CSE509 : Computer System Security

Securing Untrusted Code

Untrusted Code

- May be untrustworthy
  - Intended to be benign, but may be full of vulnerabilities
  - These vulnerabilities may be exploited by attackers (or other malicious processes) to run malicious code

Untrusted Code

- Or, may directly be malicious: may use
  - Obfuscation
    - Code obfuscation
    - Anti-analysis techniques
    - Use of vulnerabilities to hide behavior
  - (Behavioral) evasion
    - Actively subvert enforcement mechanisms
- Security is still defined in terms of policies
  - But enforcement mechanisms need to be stronger in order to defeat a strong adversary

Reference Monitors

- Security policies can be enforced by reference monitors (RM)
  - Key requirements
    - Complete mediation
    - (If interaction with user is needed) Trusted path
- With benign code, we typically assume that it won't actively evade enforcement mechanisms
  - We can possibly maintain security even if there are ways to subvert the checks made by the RM

Types of Reference Monitors

- External RM
  - RM resides outside the address space of untrusted process
  - Relies on memory protection
    - Protect RM's data from untrusted code
    - Limit access to RM's code
- Inline RM
  - Policy enforcement code runs within the address space of the untrusted process
  - Cannot rely on traditional hardware-based memory protection

External Reference Monitors

- System-call based RMs
- Linux Security Modules (LSM)
- AppArmor

System-call based RMs

- OSes already implement RMs to enforce OS security policies
  - Most aspects of policy are configured (e.g., file permissions), while the RM mainly includes mechanisms to enforce these policies
- But these are typically not flexible enough or customizable

System-call based RMs

- More powerful and flexible policies may be realized using a customized RM
- System-calls provide a natural interface at which such a customized RM can reside and mediate requests.

Why Monitor System Calls?

- Complete mediation: All security-relevant actions of processes are administered through this interface
- Performance: Associated with a context-switch --- can be exploited to protect RM without extra overheads
- Granularity
  - Finer granularity than typical access control primitives
  - But coarse enough to be tractable: a few hundred system calls

Why Monitor System Calls?

- Expressiveness
  - Clearly defined, semantically meaningful, well-understood and welldocumented interface (except for some OSes like Windows)
  - Orthogonal (each system call provides a function that is independent ofother system calls --- functions that rarely, if ever, overlap)
  - Can control operations for which OS access controls are ineffective, e.g., loading modules
    - A large number of security-critical operations are traditionally lumped into "administrative privilege"

Why Monitor System Calls?

- Portability: System call policies can be easily ported across similar OSes, e.g., various flavors of UNIX

Some drawbacks of system calls

- Interface is designed for functionality
  - Several syscalls may be equivalent for security purposes, but we a syscall policy needs to treat them separately
- Not all relevant operations are visible
  - For instance, syscall policies cannot control name-to-file translations

Some drawbacks of system calls

- Race conditions
  - Pathname based policies are prone to race conditions
  - More generally, there may be TOCTTOU races relating to system call arguments
    - Unless the argument data is first copied into RM, checked, and then this checked copy is used by the system call
      - Adds more complexity
- The window for exploiting TOCTTOU attacks can be increased by using a large sequence of symbolic links in the name

Linux Security Module Framework

- Motivated by the drawbacks of syscall monitors
  - Defines a number of "hooks" within Linux kernel
    - Includes all points where security checks need to be done
    - RMs can register to be invoked at these hooks
    - SELinux, as well as Linux capabilities are implemented using such RMs

Linux Security Module Framework

- Drawbacks
  - The framework has significant complexity --- while it simplifies some things, the increased complexity makes other things hard.
  - Requires a lot of effort to identify the things that need checking, and where all the hooks need to be placed
  - Very closely tied to the implementation details of an OS --- not easily ported to other OSes.

System call interposition approaches

- User-level interception
  - RM resides within a process
    - Library interposition
      - RM resides in the same address space
      - Advantages
        - high performance
        - Potential for intercepting higher level (semantically richer) operations
      - Drawbacks: RM is unprotected, so appropriate only for benign code

System call interposition approaches

- User-level interception
  - RM resides within a process
    - Kernel-supported interposition, with RM residing in another process
      - Advantages: Secure for untrusted code
      - Drawback: High overheads due to context switches
      - Example: ptrace interface on Linux

System call interposition approaches

- Kernel interception
  - The RM resides in the kernel
    - Advantages: high performance, secure for untrusted code
    - Drawbacks:
      - difficult to program
      - requires root privilege
      - Rootkit defense measures pose compatibility issues

Inline Reference Monitoring

- Foundations
  - Software Fault Isolation (SFI)
  - Control-flow Integrity (CFI)
- Case Study
  - Google Native Client (NaCl)

Inline Reference Monitors (IRMs)

- Provide finer granularity
  - "Variable x is always greater than y"
  - Provides much more expressive power
- Very efficient
  - Does not require a context switch
- Key challenge:
  - Protecting IRM from hostile code

Securing RMs in the same space

- Protect RM data used in enforcing policy
  - Software-based fault isolation (SFI)
- Protect RM checks from being bypassed
  - Control-flow integrity (CFI)
- Note
  - For vulnerability defenses (e.g., Stackguard), we implement the checks using an IRM
  - But we don't worry so much about these properties since we are dealing with benign (and not malicious) code

