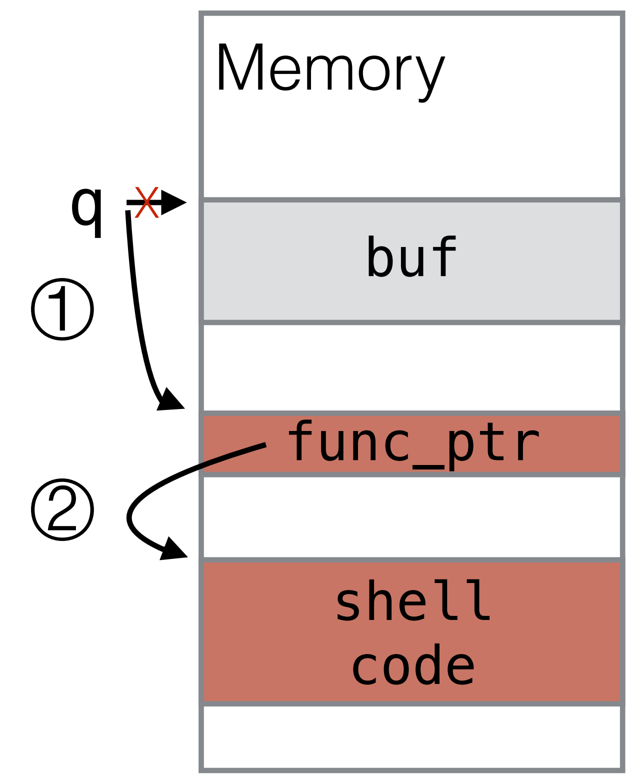
Code-Pointer Integrity

Volodymyr Kuznetsov, László Szekeres, Mathias Payer,
George Candea, R. Sekar, Dawn Song

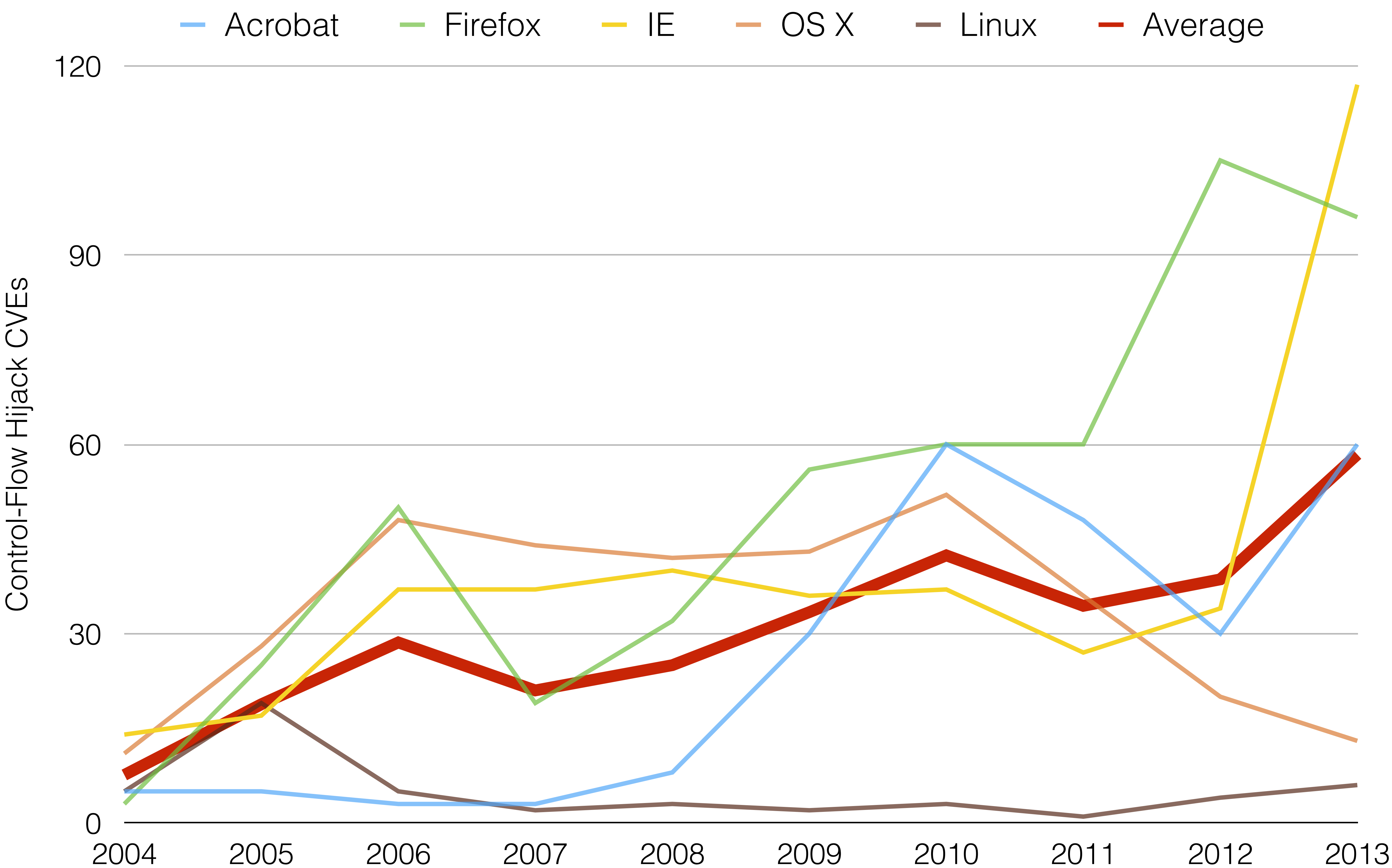
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Stony Brook University
PURDUE UNIVERSITY

