



# Software-Based Fault Isolation

Jinseong Jeon

# fault isolation?

## **isolation** NOUN

 BrE ,aɪsəˈleɪʃn

 NAmE ,aɪsəˈleɪʃn

[UNCOUNTABLE]

---

**1**

the act of separating somebody/something; the state of being separate

- *geographical isolation*
- *an **isolation hospital/ward** (= for people with infectious diseases)*
- **isolation (from somebody/something)** *The country has been threatened with complete isolation from the international community unless the atrocities stop.*
- *He lives in **splendid isolation** (= far from, or in a superior position to, everyone else).*
- *the isolation of the polio virus*

**2 isolation (from somebody/something)**

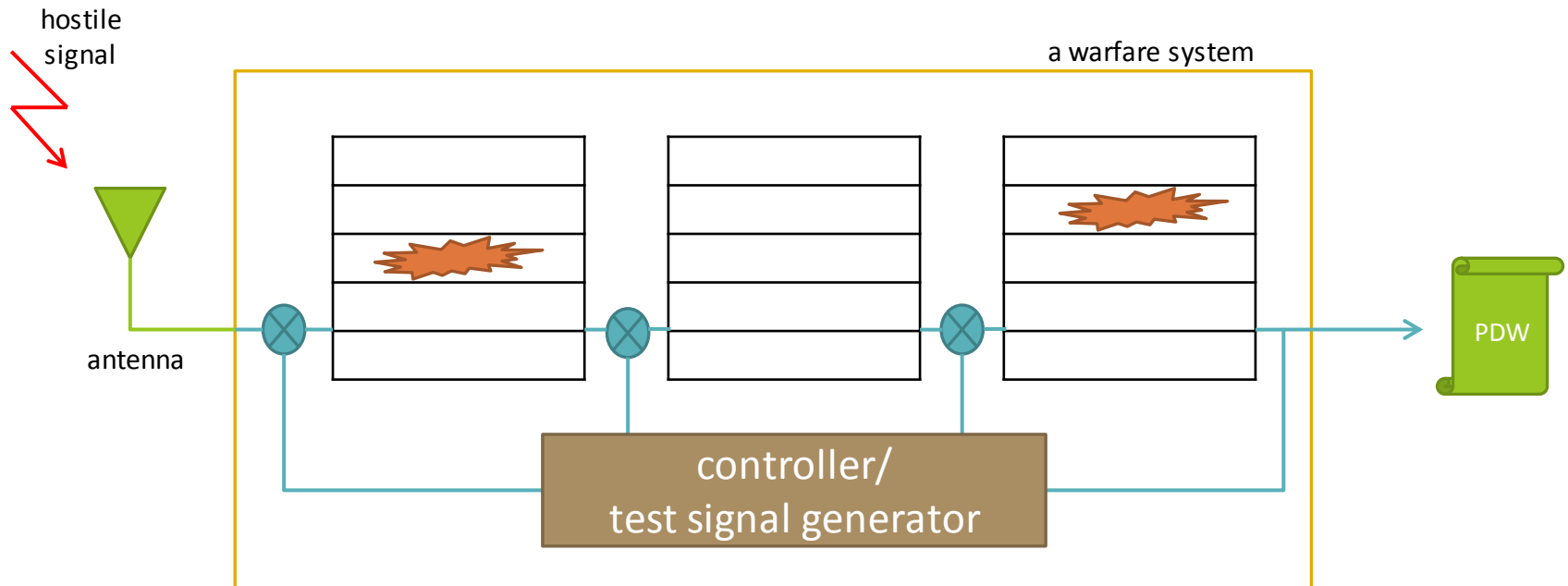
the state of being alone or lonely

- *Many unemployed people experience feelings of isolation and depression.*

- So, fault isolation is the act of separating something faulty?