Software Fault Isolation (SFI)

Background

- Fault Isolation
  - What is fault isolation?
    - when "something bad" happens, the negative consequences are limited in scope.
  - Why is it needed?
    - Untrusted plug-ins makes applications unreliable
    - Third-party modules make the OS unreliable
- Hardware based Fault Isolation
  - Isolated Address Space
  - RPC interfaces for cross boundary communication

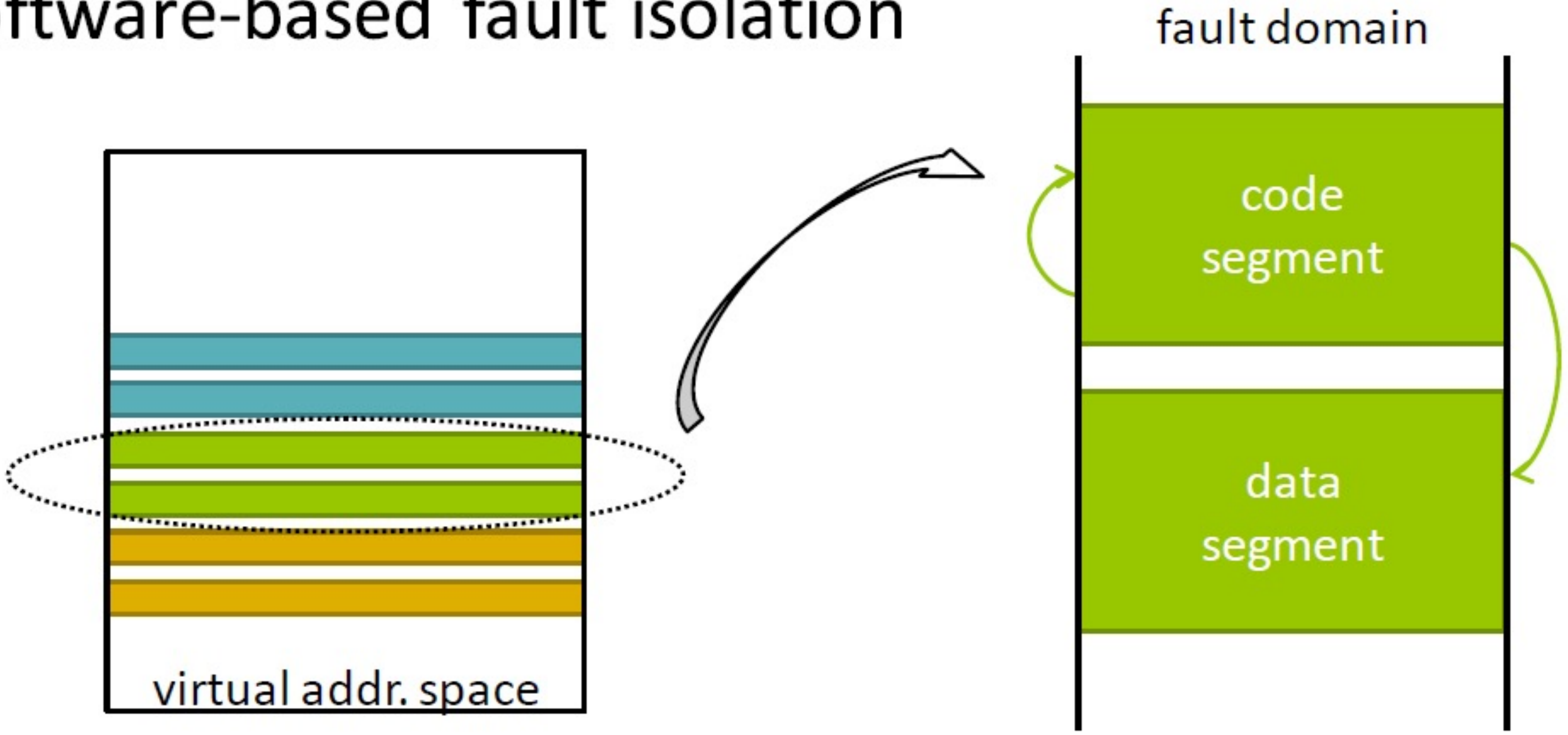
SFI [Wahbe et al 1994]

- Motivation
  - Hardware-assisted context-switches are expensive
    - TLB flushing; some caches may require flushing as well
- Key idea
  - Insert inline checks to verify memory address bounds for
    - Data accesses
    - Indirect control-flow transfers (CFT)
      - Direct CFTs can be statically checked

SFI [Wahbe et al 1994]

- Challenges
  - Efficiency
    - each memory access has the overhead of checking
  - Security
    - Preventing circumvention or subversion of checks

software-based fault isolation



- Even when running in the same virtual address space, limit some code components to access only a part of the address space
  - This subspace is called a "fault domain"

Software Fault Isolation

- Virtual address segments
  - Fault domain (guest) has two segments, one for code, the other for data.
  - Each segment share a unique upper bits (segment identifier)
  - Untrusted module can ONLY jump to or write to the same upper bit pattern (segment identifier)

Software Fault Isolation

- Components of the technique
  - Segment Matching
  - Optimization: instead of checking, simply override the segment bits
    - Originally, the term "sandboxing" referred to this overriding
  - Data sharing
  - Cross-domain Communication

