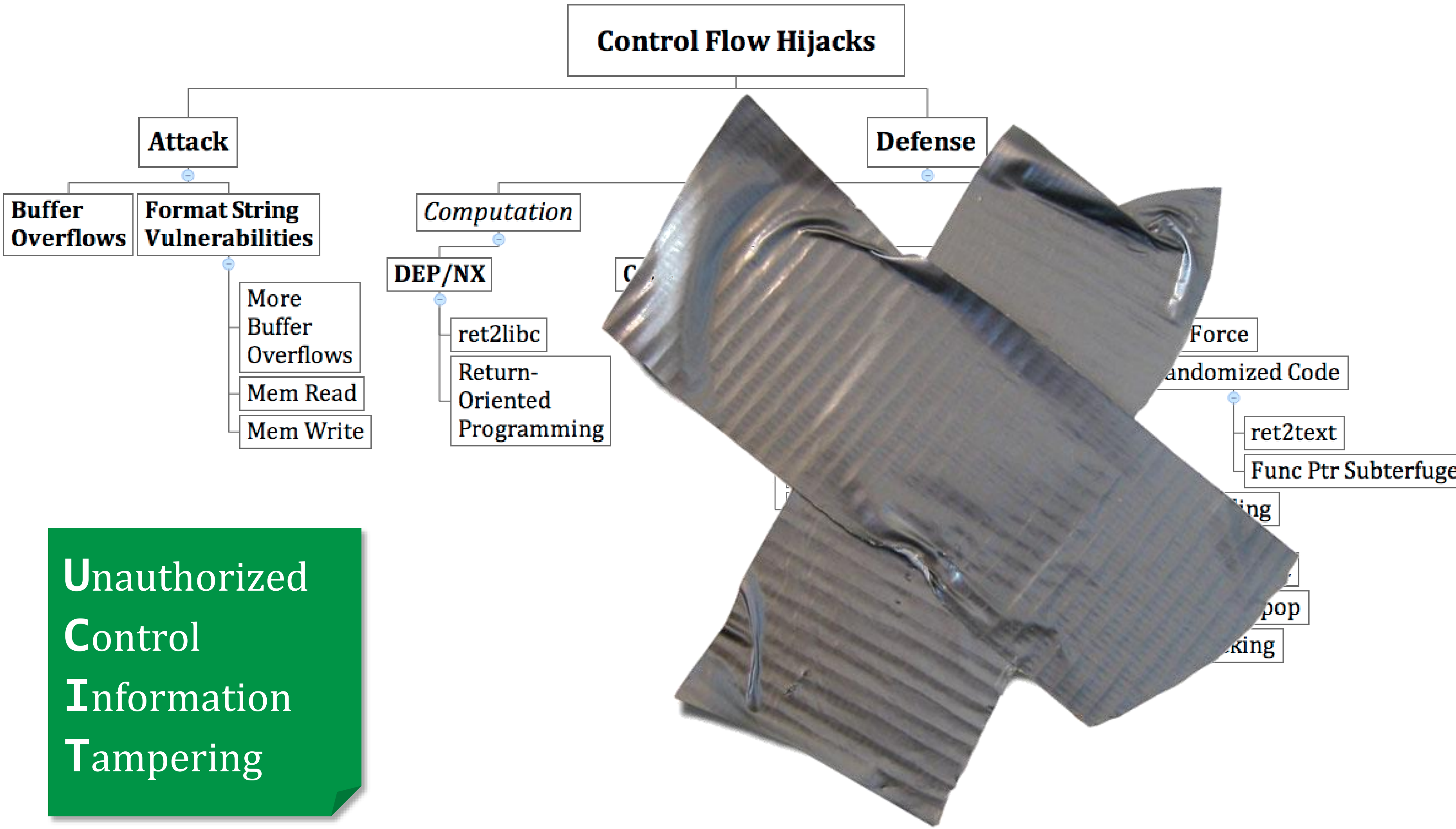
Control Flow Integrity & Software Fault Isolation

David Brumley
Carnegie Mellon University

Our story so far…



Unauthorized
Control
Information
Tampering

http://propercourse.blogspot.com/2010/05/i-believe-in-duct-tape.html

Adversary Model Matters!

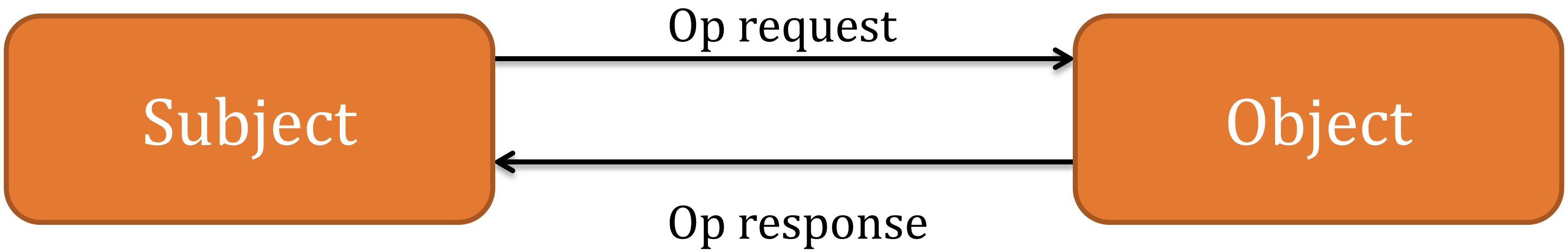
Cowan et al., USENIX Security 1998

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

"Programs compiled with StackGuard are safe from ***buffer overflow attack****, regardless of the software engineering quality of the program."*

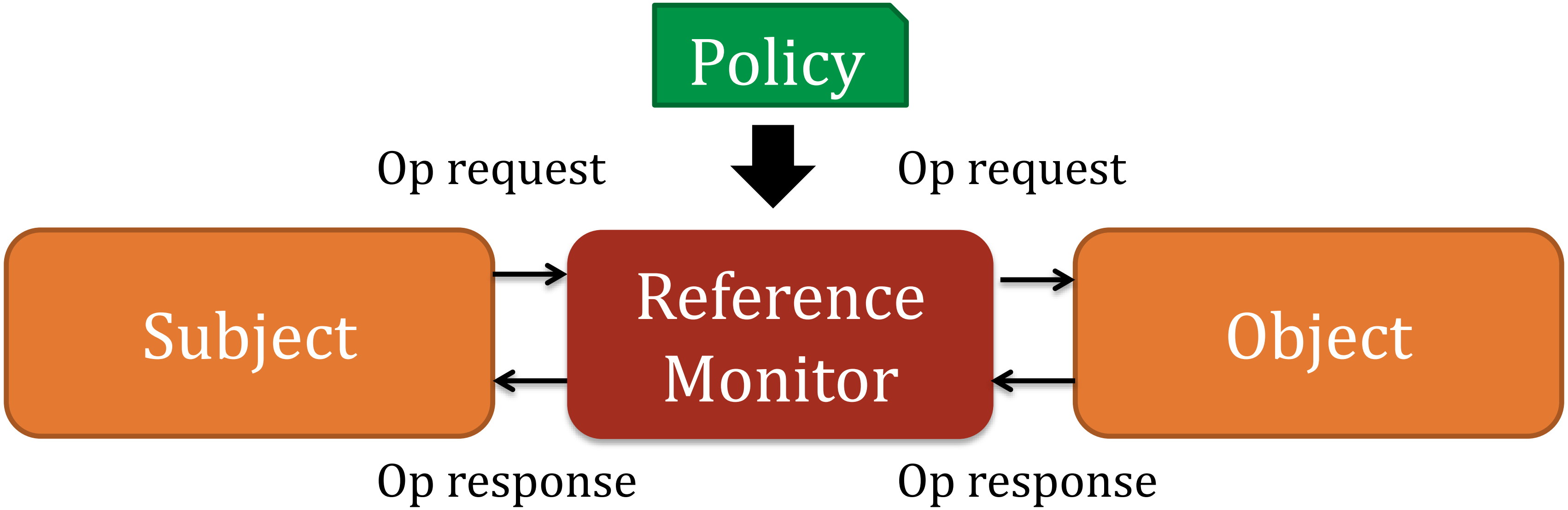
What if the adversary is more powerful?
How powerful is powerful enough?