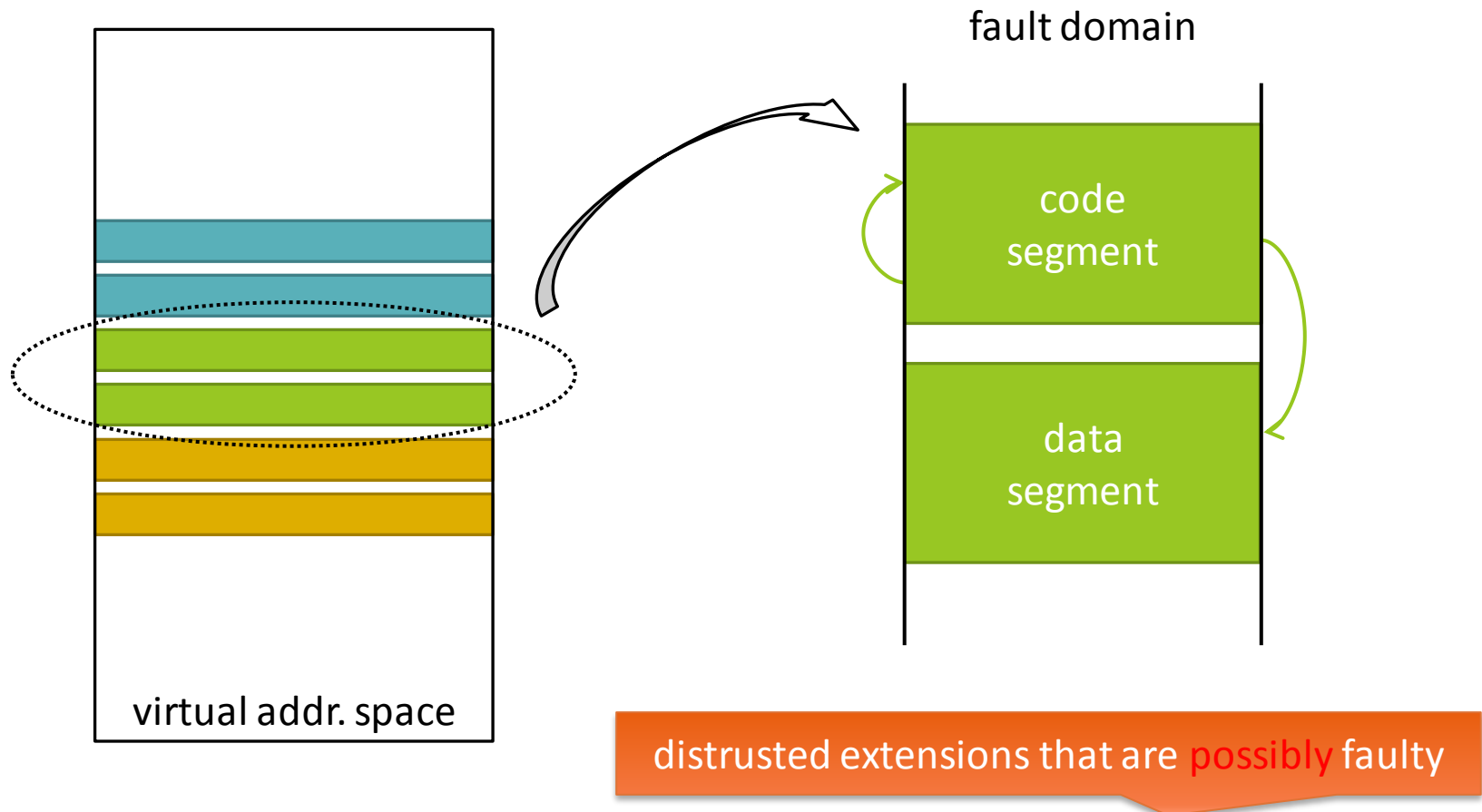
# fault detection & isolation

- “monitor a system, identify when a fault has occurred, and **pinpoint** the type of fault and its location” – Wikipedia



- Since one single operation costs a lot, warfare systems are mandatory to continue commands, detecting and isolating faulty units.

# software-based fault isolation



- Software-based fault isolation is the act of separating ~~something faulty~~.

# Efficient Software-Based Fault Isolation

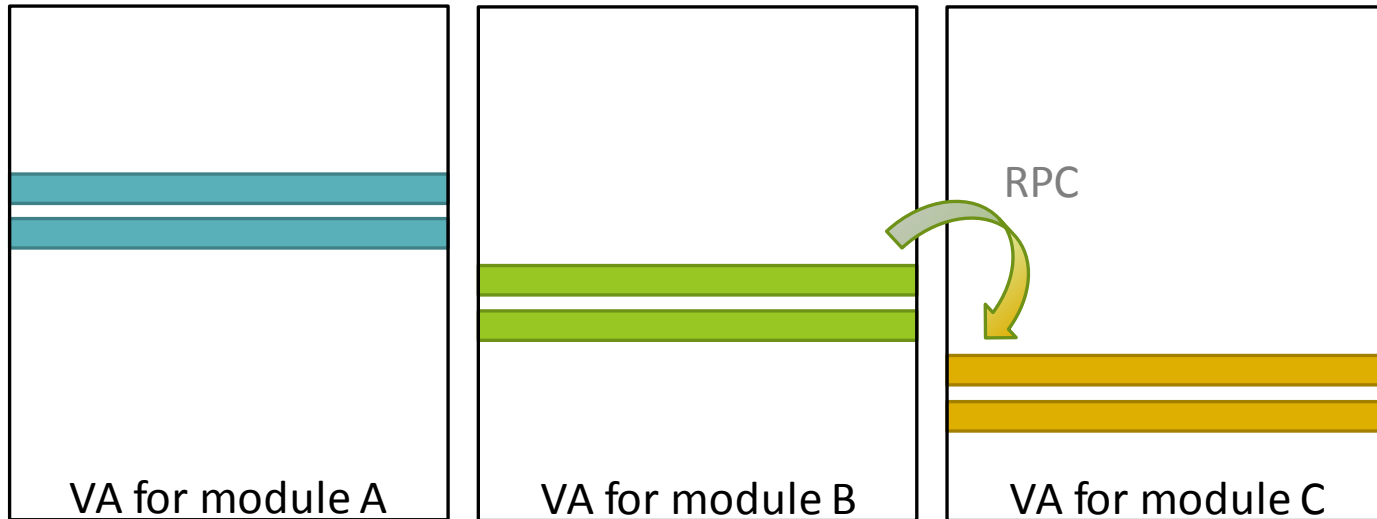
R. Vahbe, S. Lucco, T. E. Anderson, and S. L. Graham