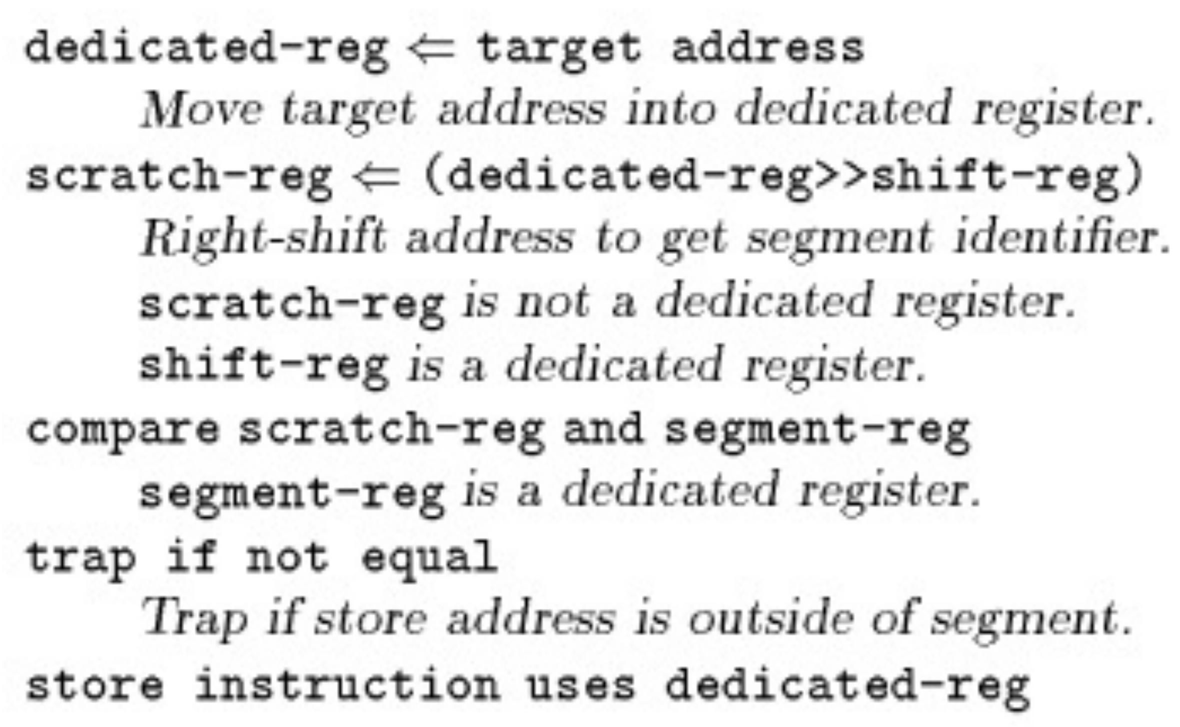
Segment Matching

- Insert checking code before every unsafe instruction
- To prevent subversion of checks, use dedicated registers, and ensure that all jumps and stores use these registers
  - Need only worry about indirect accesses
  - Don't forget that returns are indirect jumps too

Segment Matching

- Checking code determines whether the unsafe instruction has the correct segment identifier
- Trap to a system error routine if checking fails - pinpoint the offending instruction

Segment Matching



```
dedicated-reg ⇐ target address
        Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
        Right-shift address to get segment identifier.
        scratch-reg is not a dedicated register.
        shift-reg is a dedicated register.
compare scratch-reg and segment-reg
        segment-reg is a dedicated register.
trap if not equal
        Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

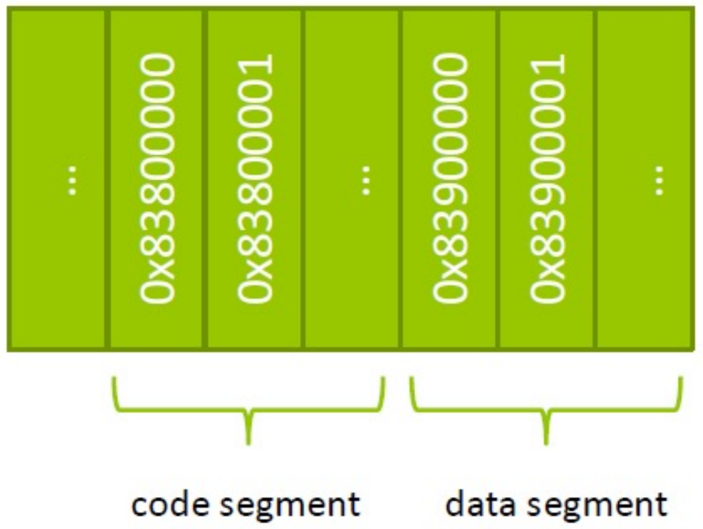
5 instructions, Need 5 dedicated registers (segment value needs to be different for code and data) and it can pinpoint the source of faults. Can reduce the number of registers by hard-coding some values (e.g., number of shift bits).

Optimization 1: Address Sandboxing

- Reduce runtime overhead further compared to segment matching by not pinpointing the offending instruction
- Before each unsafe instruction, inserting codes can set the upper bits of the target address to the correct segment identifier

Address Sandboxing