Reference Monitors



People
Processes
Computer Operations

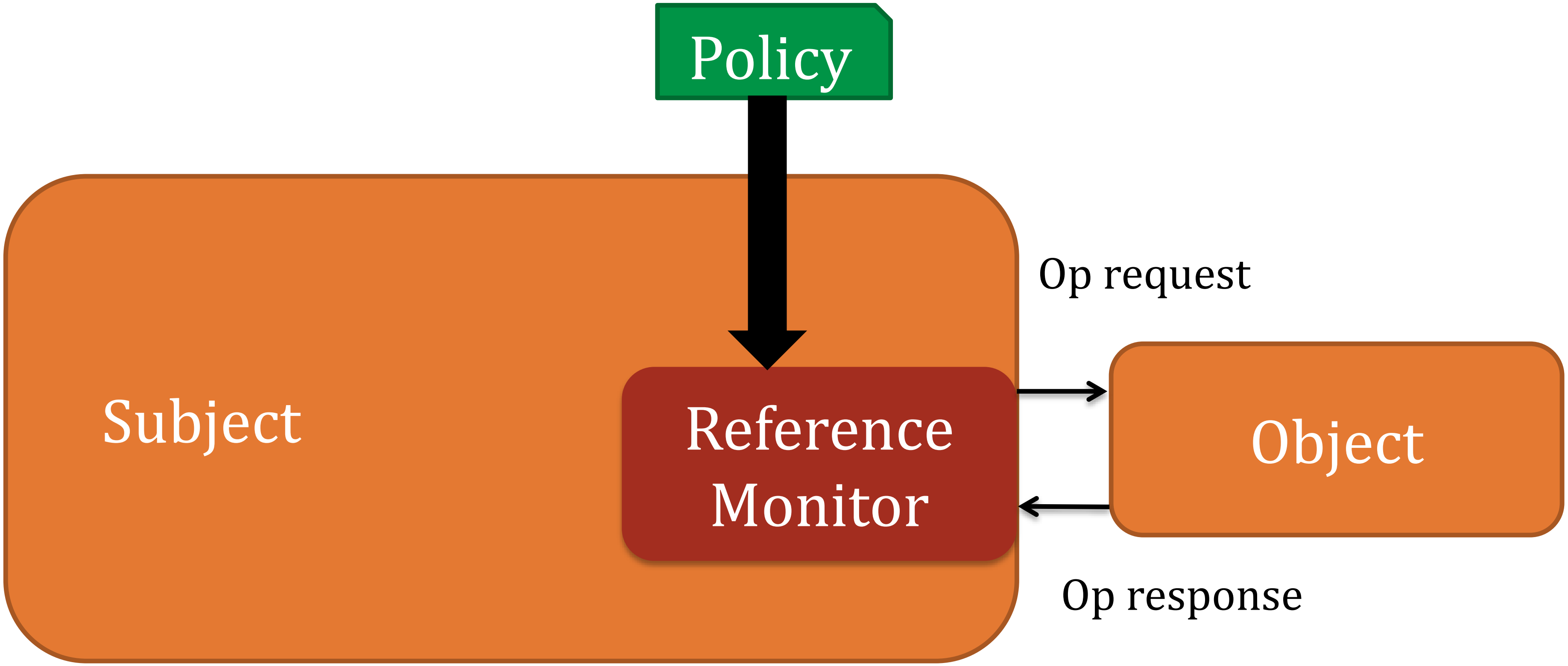
Files
Sockets
Computer Operations



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.

Inlined Referenced Monitor



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

Control Flow Integrity

Assigned Reading:

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

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

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"

Method:

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

Control Flow Graphs

Basic Block

Defn Basic Block: A consecutive sequence of instructions /

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

instructions in the sequence

1. x = y + z
2. z = t + i

3. x = y + z
4. z = t + i
5. jmp 1

6. jmp 3

3 static basic blocks

1. x = y + z
2. z = t + i
3. x = y + z
4. z = t + i
5. jmp 1

1 dynamic basic block

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.

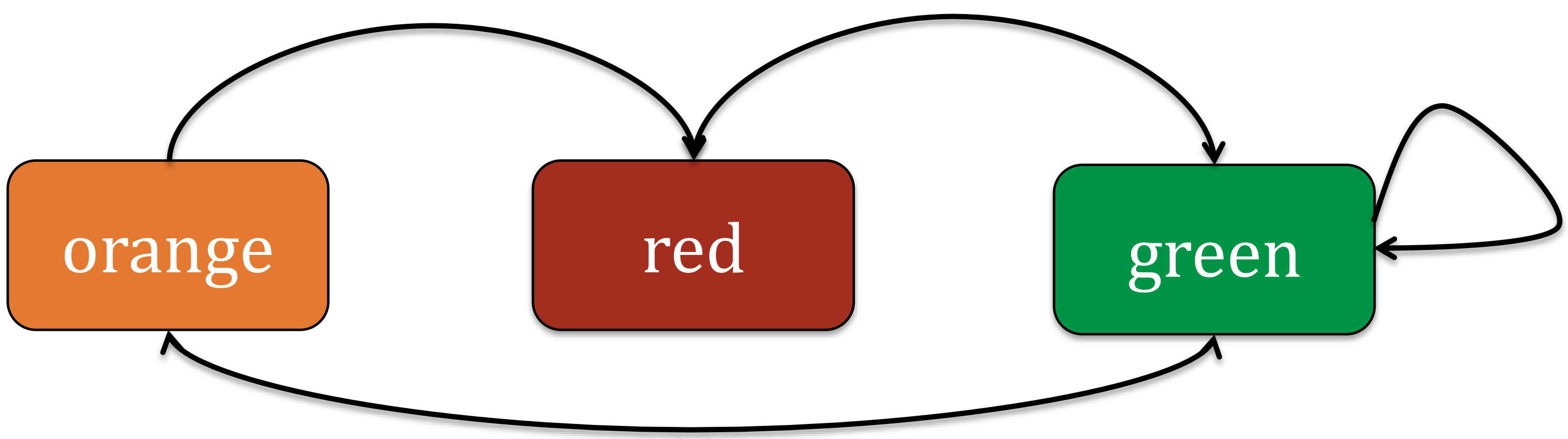
Call Graph

- Nodes are functions. There is an edge (v_i, v_j) if function v_i calls function v_j.

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

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

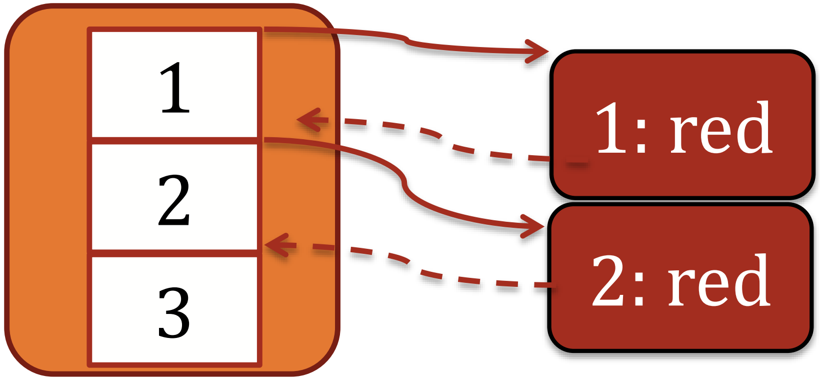
```
void green()
{
  green();
  orange();
}
```



Super Graph

- Superimpose CFGs of all procedures over the call graph

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



A *context sensitive* super-graph for orange lines 1 and 2.

Precision: Sensitive or Insensitive

The more precise the analysis, the more accurate it reflects the “real” program behavior.