SOSP '93

# fault isolation?

- need to incorporate independently developed software modules
  - micro-kernel design
    - BSD network packet filter
    - application-specific virtual memory management
    - Active Messages
  - extensible software
    - MS object linking and embedding system
    - Quark Xpress desktop publishing system
  - high I/O processes
    - POSTGRES
- need to prevent faults in extension code from corrupting other codes or permanent data while cooperating
- Hence, fault isolation is an act of separating distrusted extensions.

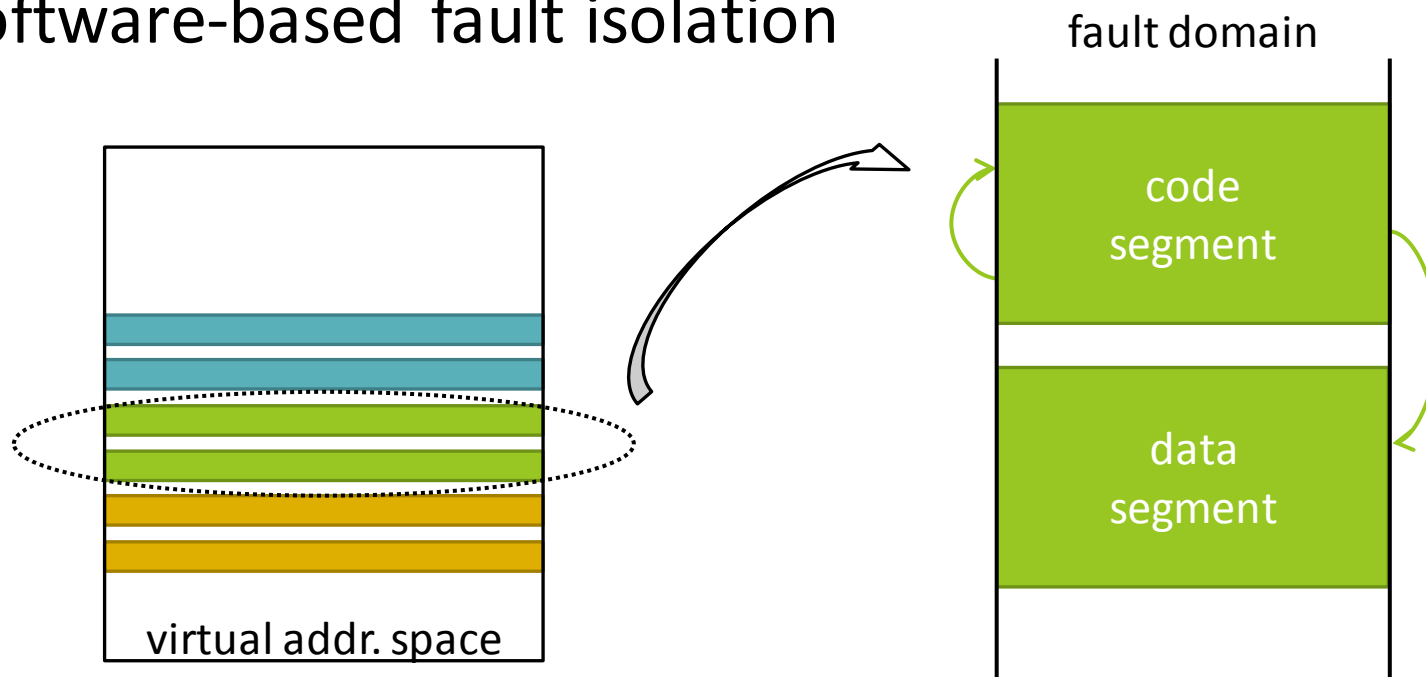
# hardware-based fault isolation



- place each software module in its own address space
- communicate through Remote Procedure Call (RPC)
  - trap into the OS kernel,
  - copying each argument from the caller to the callee,
  - saving and restoring registers,
  - switching hardware address space,
  - trap back to user level.

expensive

# software-based fault isolation



- load extension codes and their data into their own *fault domain*
  - fault domain = code segment + data segment
- enforce security policies that
  - a distrusted module is prohibited from writing or jumping outside its fault domain.
  - i.e. those distrusted modules cannot modify/execute each other's data/code.
  - the only way to do is to use explicit cross fault-domain communication.