Control-Flow Hijack Attack

```
① int *q = buf + input;
② *q = input2;
  …
③ (*func_ptr)();
```



① Attacker corrupts a data pointer
② Attacker uses it to overwrite a code pointer
③ Control-flow is transferred to shell code



Control-flow hijacks are still abundant today!

Memory safety prevents control-flow hijacks

python™ Java™ C# Swift …

… but memory safe programs still rely on C/C++ …

Sample Python program
(Dropbox SDK example):

Python program	3 KLOC of Python
Python runtime	500 KLOC of C
libc	2500 KLOC of C

Memory safety can be retrofitted to C/C++

C/C++	**Overhead**
SoftBound+CETS	116%
CCured (language modifications)	56%
Watchdog (hardware modifications)	29%
AddressSanitizer (approximate)	73%

State of the art:

Control-Flow Integrity

Static property:
limit the set of functions that
can be called at each call site

Coarse-grained CFI can be bypassed [1-4]

and

Finest-grained CFI has 10-21% overhead [5-6]

[1] Göktaş et al., IEEE S&P 2014
[2] Göktaş et al., USENIX Security 2014
[3] Davi et al., USENIX Security 2014
[4] Carlini et al., USENIX Security 2014
[5] Akritidis et al., IEEE S&P 2008
[6] Abadi et al., CCS 2005

Programmers have to choose

Safety
Security

vs

Flexibility
Performance

Code-Pointer Integrity

provides both

Control-flow hijack protection		**Unmodified C/C++**
Practical protection	**and**	**0.5 - 1.9% overhead**
Guaranteed protection		**8.4 - 10.5% overhead**

Key insight: memory safety for code pointers only

Tested on:

FreeBSD hardened
python
SQLite
lame
OpenSSL
PostgreSQL
GraphicsMagick
Apache

Overview

Does it solve a real problem?

➡ How does it work?

- Threat model & background
- Practical protection: CPS
- Guaranteed protection: CPI

How secure is it?

How practical is it?