```
dedicated-reg ⇐ target-reg&and-mask-reg
        Use dedicated register and-mask-reg
        to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
        Use dedicated register segment-reg
        to set segment identifier bits.
store instruction uses dedicated-reg
```



- 3 instructions, Require 5 dedicated registers (since mask and segment registers will be different for code and data)
- Correctness: Relies on the invariant that dedicated registers always contain valid values before any control transfer instruction.

Optimization 2: Guarding pages

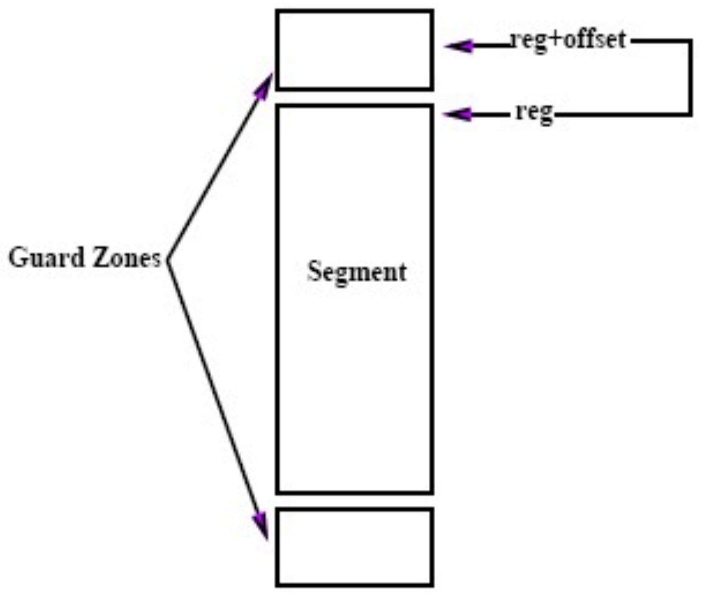


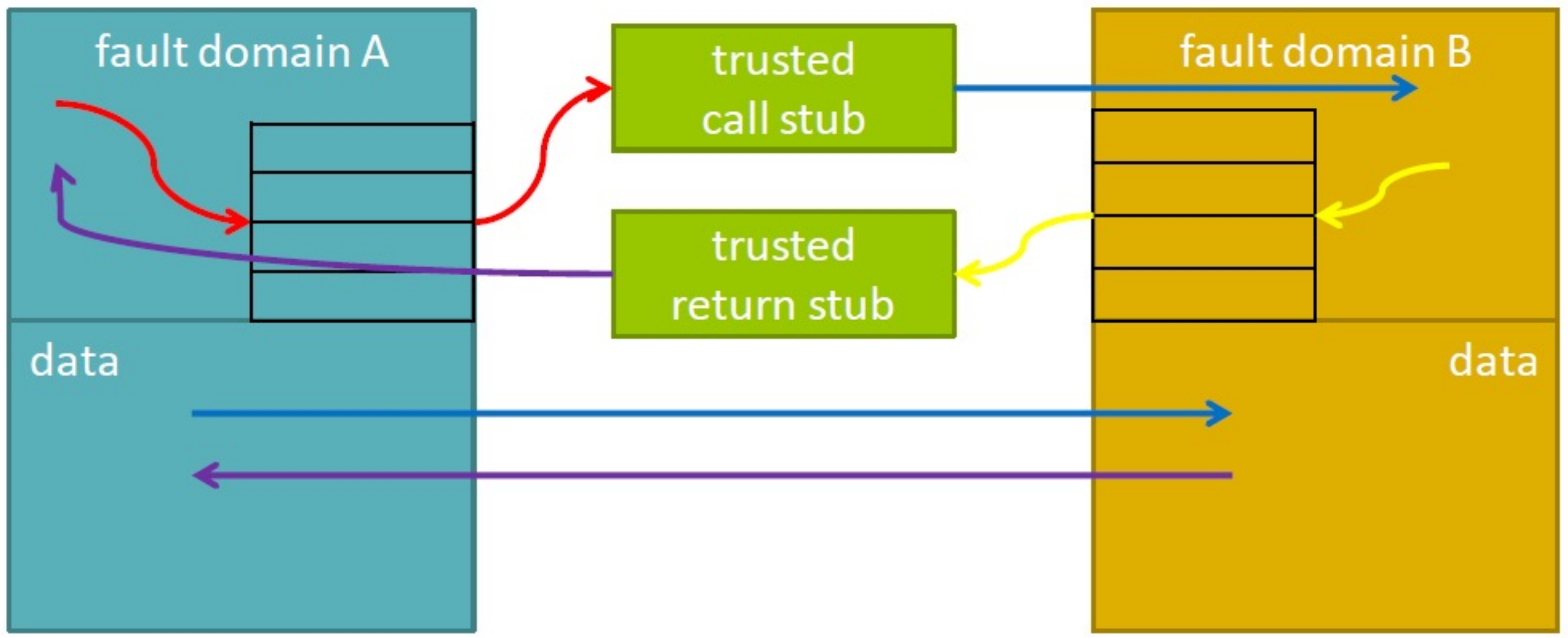
Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.

- A single instruction accesses multiple bytes of memory (4, 8, or may be more)
- Need to check whether all bytes are within the segment
  - Require at least two checks!
- Optimization
  - Sandboxing reg, ignore reg+offset
  - Guard zones ensure that reg+offset will also be in bounds (or that there will be a hardware fault)

Data sharing

- Read-only sharing can be achieved in several ways:
  - Option 1: Don't restrict read acceses
  - Option 2: Allow reads to access some segments other than that of untrusted code
  - Option 3: Remap shared memory into the address space of both the untrusted and trusted domains
- Read-write sharing can use similar techniques.

cross fault domain communication



- trusted stubs to handle RPC
  - for each pair of fault domains
  - stub: copy arguments, re/store registers, switch the exe. stack, validate dedicated regs but! no traps or address space switching (thus, cheaper than HW RPC)
- jump tables to transfer control
  - consists of jump instructions of which target address is legal, outside the domain

SFI details (continued)

- Need compiler assistance
  - To set aside dedicated registers
  - But we cannot trust the compiler
    - Programs may be distributed as binaries, and we can't trust the compiler used to compile that untrusted binary

SFI details (continued)

- Need a verifier
  - Verification is quite simple
    - Dedicated registers should be loaded only after address-sandboxing operations
    - All direct memory accesses and direct jumps should stay within untrusted domain. Implementation operates on binary code
      - Note that SFI checks all indirect accesses and control-transfers at runtime
  - Was implemented on RISC architectures