# possible questions

- how to enforce such security policies?
  - by binary rewriting
- what to rewrite, and how?
  - unsafe instruction
    - that cannot be statically verified to be within the correct segment
  - use dedicated registers
    - segment matching
    - address sandboxing
- how to share process resources and data?
  - trusted arbitration code
  - virtual address aliasing (or, shared segment matching)
- how to communicate with other fault domains?
  - explicit cross-fault-domain RPC interface
  - stub and jump table

# segment matching

- fault domain

- = code segment + data segment
- shares a unique pattern of upper bits, “segment identifier”

- insert checking code before every unsafe instruction

- indirect jumps or stores, i.e. via registers of which value is determined at runtime

- pseudo code

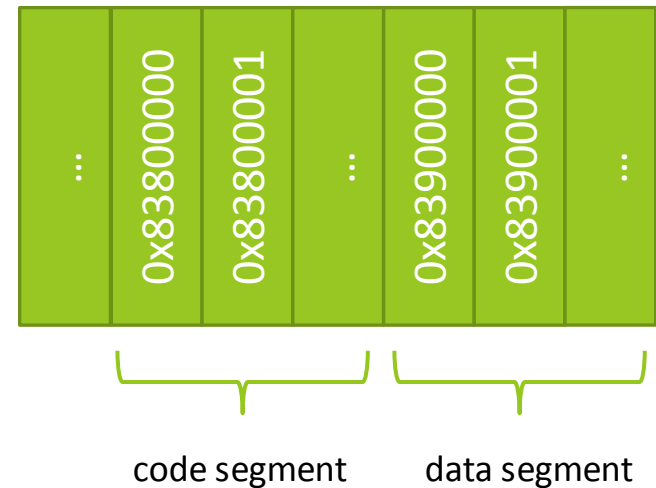
dedicated-reg ← target address

scratch-reg ← (dedicated-reg >> shift-reg)

compare scratch-reg and segment-reg

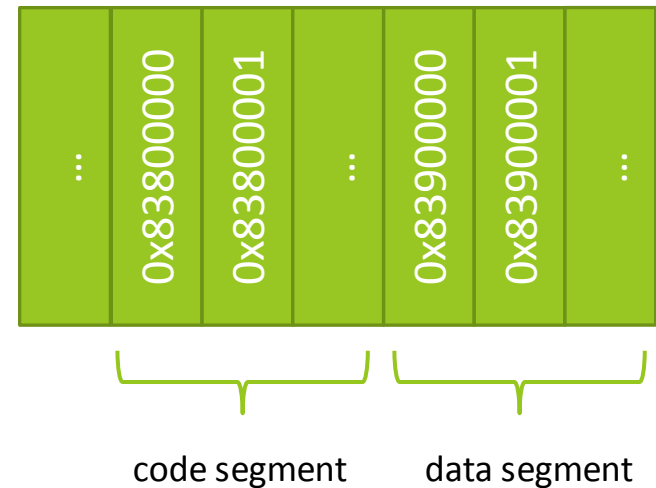
trap if not equal

store/jump using dedicated-reg



# address sandboxing

- instead of checking,  
just setting the upper bits  
to the correct segment identifier
- in the section “Ensure, don’t check”  
at the next paper,
  - check = segment matching
  - ensure = address sandboxing



- pseudo code

$\text{dedicated-reg}^{x2} \leftarrow \text{target-reg} \ \& \ \text{and-mask-reg}$