Threat Model

- Attacker can read/write data, read code
- Attacker cannot:
  - Modify program code
  - Influence program loading

Memory Safety

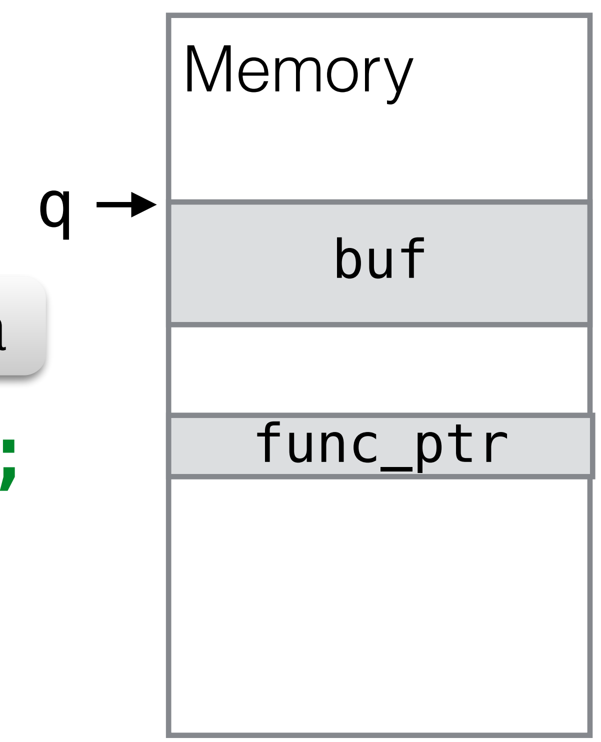
program instrumentation

```
char *buf = malloc(10);
buf_lower = p; buf_upper = p+10;
…
char *q = buf + input;
q_lower = buf_lower; q_upper = buf_upper;
if (q < q_lower || q >= q_upper-size)
  abort();
*q = input2;
…
(*func_ptr)();
```

1. Assign metadata

2. Propagate metadata

3. Check metadata

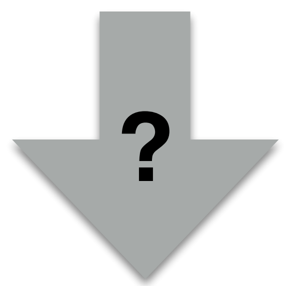


116% average performance overhead

(Nagarakatte et al., PLDI'09 and ISMM'10)

All-or-nothing protection

Memory Safety
116% average performance overhead



Can memory safety be enforced
for code pointers only ?