SFI details (continued)

- Precursor to proof-carrying code [Necula et al]
  - Code producer provides the proof, consumer needs to check it.
    - Proof-checking is much easier than proof generation
    - Especially in an automated verification setting:
      - producer needs to navigate a humongous search space to construct a proof tree
      - consumer needs to just verify that the particular tree provided is valid

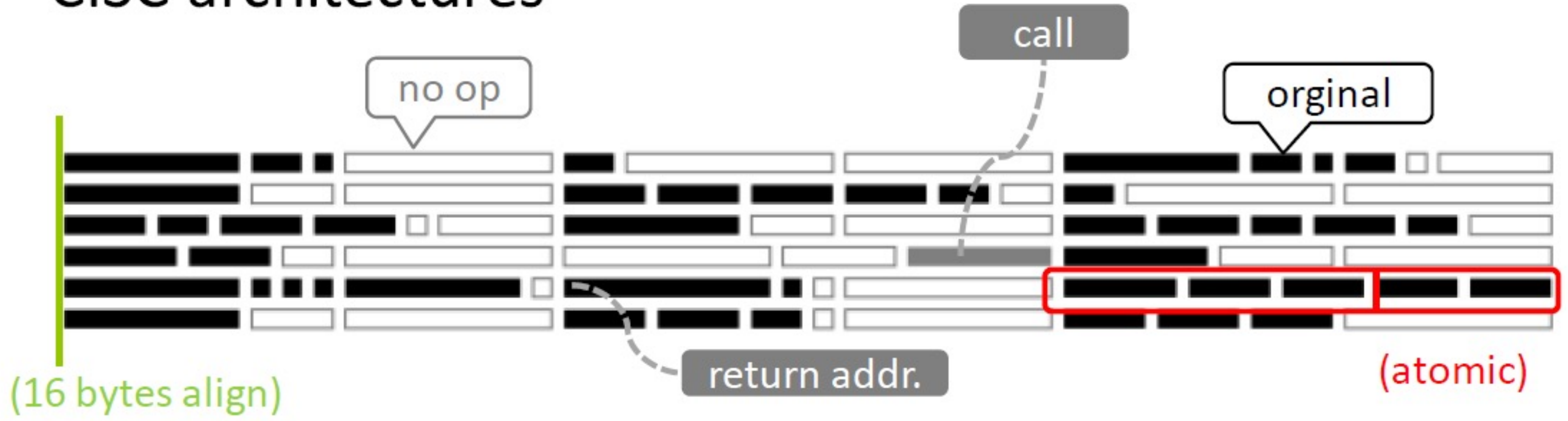
SFI for CISC Architectures (x86)

- Difficulties of bringing SFI to CISC
  - Problem 1: Variable-length instructions
    - What happens if code jumps to the middle of an instruction

SFI for CISC Architectures (x86)

- Difficulties of bringing SFI to CISC
  - Problem 2: Insufficient registers
    - SFI requires 5 dedicated registers for segment matching
    - SFI requires 5 dedicated registers for address sandboxing
    - x86 has very few general-purpose registers available
      - eax, ebx, ecx, edx, esi, edi
    - PittsSFIeld: uses ebx as a dedicated register AND treats esp and ebp as sandboxed registers (adds needed checks)

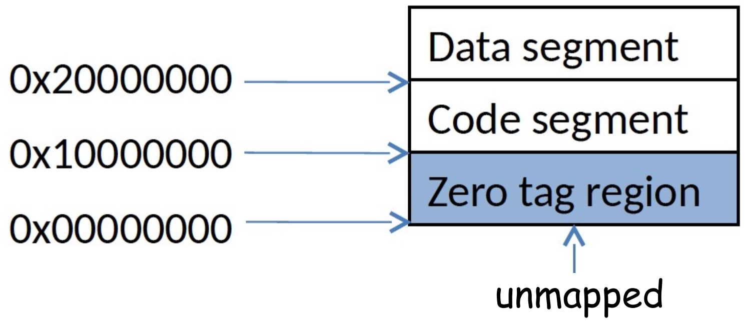
CISC architectures



- padding with no-ops to enforce alignment constraints (power of two)
  - because CISC architectures allow various instruction streams, which makes SFI harder
- `call` placed at the end of chunks
  - because the next addresses are targets of returns
  - they also have <u>low 4 bits zero</u> due to 16 bytes align
- put unsafe operation and its corresponding check together in a chunk
  - atomic, i.e. unsafe op. must be followed by check; no dedicated registers required

Solution to Problem 2

- Hardcode segments
  - Avoids need for segment registers etc.



- Make code and data segments adjacent, and differ by only one bit in their addresses
  - Sandboxing now achieved using a single instruction
    - and 0x20ffffff, %ebx
    - Store using ebx
  - For indirect jumps, use:
    - and 0x10fffff0, %ebx
    - Jump using ebx
- Alternative approach
  - Use x86 segment (CS, DS, ES) registers!
    - Very efficient but not available on x86_64

Control Flow Integrity (CFI)

Control-flow Integrity (CFI) [Abadi et al]

- Unrestricted control-flow transfers (CFTs) can subvert the IRM
  - Simply jump past checks, or
  - Jump into IRM code that updates critical IRM data

Control-flow Integrity (CFI) [Abadi et al]

- Approaches
  - Compute a control-flow graph using static analysis, enforce it at runtime