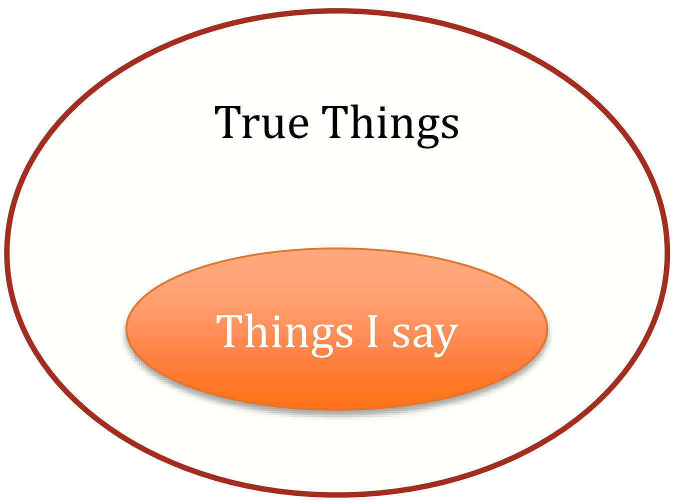
- More precise = more time to compute
- More precise = more space
- Limited by ***soundness/completeness*** tradeoff

Common Terminology in any Static Analysis:

- ***Context*** sensitive vs. context insensitive
- ***Flow*** sensitive vs. flow insensitive
- ***Path*** sensitive vs. path insensitive

Soundness

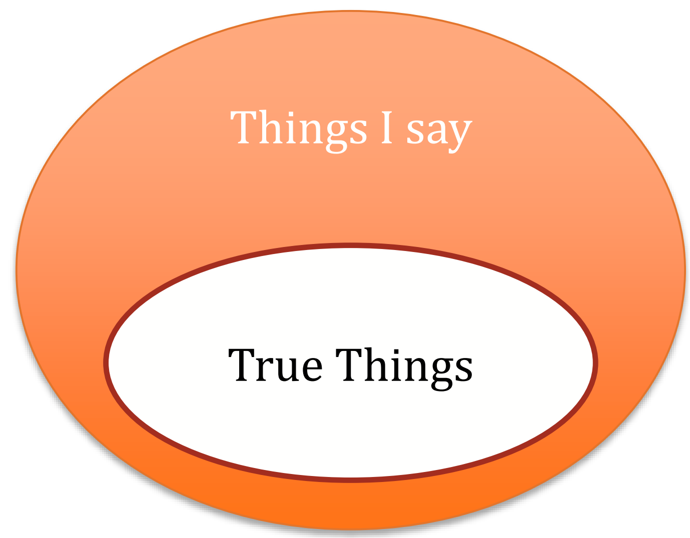
If analysis says X is true, then X is true.



Trivially Sound: Say nothing

Completeness

If X is true, then analysis says X is true.



Trivially complete: Say everything

Sound and Complete: Say exactly the set of true things!

Context Sensitive

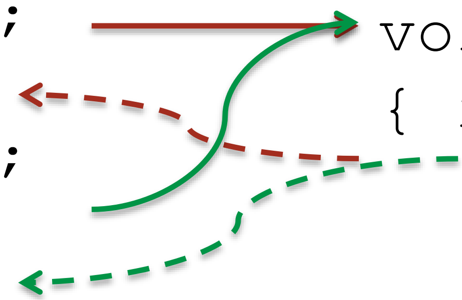
Whether different calling contexts are distinguished

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

Context sensitive distinguishes 2 different calls to red(-)

Context Sensitive Example

```
a = id(4);

b = id(5);
```

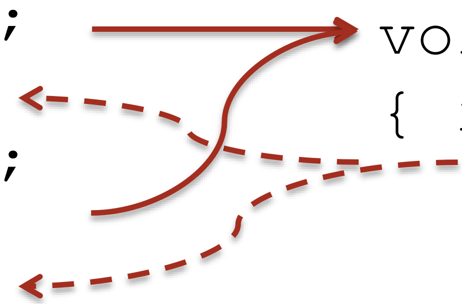


```
void id(int z)
{ return z; }
```

Context-Sensitive
(color denotes matching call/ret)

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

```
a = id(4);

b = id(5);
```



```
void id(int z)
{ return z; }
```

Context-Insensitive
(note merging)

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

Flow Sensitive

- A ***flow*** sensitive analysis considers the order (flow) of statements
  - Flow insensitive = usually linear-type algorithm
  - Flow sensitive = usually at least quadratic (dataflow)
- Examples:
  - Type checking is flow insensitive since a variable has a single type regardless of the order of statements
  - Detecting uninitialized variables requires flow sensitivity

```
x = 4;
....
x = 5;
```

Flow sensitive can distinguish values of x, flow insensitive cannot

Flow Sensitive Example

```
1. x = 4;
....
n. x = 5;
```

Flow sensitive:
x is the constant 4 at line 1,
x is the constant 5 at line n

Flow insensitive:
x is not a constant

Path Sensitive

A path sensitive analysis maintains branch conditions along each ***execution path***

- Requires extreme care to make scalable
- Subsumes flow sensitivity

Path Sensitive Example

```
1. if(x >= 0)
2.     y = x;
3. else
4.     y = -x;
```

path sensitive:
y >= 0 at line 2,
y > 0 at line 4

path insensitive:
y is not a constant

Precision

Even path sensitive analysis approximates behavior due to:

- loops/recursion
- unrealizable paths

```
if(a^n + b^n = c^n && n>2 && a>0 && b>0 && c>0)
    x = 7;
else
    x = 8;
```

Unrealizable path.
x will always be 8

Control Flow Integrity (Analysis)

CFI Overview

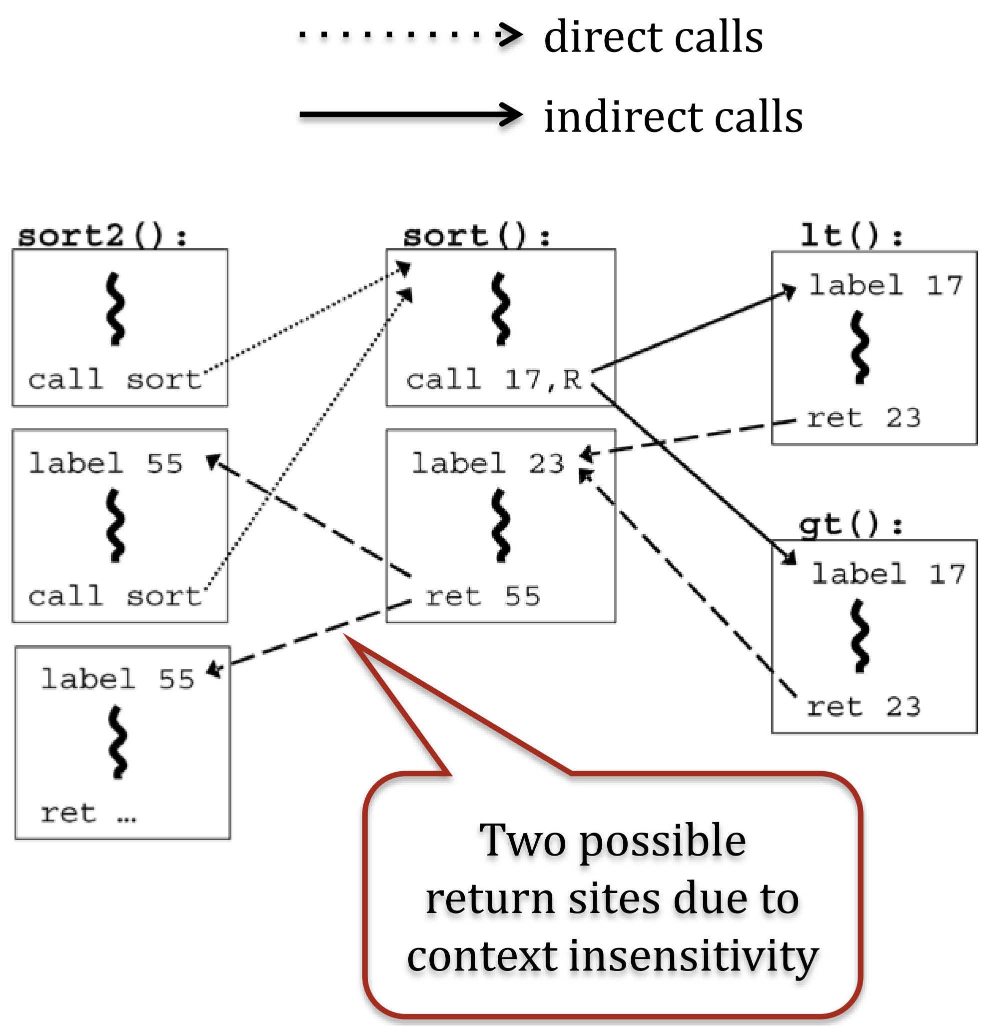
Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

Method:

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

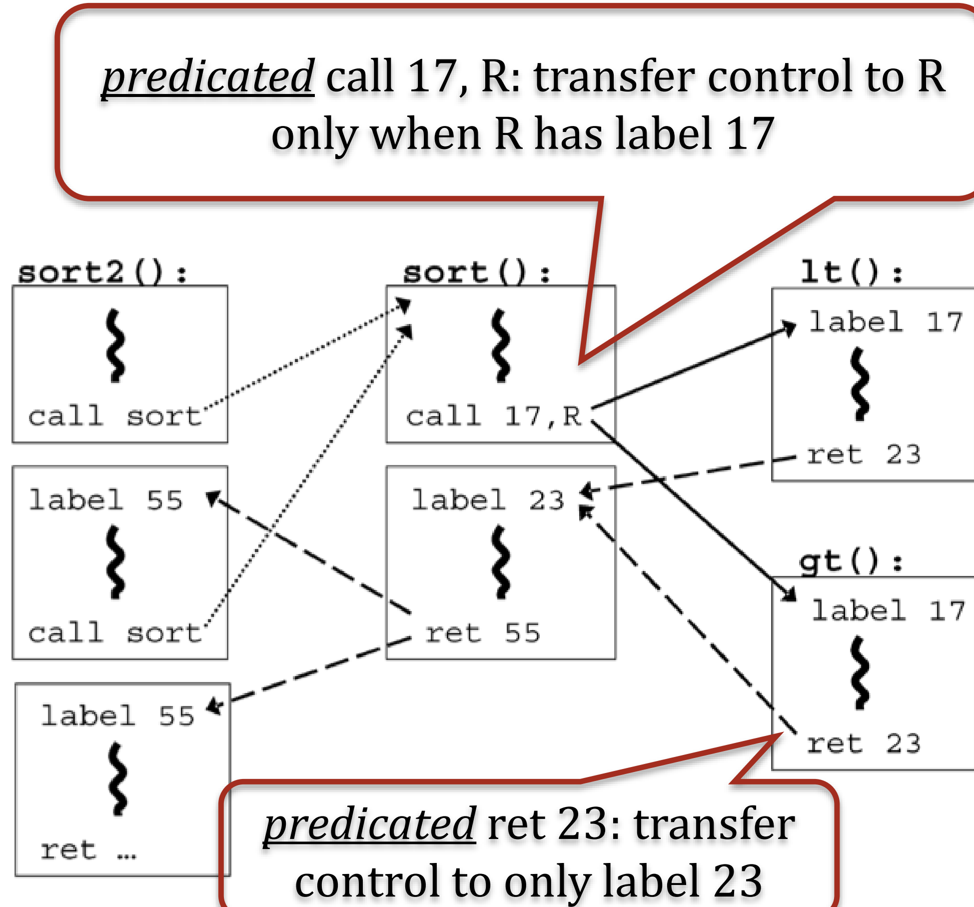
Build CFG

```
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

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

What about indirect jumps and ret?

ID Checks

Check dest label

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

is instrumented using `prefetchnta` destination   , 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
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Check dest label

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return

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

```
8B 0C 24                    mov   ecx, [esp]        ; load  address into register
83 C4 14                    add   esp, 14h          ;     p 20 bytes off the stack
3E 81 79 04 DD CC BB AA     cmp   [ecx+4], AABBCCDDh ; compare opcodes at destination
75 13                       jne   error_label       ; if not ID value, then fail
FF E1                       jmp   ecx               ; jump to return address
```

Performance

Size: increase 8% avg

Time: increase 0-45%; 16% avg

– I/O latency helps hide overhead

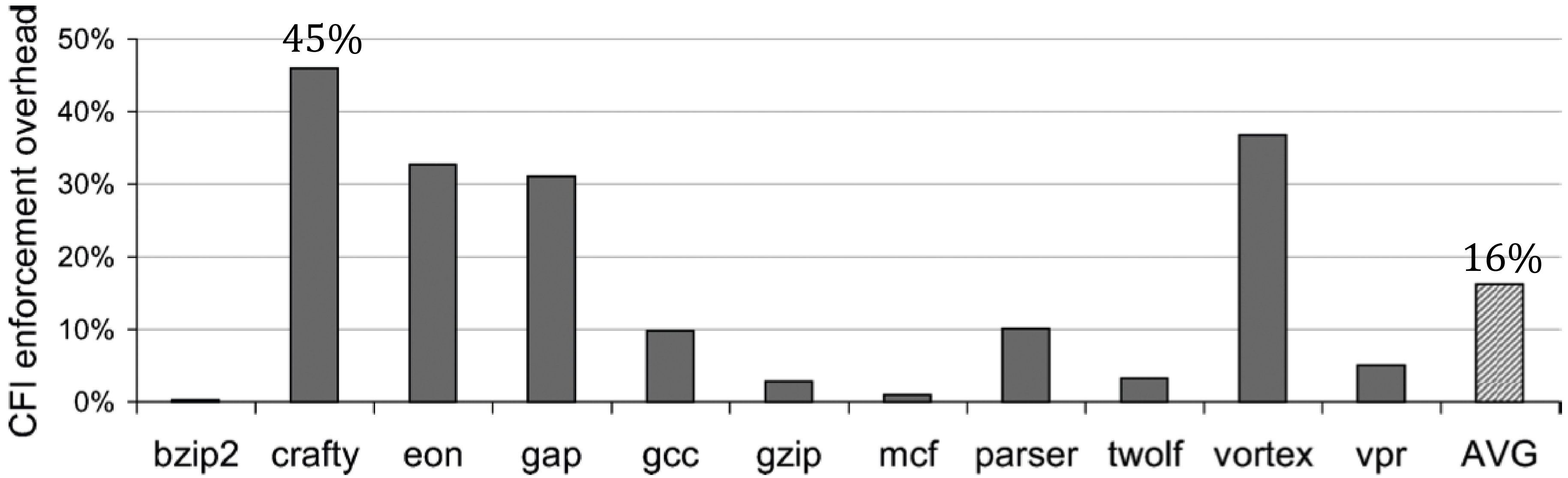


Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

CFI Adversary Model

CAN

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

Assumptions are often vulnerabilities!

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

Let’s check our assumptions!

- **Non-executable Data**
  - let’s inject code with desired ID…
- **Non-writable Code**
  - let’s overwrite the check instructions…
  - can be problematic for JIT compilers
- **Context-Switching Preserves Registers**
  - time-of-check vs. time-of-use
  - **BONUS** point: why don’t we use the `RET` instruction to return?

Time-of-Check vs. Time-of-Use

```
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
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment

```
C2 10 00                ret   10h                  ; return, and pop 16 extra bytes
```

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

```
8B 0C 24                mov   ecx, [esp]
83 C4 14                add   esp, 14h
3E 81 79 04 DD CC BB AA cmp   [ecx+4], AABBCCDDh
75 13                   jne   error_label
FF E1                   jmp   ecx
```

what if there is a context switch here?

Security Guarantees

Effective against attacks based on illegitimate control-flow transfer

– buffer overflow, ret2libc, 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)

Software Fault Isolation

- SFI ensures that a module only accesses memory within its region by adding ***checks***
  - e.g., a plugin can accesses only its own memory

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

SFI Check

- CFI ensures inserted memory checks are executed

Inline Reference Monitors

- IRMs inline a security policy into binary to ensure security enforcement

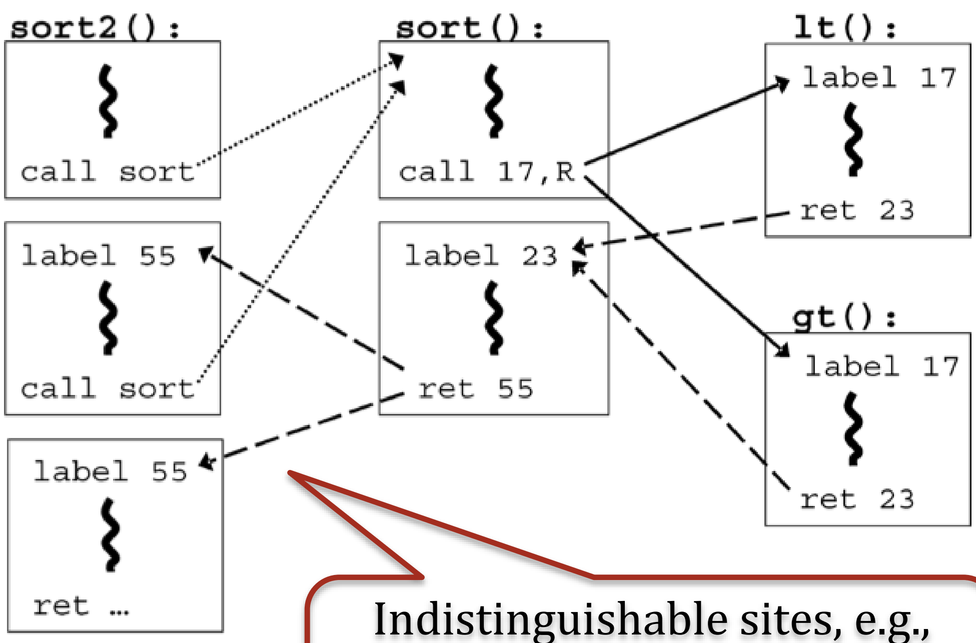
- Any IRM can be supported by CFI + Software Memory Access Control
  - **CFI:** IRM code cannot be circumvented