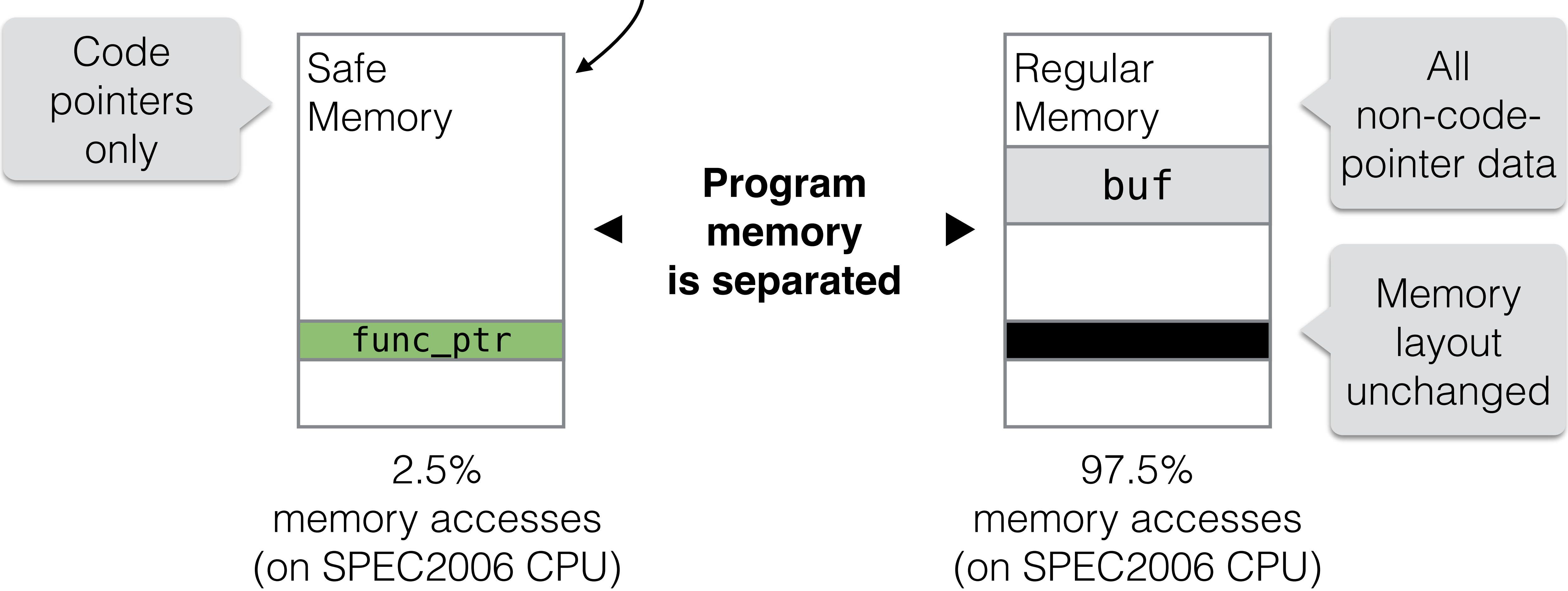
Control-flow hijack protection
1.9% or 8.4% average performance overhead

Practical Protection (CPS): Heap

```
int *q = buf + input;
*q = input2;
…
(*func_ptr)();
```

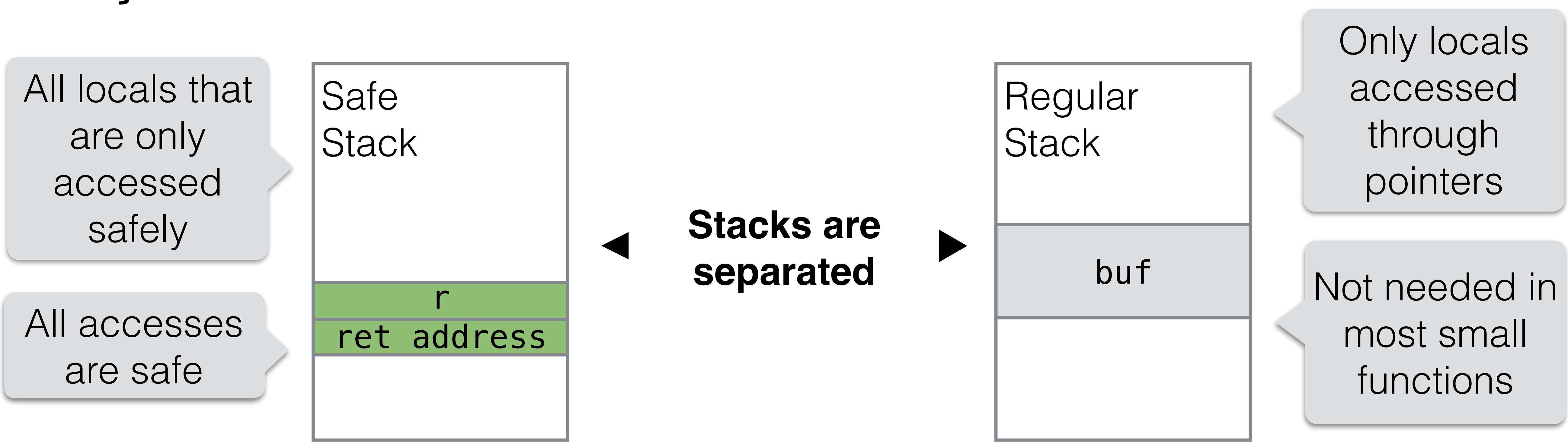
Instructions that access code pointers are identified using type-based static analysis