    - Benefits: With accurate static analysis, can closely constrain CFTs.
    - Drawback: Requires reasoning about targets of indirect CFTs (hard!)
  - Enforce coarse-grained CFI properties
    - All calls should go to beginning of functions
    - All returns should go to instructions following calls
    - No control flow transfers can target instructions belonging to IRM

CFI (Continued)

- Coarse-grained version is sufficient to protect IRM
  - Like SFI, CFI is self-protecting
    - CFI checks the targets of jump, so it can prevent unsafe CFTs that attempt to jump just beyond CFI checks
    - In PittSFIeld, this was achieved by ensuring that the check and access operations were within the same bundle
      - Jumps can only go to the beginning of a bundle, so you can't jump between check and use

CFI (Continued)

- Coarse-grained version is sufficient to protect IRM
  - Because of this, SFI and CFI provide a foundation for securing untrusted code using inline checks.
  - CFI can also be applied to protect against control-flow hijack attacks
    - Jump to injected code (easy)
    - Return to libc (most obvious cases are easy)
    - Return-oriented programming (requires considerable effort to devise ROP attacks that defeat CFI)
    - But not a foolproof defense

CFI (Continued)

- In addition:
  - IRM code shouldn't assume that untrusted code will follow ABI conventions on register use
  - IRM code should use a separate stack
    - To prevent return-to-libc style attacks within IRM code

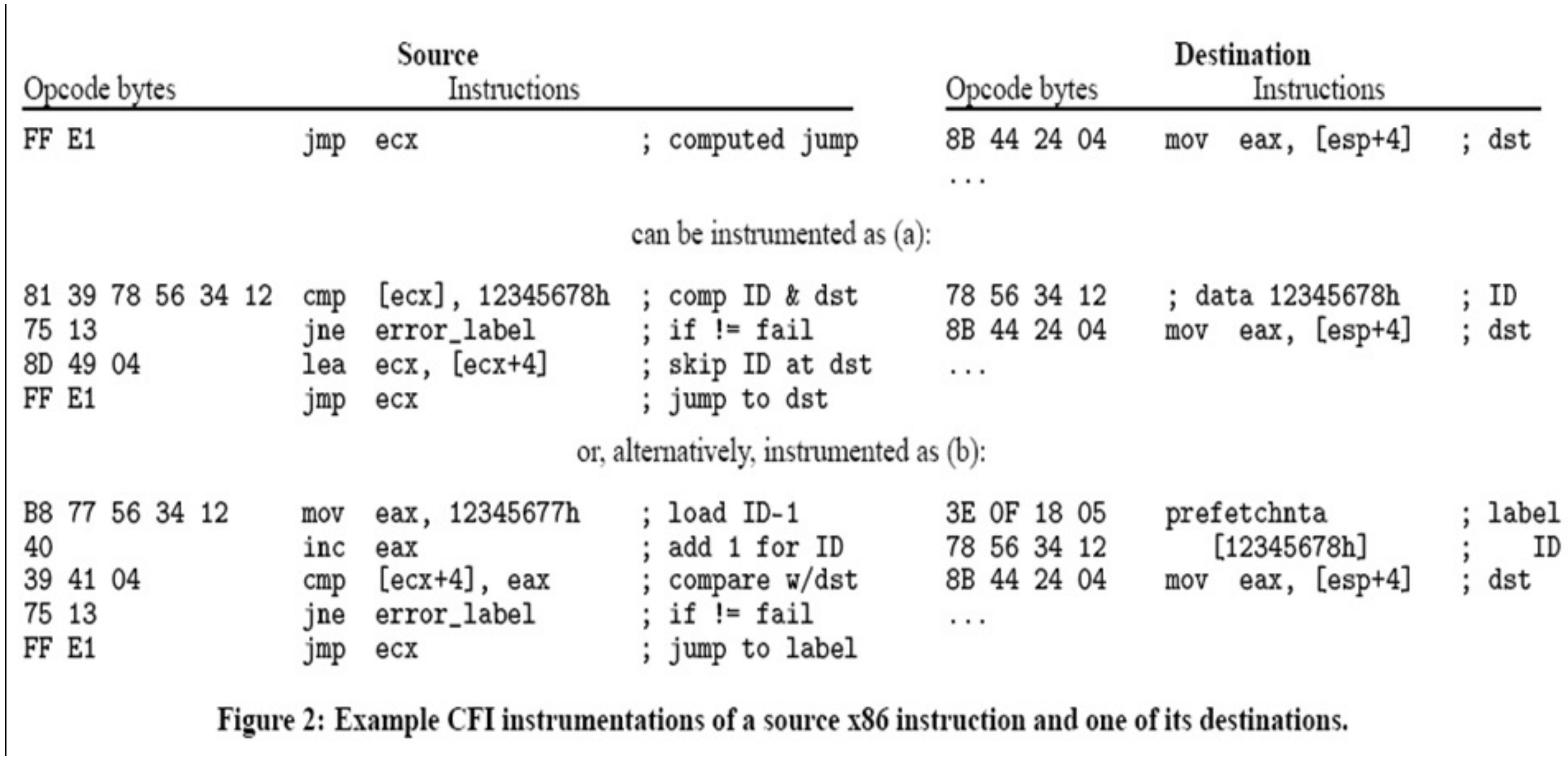
CFI Implementation Strategies

- Approach 1 (proposed in the original CFI paper)
  - Associate a constant index with each CFT target
  - Verify this index before each CFT
  - Ideal for fine-grained approach, where static analysis has computed all potential targets of each indirect CFT instruction

CFI Implementation Strategies

- Approach 1 (proposed in the original CFI paper)
  - Issues
    - If locations L1 and L2 can be targets of an indirect CFT, then both locations should be given the same index
    - If another CFT can go to either L2 or L3, then all three must have same index
    - A particular problem when you consider returns
      - Accuracy can be improved by using a stack, but then you run into the same compatibility issues as stacksmashing defenses that store a second copy of return address

CFI Instrumentation