    +
  - **SMAC:** IRM state cannot be tampered

Accuracy vs. Security

The accuracy of the CFG will reflect the level of enforcement of the security mechanism.

```
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 );
}
```



Indistinguishable sites, e.g., due to lack of context sensitivity will be merged

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?

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

Proof of Security

Theorem (Informal):

Given state S_0 with

- non-writeable, well-instrumented code mem M_0

Then for all runtime steps S_i -> S_{i+1},

- S_{i+1} is one of the allowed successors in the CFG, or
- S_{i+1} is an error state

We can make these sorts of statements precise with ***operational semantics***.

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

Software Fault Isolation

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

Isolation Mechanisms

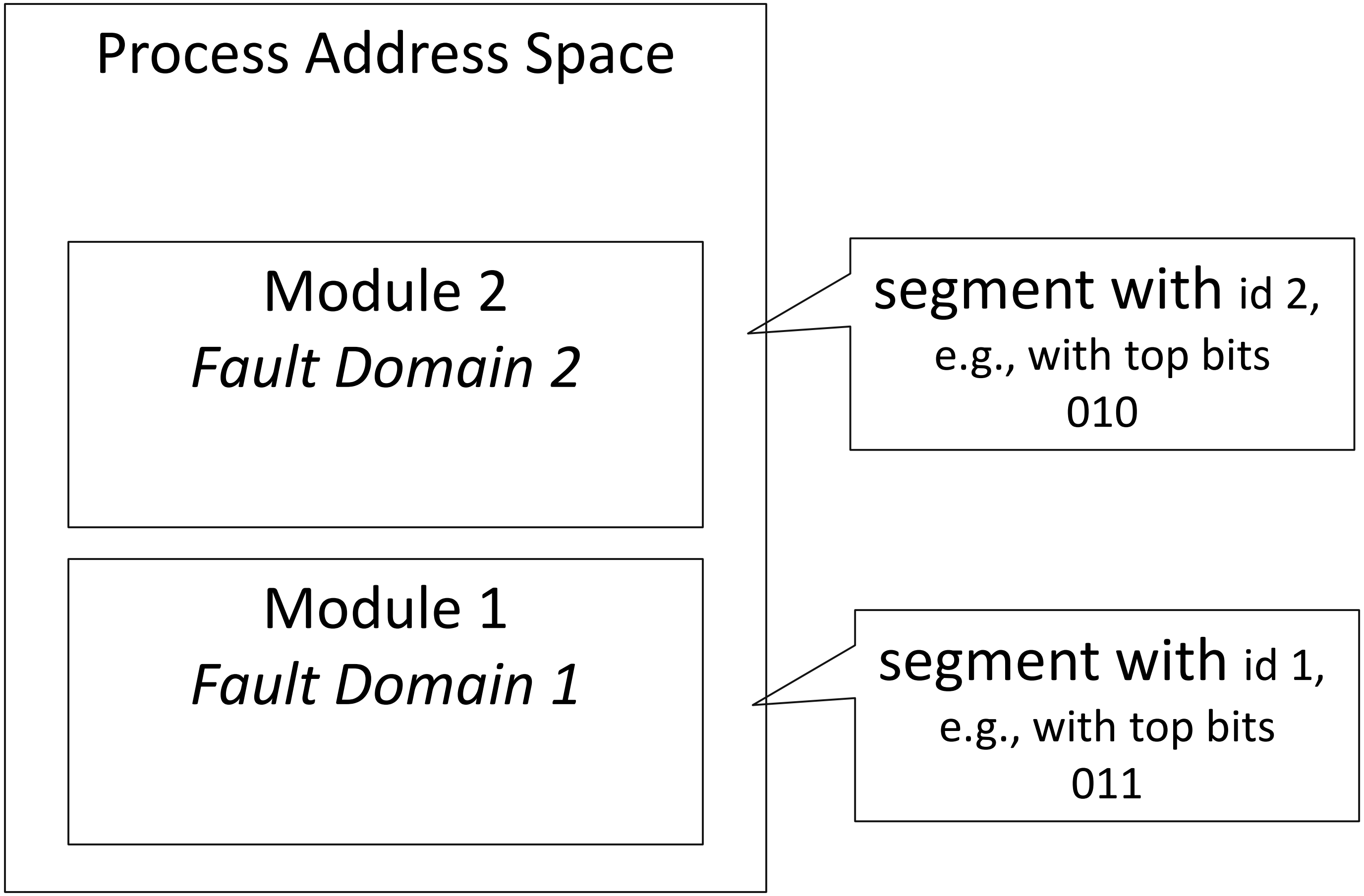
- Hardware
  - Memory Protection (virtual address translation, x86 segmentation)
- Software
  - Sandboxing ← Software Fault Isolation ≈ Memory Protection in Software
  - Language-Based
- Hardware + Software
  - Virtual machines

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

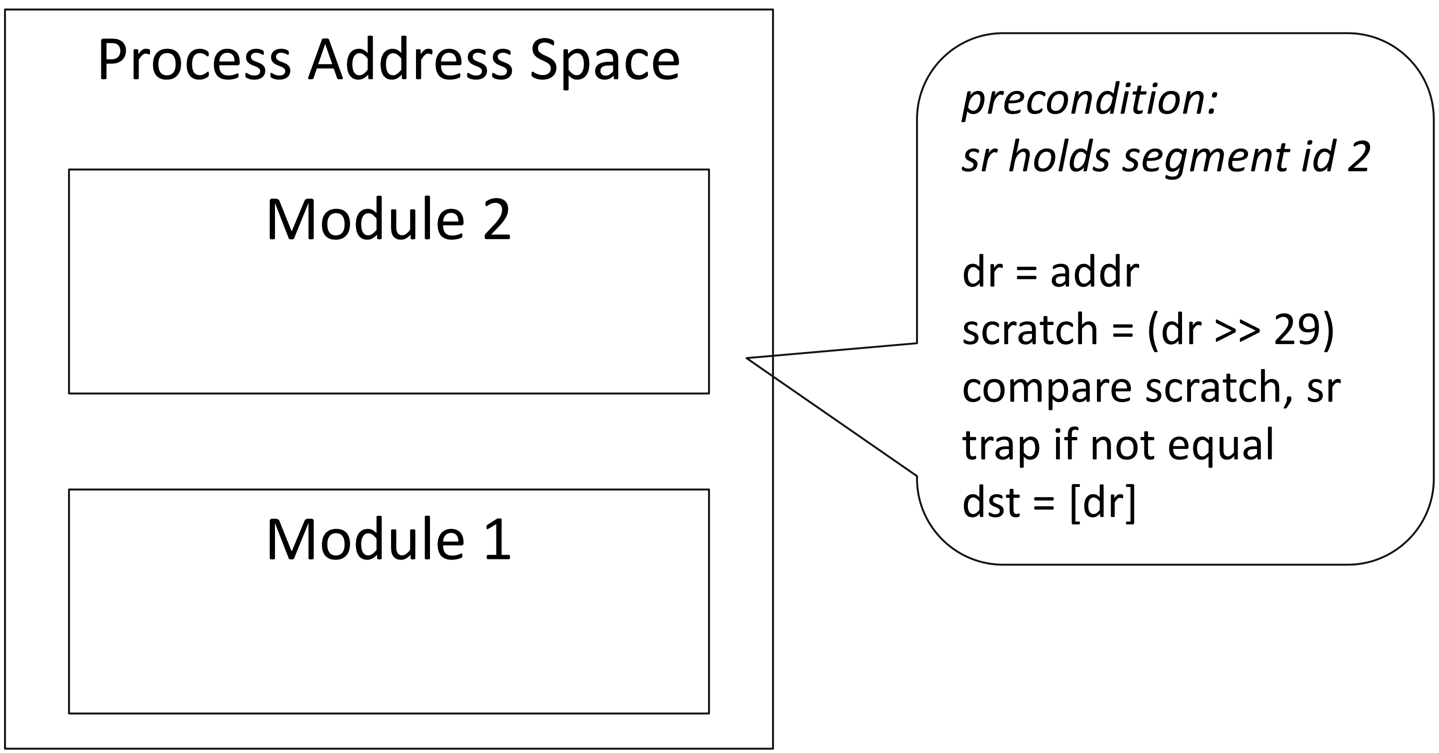
Main Idea



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

Scheme 1: Segment Matching

- *Check* every mem access for matching seg id
- assume dedicated registers segment register (sr) and data register (dr)
  - not available to the program (no big deal in Alpha)