Separation is enforced using hardware-enforced instruction-level isolation



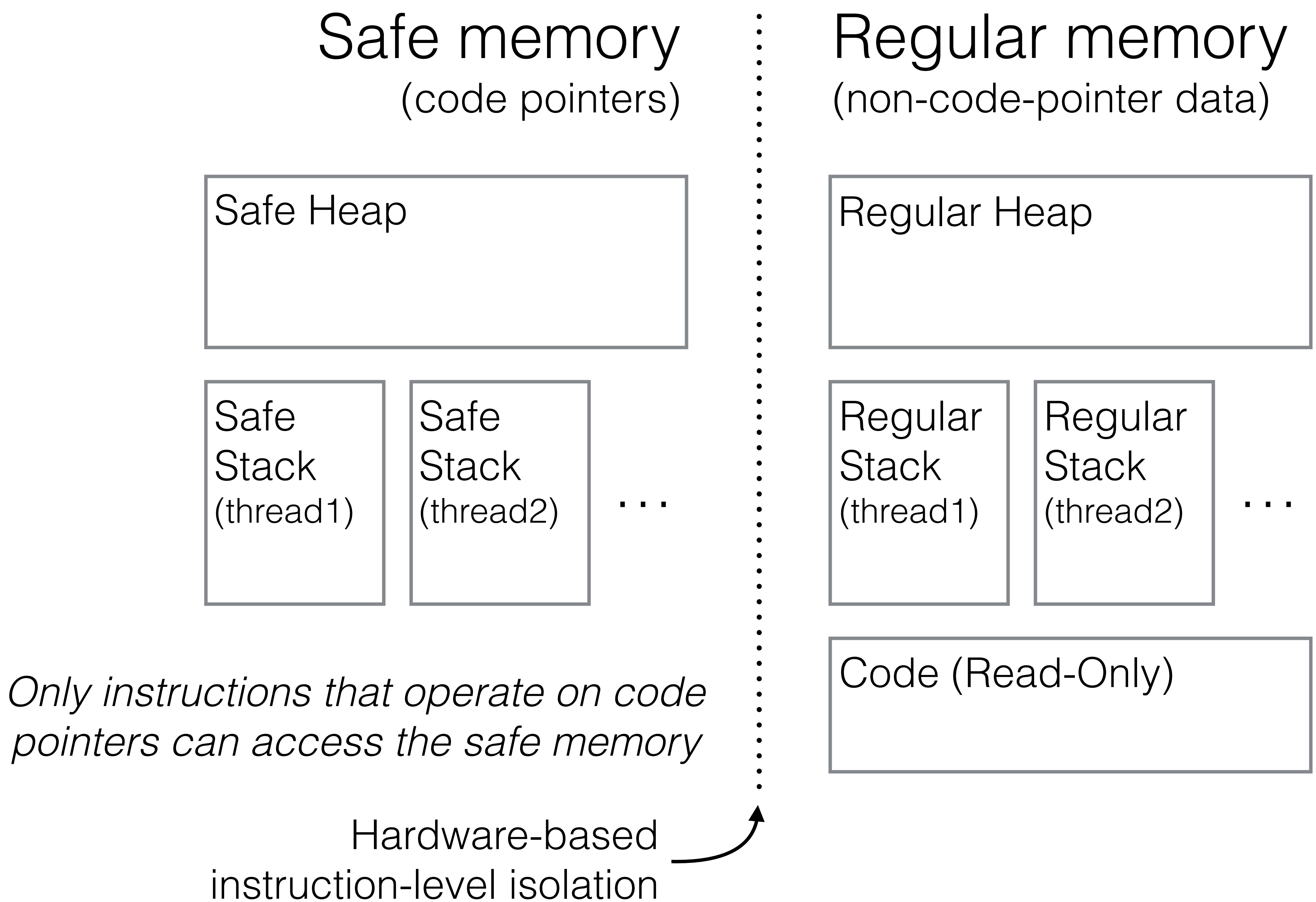
Practical Protection (CPS): Stack

```
int foo() {
  char buf[16];
  int r;
  r = scanf(“%s”, buf);
  return r;
}
```



Safe stack adds <0.1% performance overhead!