```
                           Source                                                Destination
Opcode bytes                    Instructions                       Opcode bytes          Instructions
FF E1               jmp  ecx                ; computed jump        8B 44 24 04     mov  eax, [esp+4]    ; dst
                                                                   ...

                                        can be instrumented as (a):

81 39 78 56 34 12   cmp  [ecx], 12345678h   ; comp ID & dst        78 56 34 12     ; data 12345678h     ; ID
75 13               jne  error_label        ; if != fail           8B 44 24 04     mov  eax, [esp+4]    ; dst
8D 49 04            lea  ecx, [ecx+4]       ; skip ID at dst       ...
FF E1               jmp  ecx                ; jump to dst
                                    or, alternatively, instrumented as (b):

B8 77 56 34 12      mov  eax, 12345677h     ; load ID-1            3E 0F 18 05     prefetchnta          ; label
40                  inc  eax                ; add 1 for ID         78 56 34 12        [12345678h]       ;    ID
39 41 04            cmp  [ecx+4], eax       ; compare w/dst        8B 44 24 04     mov  eax, [esp+4]    ; dst
75 13               jne  error_label        ; if != fail           ...
FF E1               jmp  ecx                ; jump to label
```

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

- Method (a): unsafe, since ID is embedded in callsite (could be used by attacker)
- Method (b): safe, but pollute the data cache

CFI Implementation

- CFG construction is conservative
  - Each computed call instruction may go to ANY function whose address is taken (too coarse)
  - Discover those functions by checking relocation entries.
    - Won't work on stripped code

CFI Assumption

- UNQ: Unique IDs.
  - choose longer ID to prevent ensure the uniqueness
  - Otherwise: jump in the middle of a instruction or arbitrary place (in data or code)
- NWC: Non-Writable Code.
  - Code could not be modified. Otherwise, verifier is meaningless, thus all the work is meaningless……
- NXD: Non-Executable Data
  - Otherwise, attacker can execute data that begins with a correct ID.
- All the assumptions should hold. Otherwise, this CFI implementation can be defeated.

CFI Implementation Strategies

- Approach 2
  - Use an array V indexed by address, and holding the following values
    - Function_begin, Valid_return, Valid_target, Invalid
  - A call to target X is permitted if V[X] == Function_begin
  - A return to target X is permitted if V[X] == Valid_return
  - A jump to target X is permitted if V[X] != Invalid
  - Otherwise, CFT is not permitted
    - Note that CFI implementations need only check indirect CFTs