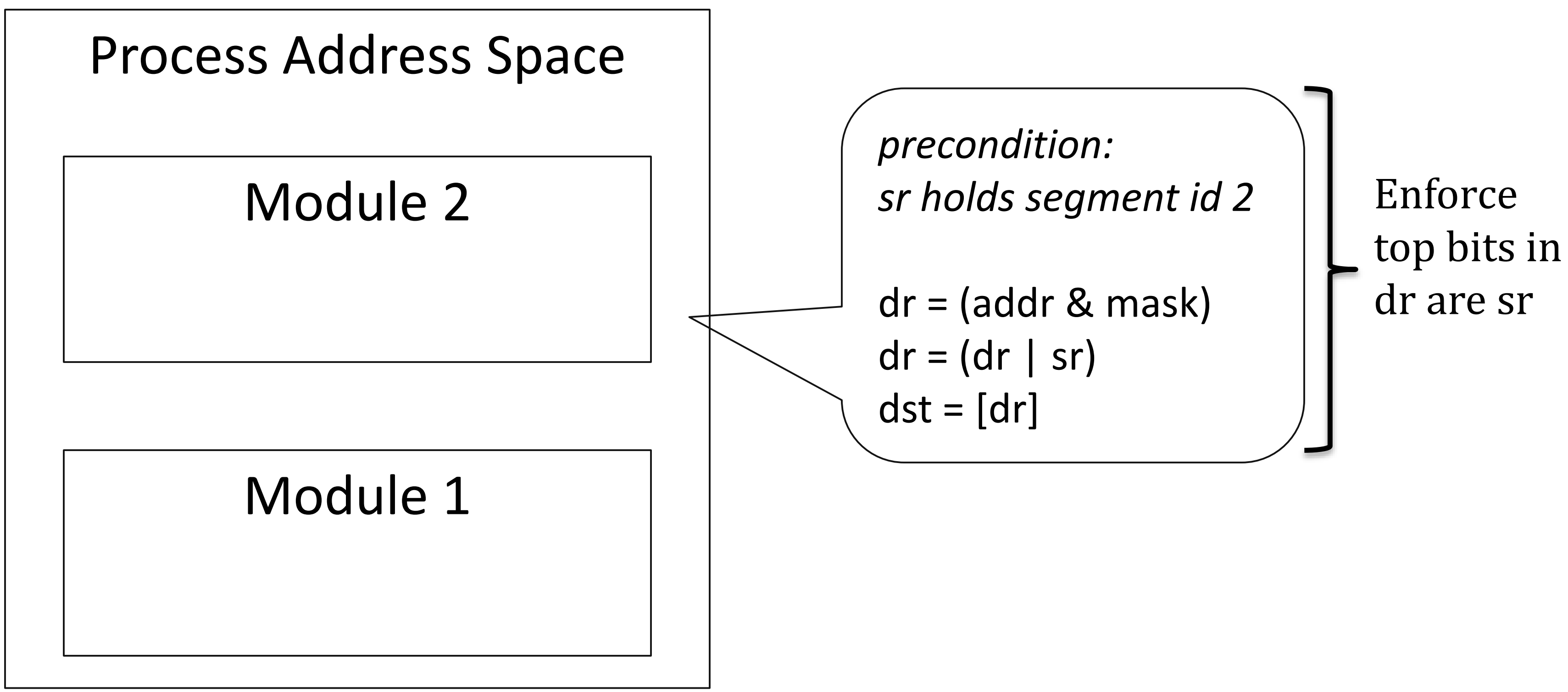
Safety

- 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

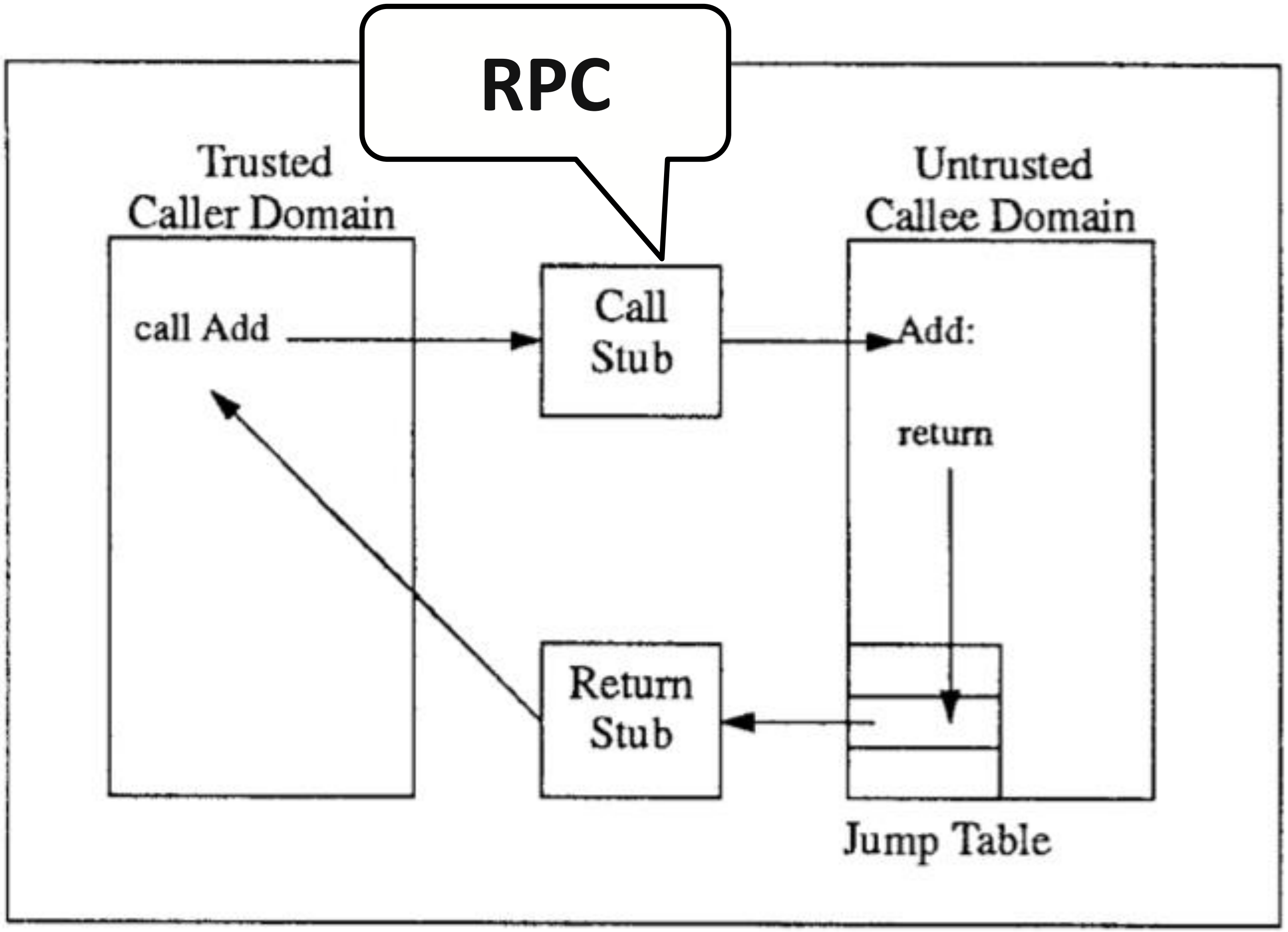
Segment Matching

- 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)

