ; load ret
83 C4 14       add  esp, 14h    ; pop 20
3E 81 79 04    cmp  [ecx+4],    ; compare
DD CC BB AA           AABBCCDDh ;     w/ID
75 13          jne  error_label ; if!=fail
FF E1          jmp  ecx         ; jump ret
```

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

Jump to the destination only if the tag is equal to "AABBCCDD"

Prefetchnta: prefetch data into non-temporal cache structure and into a location close to the processor

Verify CFI Instrumentation

- **Direct jump targets (e.g. `call 0x12345678`)**
  - are all targets valid according to CFG?
- **IDs**
  - is there an ID right after every entry point?
  - does any ID appear in the binary by accident?
- **ID Checks**
  - is there a check before every control transfer?
  - does each check respect the CFG?

easy to implement correctly => trustworthy

CFI: Preventing Circumvention

- **Unique IDs**
  - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks
- **Non-writable code**
  - Program should not modify code memory at runtime
    - What about run-time code generation and self-modification?
- **Non-executable data**
  - Program should not execute data as if it were code
- **Enforcement**: hardware support + prohibit system calls that change protection state + verification at load-time

CFI: Security Guarantees

- **Effective against attacks based on illegitimate control-flow transfer**
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- **Does not protect against attacks that do not violate the program's original CFG**
  - Incorrect arguments to system calls
  - Substitution of file names
  - Other data-only attacks

End of Lecture 12