SFI, CFI and Follow-ups

- SFI originally implemented for RISC instruction set, later extended to x86
  - Efficient implementation on x86, x86-64 and ARM architectures have been the focus of recent works
- CFI originally implemented using Microsoft's Phoenix compiler framework
  - Binary instrumentation requires a lot of information unavailable in normal binaries, and hence reliance on specific compiler
  - But the concept has had broad impact

SFI, CFI and Follow-ups

- Google's Native Client (NaCl) project is the most visible application of SFI and CFI techniques
  - Supports untrusted native code in browsers
  - Part of recent WebAssembly standard
    - Included in Firefox 52 and later

Case Study:
Google Native Client (NaCl)

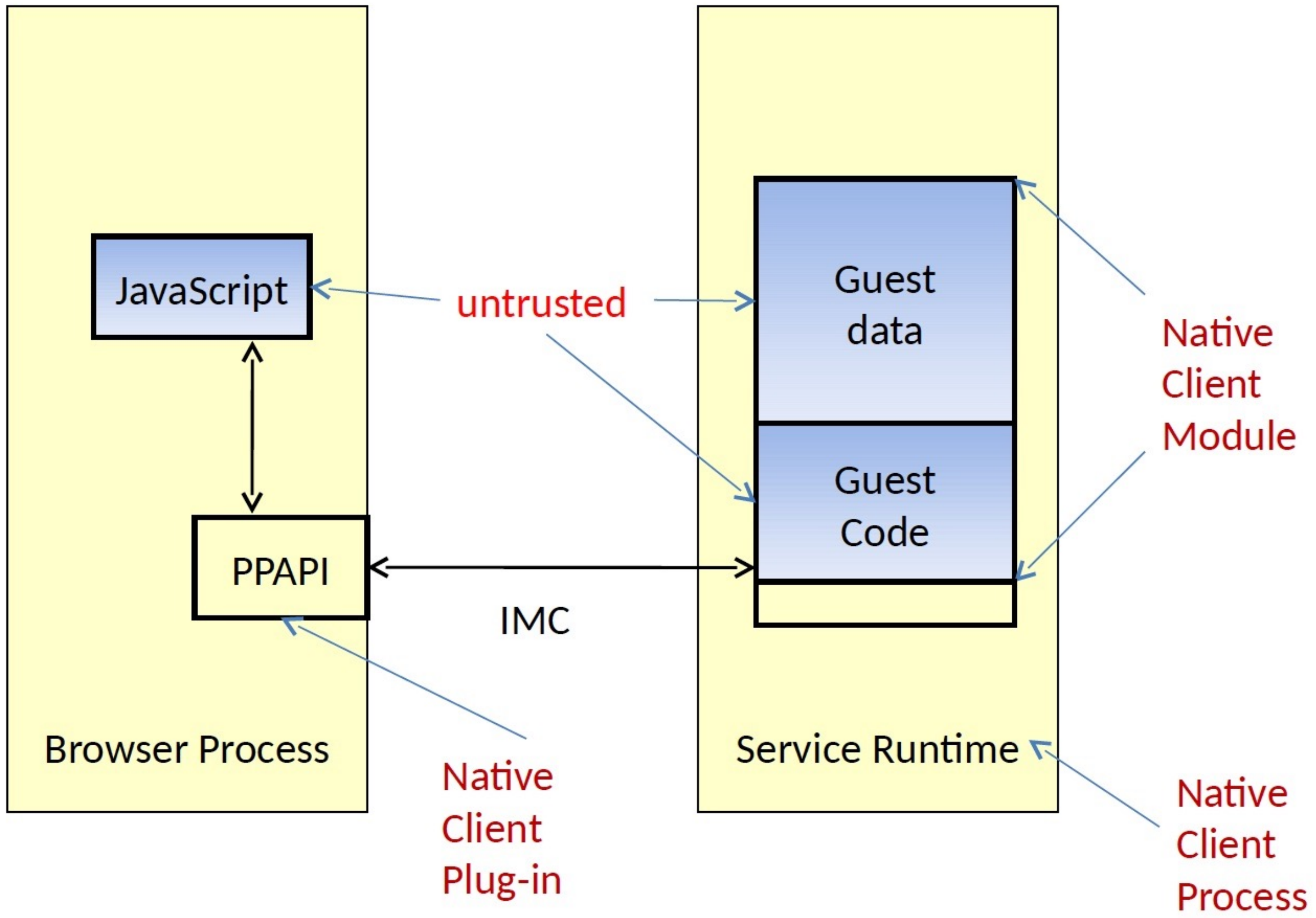
Motivation

- Browsers already allow Javascript code from arbitrary sites, but its performance is inadequate for some applications
  - Games
  - Fluid dynamics (physics simulation)
- Permitting native code from arbitrary sites is too dangerous!

Native Client Approach

- Sandboxed environment for execution of native code. Two parts:
  - SFI using x86 segment as inner sandbox
  - Runtime for allowing safe operations from outer sandbox
- Good runtime facilities
  - Multi-threading support
  - IPC: PPAPI
  - Performance: 5% overhead on average

System Architecture



Design

- Inner Sandbox
  - Static verification to ensure all security properties hold for the untrusted code
  - 32-byte instruction bundles to ensure CFI
  - Trampoline/springboard to allow safe control transfer from untrusted to trusted and vice versa
- Runtime Facilities
  - Safe execution of possible “unsafe” operations
  - Inter module communication: PPAPI & IMC

Binary Constraints & Properties

- Constraints
  - No self modifying code
  - Static linked with a fix start address of text segment
  - All indirect control transfer use nacljmp instruction
  - The binary is padded up to the nearest page with hlt
  - No instructions overlap 32-byte boundary
  - All instructions are reachable by fall-through disassembly from starting address
  - All direct control transfers target valid instructions

Control Flow Integrity

- All control transfers must target an instruction identified during disassembly
- Direct control flow
  - Target should be one of reachable instructions
- Indirect Control flow
  - Segmented support (works because a fix start address)
  - No returns
  - Limit target to 32 byte boundary (nacljmp on the right)

```
jmp eax -> and eax,0xffffffe0
           jmp eax
```

  - Nacljmp is atomic

Data Integrity

- Segmented memory support
- Limited instruction set (no assignment to segment register)
  - i.e. move ds, ax is forbidden

Questions