$\text{dedicated-reg} \leftarrow \text{dedicated-reg} \mid \text{segment-reg}^{x2}$

store/jump using dedicated-reg

# sharing

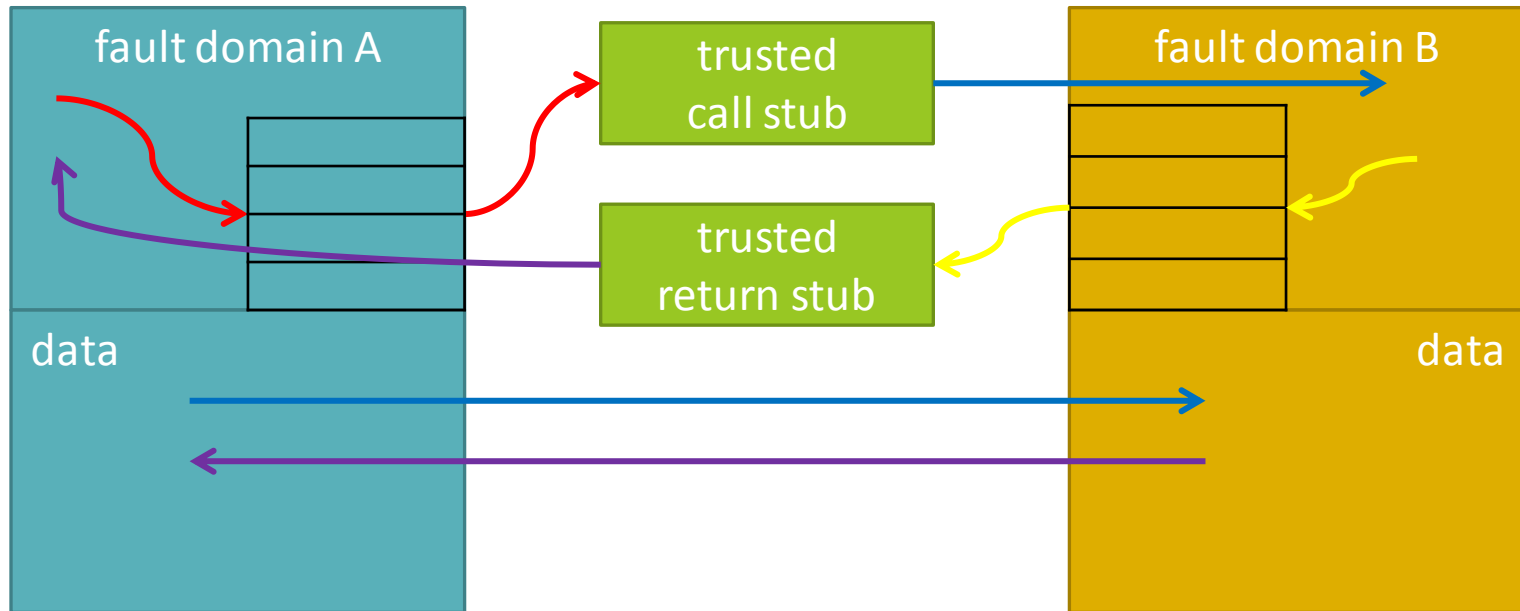
## ■ process resources

- allocated on a per-address-space basis, e.g. file handles
- making OS aware of fault domain – not portable
- modifying distrusted modules' accesses into RPC calls
- allowing “the” trusted part to perform a direct system call and share the result
- seemed similar to the way of sharing the same result of system calls in N-Var. paper?

## ■ data

- read-only
  - trivial because load instructions are not either checked or sandboxed
- read-write
  - lazy pointer swizzling
    - alias shared region into each fault domain (via the same low order bits)
  - shared segment matching
    - a bitmap that holds a mapping from fault domains to accessible segments

# cross fault domain communication

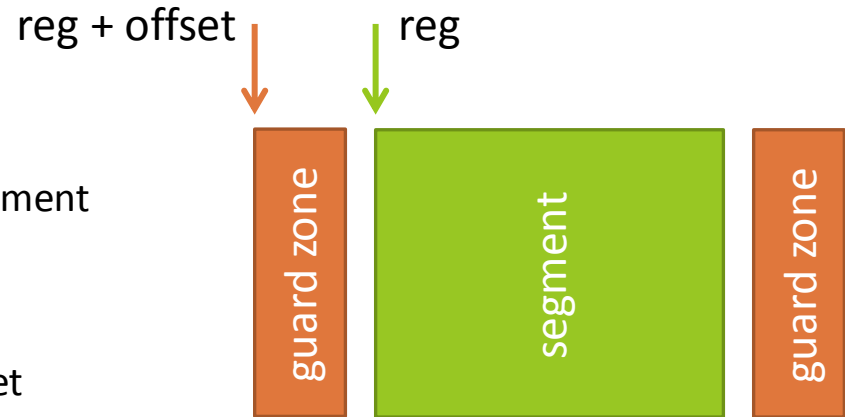


- trusted stubs to handle RPC
  - for each pair of fault domains
  - stub: copy arguments, re/store registers, switch the exe. stack, validate dedicated regs but! no traps or address space switching (thus, cheaper than HW RPC)
- jump tables to transfer control
  - consists of jump instructions of which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

# optimizations

## ■ guard zone

- virtual memory pages adjacent to the segment
- unmapped! i.e. trapped if accessed
- `store value, offset(reg)`
- sandboxing `reg` only, rather than `reg+offset`



## ■ stack pointer as a dedicated register

- # setting stack pointer < # using stack pointer to form address
- once sandboxing the stack pointer whenever it is set, then no sandboxing is required for any other uses of this register.