Communication between domains

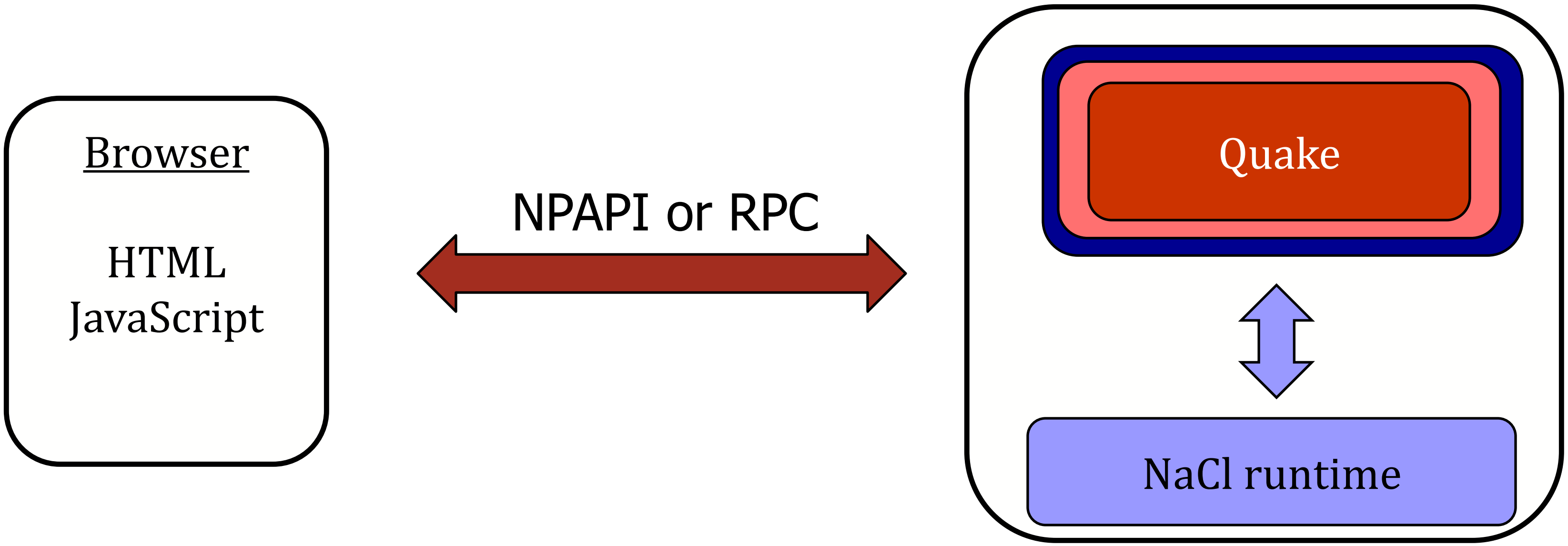


Native Client

Optional Reading:

Native Client: A Sandbox for Portable, Untrusted x86 Native Code
by Yee et al.

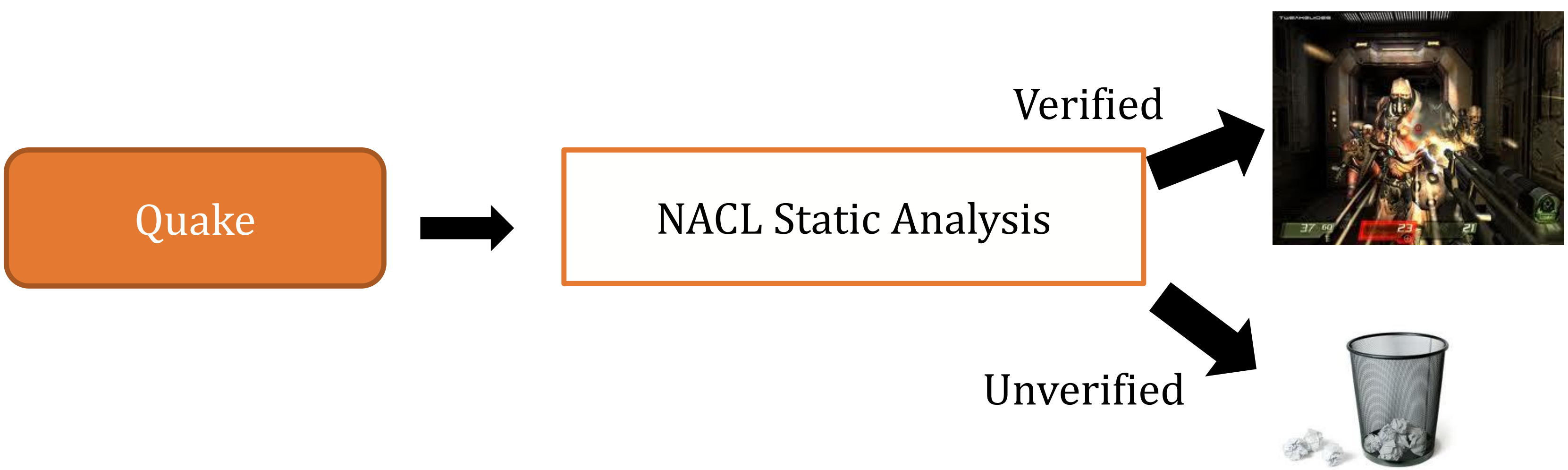
NaCL: A Modern Day Example



- Two sandboxes:
  - an inner sandbox to mediate x86-specific runtime details (using what technique?)
  - an outer sandbox mediates system calls (Using what technique?)

Security Goal

- Achieve comparable safety to accepted systems such as JavaScript.
  - Input: arbitrary code and data
    - support multi-threading, inter-module communication
  - NaCL checks that code conforms to security rules, else refuses to run.



Obligations

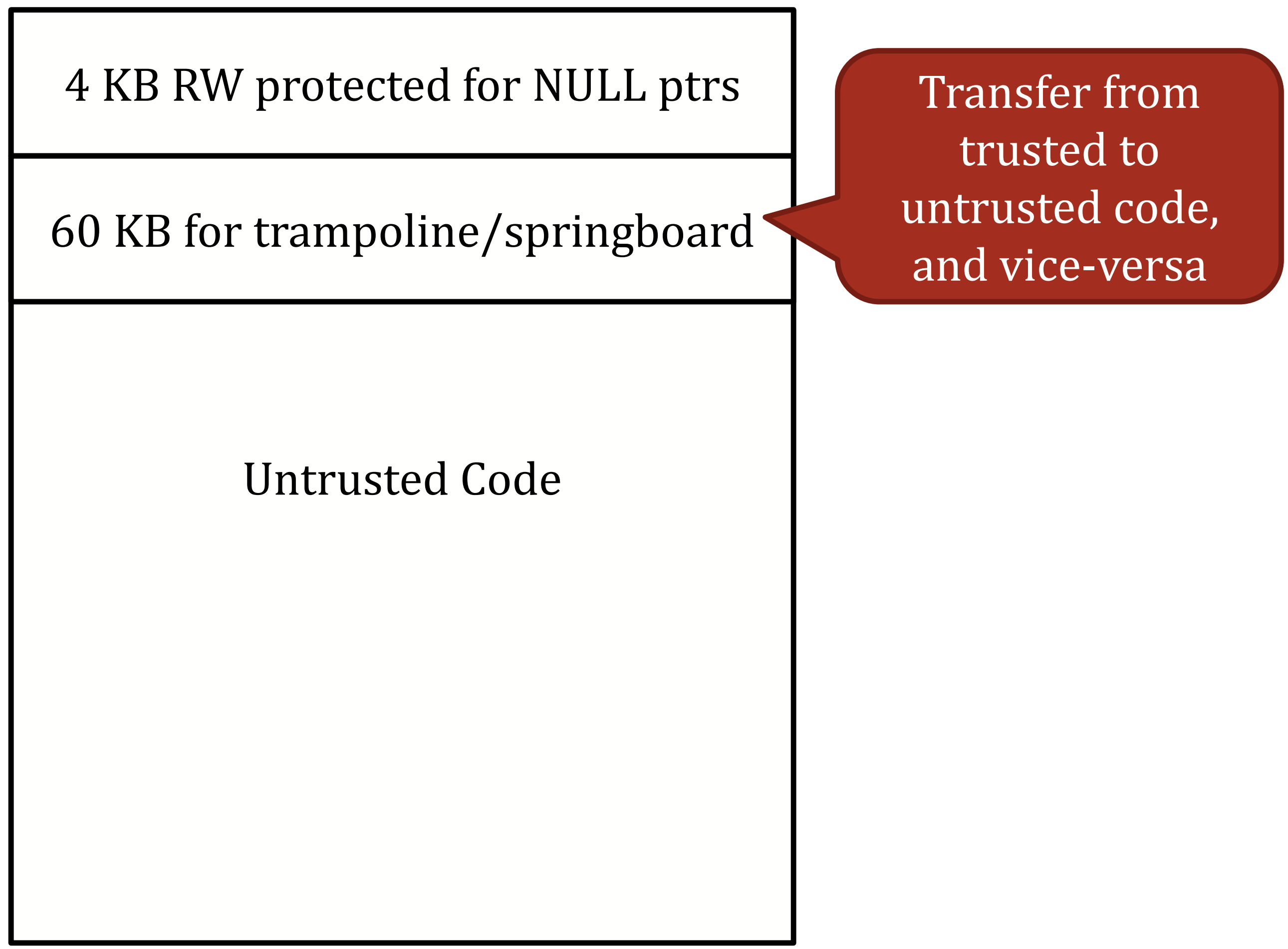
C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