## ■ avoid sandboxing the stack pointer when modified by a small constant

- unless it is used to transfer control

## ■ removing sandboxing sequences from loops

- sounds like ABCD: Eliminating Array-Bound Checks on Demand

# verification

- divide the program into unsafe regions
  - starting with any modification of dedicated store/jump register
  - ending with one of the followings
    - next instruction is a store/jump to dedicated register
    - next instruction is guaranteed not to be executed
    - no more instructions in the segment
- for each unsafe regions,  
check whether dedicated registers are valid at region exit
  - sounds like computing reaching definition analysis in a classic data-flow analysis  
then, checking reaching definition of dedicated registers spans the above definition

# encapsulation overhead

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%

- about 5%
- fairly correct prediction  
 (# additional instruction - # saved floating point interlock cycles) / cycle-per-second

-----  
 original-execution-time-seconds



## when to use SFI



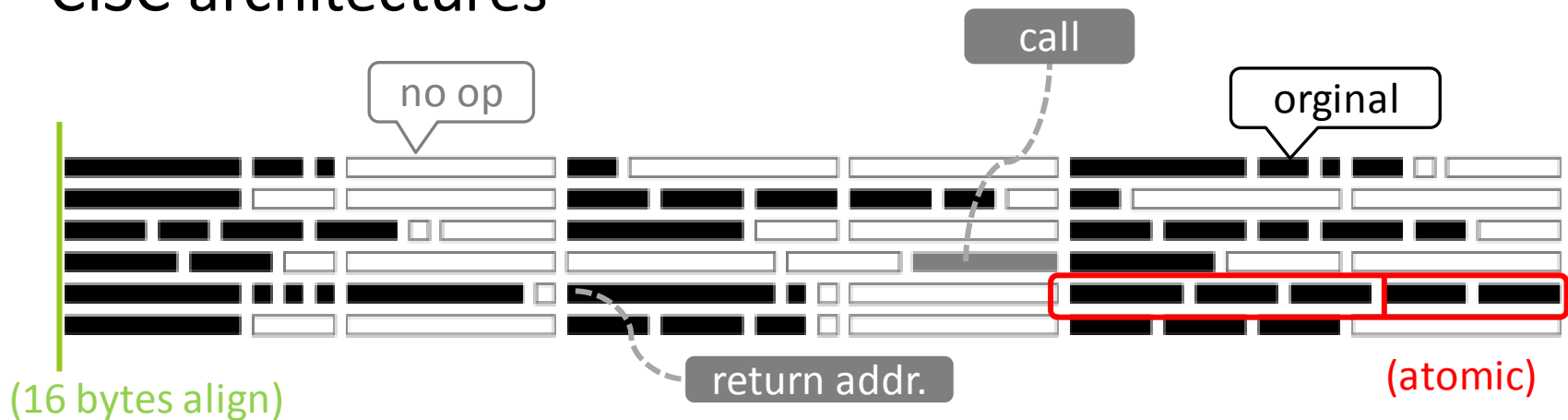
- $(1-r) t_c = h t_d$ 
  - $t_d$ : % of time spent in distrusted code (1.00, 100%)
  - $t_c$ : % of time spent in crossing among fault domains
  - $h$ : overhead of encapsulation (0.043)
  - $r$ : ratio of time of SFI RPC to that of “competing” HW RPC

# Evaluating SFI for a CISC Architecture

S. McCamant and G. Morrisett

USENIX-SS '06

# 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 low 4 bits zero 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

# optimizations

- three introduced by R. Wahbe et al.
- one-instruction address operations
  - choose code and data region tags that have only a single bit difference
  - then, address need to be cleared only, without being set
  - e.g. code: 0x10000000, data: 0x20000000
  - and \$0x20ffffff, %ebx
- efficient returns
  - modern x86 processors have a shadow stack where return addresses are cached.
  - (as long as a single thread is running)

before:	after:
popl %ebx	and \$0x10ffffff0, (%esp)
and \$0x10ffffff0, %ebx	
Jmp *%ebx	

# verification

- security property to check:
  - a program never jumps outside code segment
  - or writes outside its data segment
- for each position in the rewritten instruction stream,
  - conservatively collect all possible contents of the processor's registers at anytime when execution might reach that point
    - meet-over-all-paths (MOP) data-flow analysis?

# rewriter implementation

```
push %ebp
mov  %esp, %ebp
mov  8(%ebp), %edx
mov  48(%edx), %edx
lea  1(%eax), %ecx
```

```
push %ebp
mov  %esp, %ebp
mov  8(%ebp), %edx
mov  48(%edx), %edx
lea  1(%eax), %ecx
lea  0(%esi), %esi
```

no sandbox  
because  
%esp is safe

no op

(.p2align)

```
mov  %ecx, (%ebx)
pop  %ebp
```

```
lea  48(%edx), %ebx
lea  0(%esi), %esi
lea  0(%edi), %edi
```

rsvd

```
and  $0x20ffffff, %ebx
mov  %ecx, (%ebx)
pop  %ebp
lea  0(%esi), %esi
```

sandbox  
because %ebp  
changed

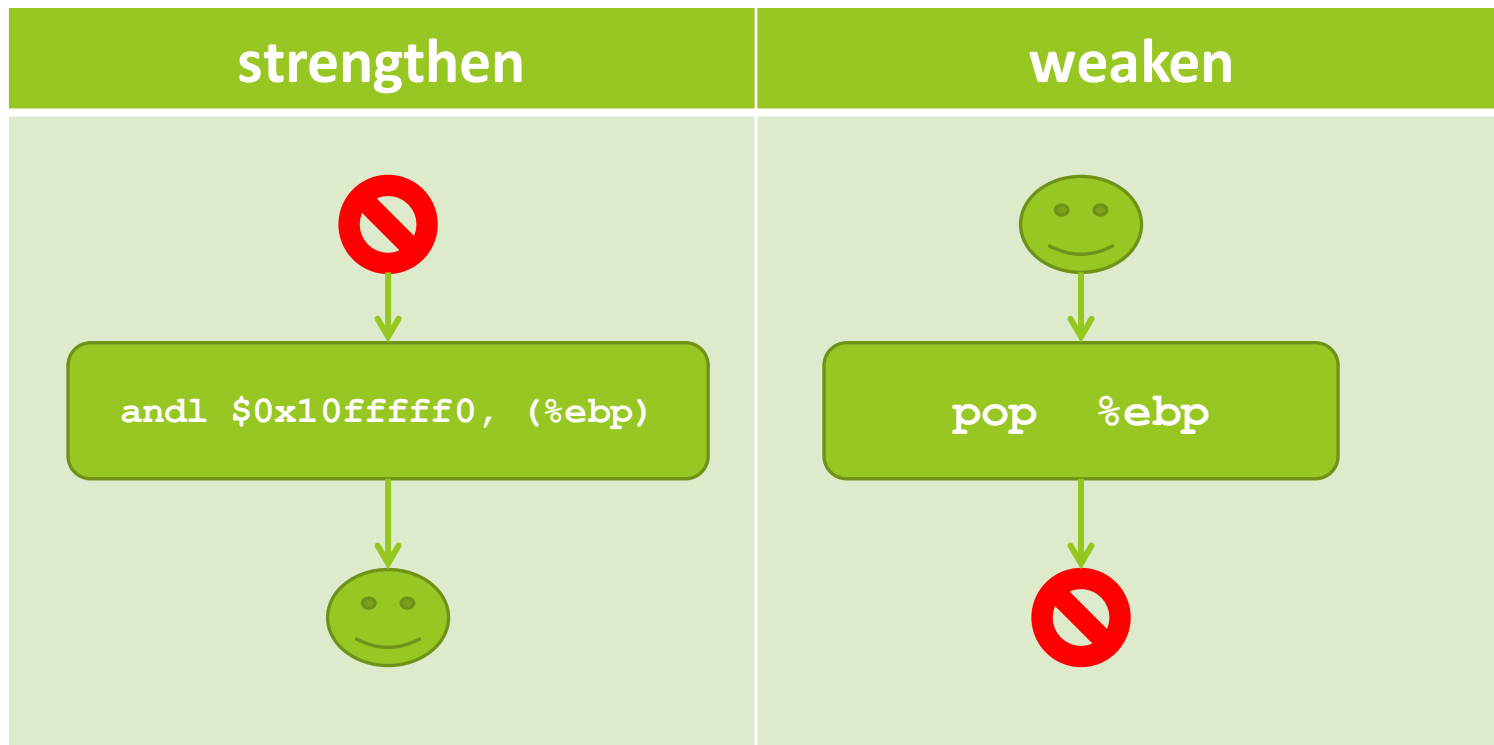
```
ret
```

```
and  $0x20ffffff, %ebp
andl $0x10ffffff0, (%ebp)
ret
```