What do these obligations guarantee?

Guarantees

- Data integrity: no loads or stores outside of sandbox
  - Think back to SFI paper
- Reliable disassembly
- No unsafe instructions
- Control flow integrity

NACL Module At Runtime



Performance - Quake

Run #	Native Client	Linux Executable
1	143.2	142.9
2	143.6	143.4
3	144.2	143.5
Average	143.7	143.3

Table 8: Quake performance comparison. Numbers are in frames per second.


Questions?

END

TOC/TOU

- Time of Check/Time of Use bugs are a type of race condition

time

↓

$ open("myfile");
monitor does complex check

$ ln –s myfile /etc/passwd
monitor OK's
Action performed

monitor OK's
OS carries out action

Software Mandatory Access Control

Fine-grained SFI: SMAC can have different access checks at different instructions.

- isolated code region => no need for NX data

```
call   eax                  ; call a function pointer (destination address)
```

with CFI, and SMAC discharging the NXD requirement, can become:

```
and   eax, 40FFFFFFh        ; mask to ensure address is in code memory
cmp   [eax+4], 12345678h    ; compare opcodes at destination
jne   error_label           ; if not ID value, then fail
call  eax                   ; call function pointer
prefetchnta [AABBCCDDh]     ; label ID, used upon the return
```

Context Sensitivity Problems

Suppose A calls C

and B calls C, D.

- CFI uses same call label for C and D due to B.

How to prevent A from calling D?

- duplicate C into C_A and C_B, or
- use more complicated labeling mechanism

Optimizations

Guard Zones

- unmapped pages around segment to avoid checking offsets

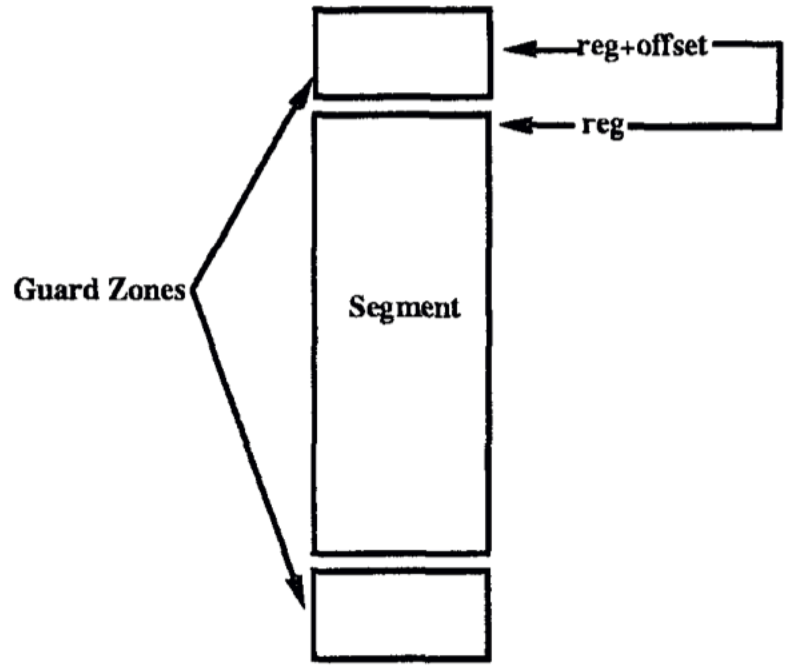


Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.

Lazier SP Check

- check SP only before jumps

Performance

Benchmark		DEC-MIPS					DEC-ALPHA	
		Fault Isolation Overhead	Protection Overhead	Reserved Register Overhead	Instruction Count Overhead	Fault Isolation Overhead (predicted)	Fault Isolation Overhead	Protection Overhead
052.alvinn	FP	1.4%	33.4%	-0.3%	19.4%	0.2%	8.1%	35.5%
bps	FP	5.6%	15.5%	-0.1%	8.9%	5.7%	4.7%	20.3%
cholesky	FP	0.0%	22.7%	0.5%	6.5%	-1.5%	0.0%	9.3%
026.compress	INT	3.3%	13.3%	0.0%	10.9%	4.4%	-4.3%	0.0%
056.ear	FP	-1.2%	19.1%	0.2%	12.4%	2.2%	3.7%	18.3%
023.eqntott	INT	2.9%	34.4%	1.0%	2.7%	2.2%	2.3%	17.4%
008.espresso	INT	12.4%	27.0%	-1.6%	11.8%	10.5%	13.3%	33.6%
001.gcc1.35	INT	3.1%	18.7%	-9.4%	17.0%	8.9%	NA	NA
022.li	INT	5.1%	23.4%	0.3%	14.9%	11.4%	5.4%	16.2%
locus	INT	8.7%	30.4%	4.3%	10.3%	8.6%	4.3%	8.7%
mp3d	FP	10.7%	10.7%	0.0%	13.3%	8.7%	0.0%	6.7%
psgrind	INT	10.4%	19.5%	1.3%	12.1%	9.9%	8.0%	36.0%
qcd	FP	0.5%	27.0%	2.0%	8.8%	1.2%	-0.8%	12.1%
072.sc	INT	5.6%	11.2%	7.0%	8.0%	3.8%	NA	NA
tracker	INT	-0.8%	10.5%	0.4%	3.9%	2.1%	10.9%	19.9%
water	FP	0.7%	7.4%	0.3%	6.7%	1.5%	4.3%	12.3%
Average		**4.3%**	**21.8%**	**0.4%**	**10.5%**	**5.0%**	**4.3%**	**17.6%**

store and jump checked

load, store and jump checked

Is it counter-intuitive?

- Slow down “common” case of intra-domain control transfer in order to speed up inter-domain transfer
  - Check every load, store, jump within a domain

- Faster in practice than hardware when inter-domain calls are frequent
  - Context switches are expensive
  - Each cross-module call requires a context switch

Differences between NaCL SFI and Wahbe SFI

- NaCL uses segments for data to ensure loads/stores are within a module
  - Do not need sandboxing overhead for these instructions
- Others?
- After reading Wahbe et al, how would you implement inter-module communication efficiently?

Performance – Micro Benchmarks

	static	aligned	NaCl	increase
ammp	200	203	203	1.5%
art	46.3	48.7	47.2	1.9%
bzip2	103	104	104	1.9%
crafty	113	124	127	12%
eon	79.2	76.9	82.6	4.3%
equake	62.3	62.9	62.5	0.3%
gap	63.9	64.0	65.4	2.4%
gcc	52.3	54.7	57.0	9.0%
gzip	149	149	148	-0.7%
mcf	65.7	65.7	66.2	0.8%
mesa	87.4	89.8	92.5	5.8%
parser	126	128	128	1.6%
perlbmk	94.0	99.3	106	13%
twolf	154	163	165	7.1%
vortex	112	116	124	11%
vpr	90.7	88.4	89.6	-1.2%

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.