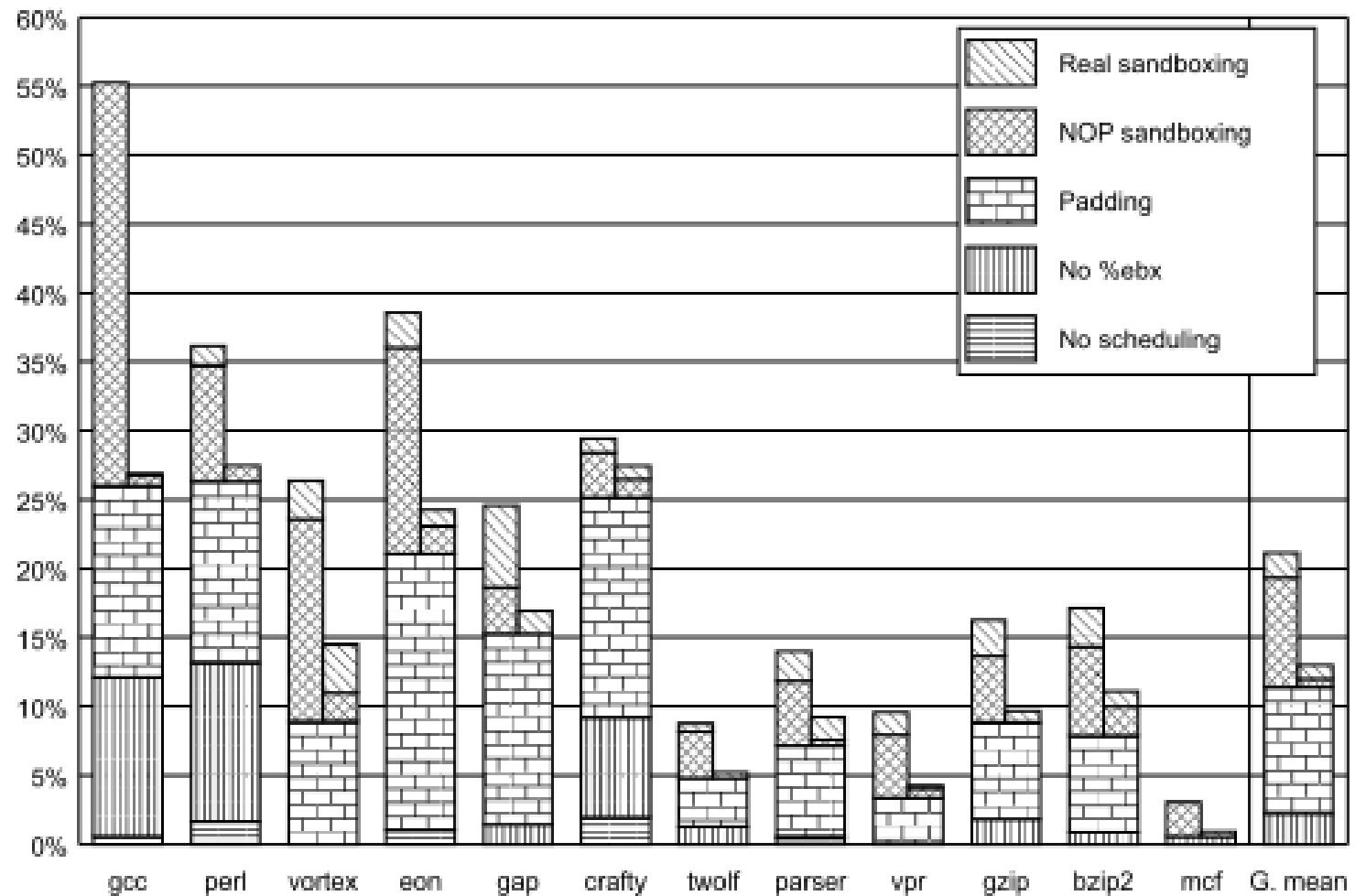
sandbox before  
return

# verifier implementation

- merely a finite-state machine (with only two states?)



# performance





## case study

- VXA, an archiving system where archives contain their own decompressor
- uses a virtualized execution environment VX32 to isolate decompressor code modules
- VX32 relies on hardware support for protecting against unsafe writes
  - less portable; not supported in the 64-bit mode, but work in 32-bit compatibility mode

# formal analysis

- using ACL2, a theorem-proving system, prove the soundness of verifier
- ACL2, a restricted subset of Common Lisp
- proof is a simplified model of the verifier, along with a simulator for x86 instruction set
- proves that  
if the verifier approves the rewritten code,  
it will only execute safe instructions for all possible input states

# references

- Efficient Software-based Fault Isolation, R. Wahbe et al., 1993
- Evaluating SFI for a CISC Architecture, S. McCamant and G. Morrisett, 2006
- Several presentations by
  - Maitree Kanungo,
  - J. Garrett Morris,
  - Christopher Head,
  - Kenneth Chiu,
  - Cole Trapnell,
  - and an anonymous presenter