Practical Protection (CPS): Memory Layout



The CPS Promise

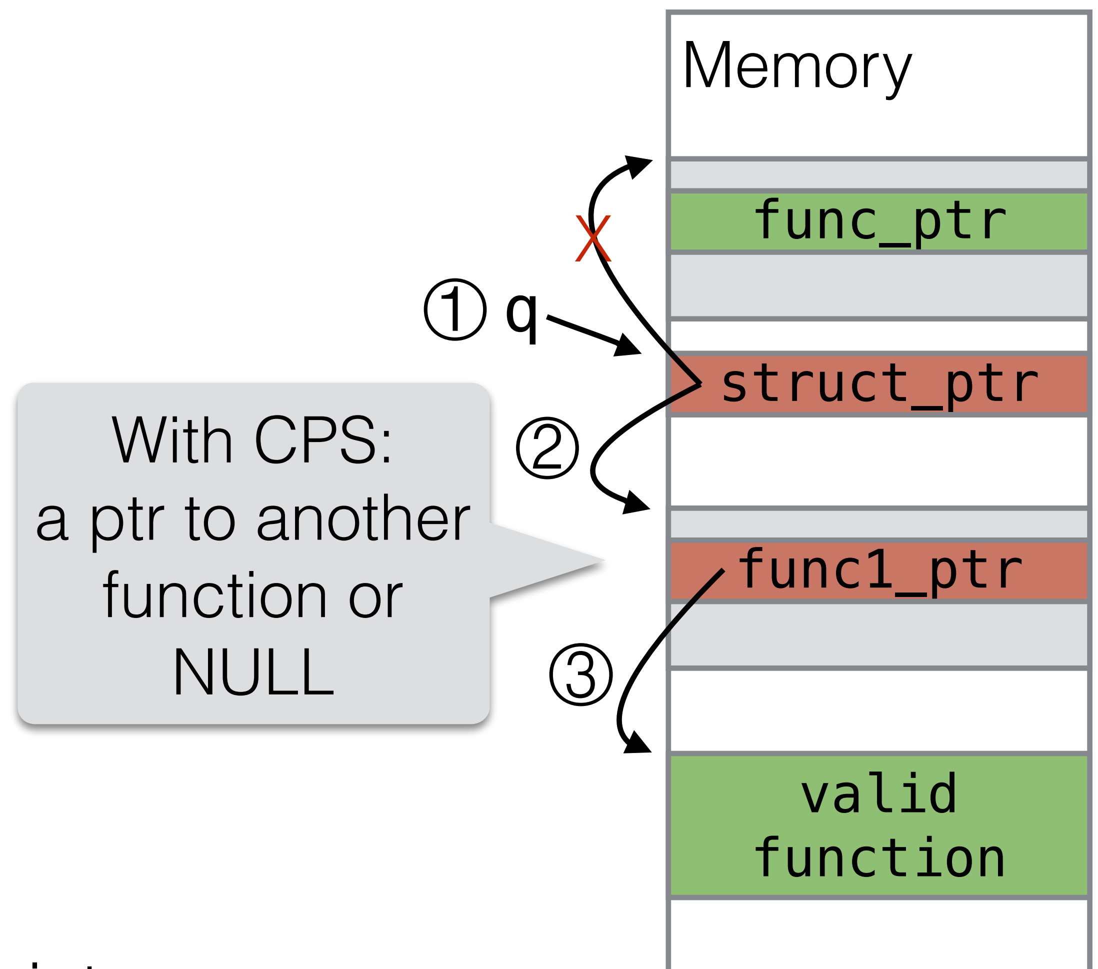
Under CPS, an attacker cannot forge a code pointer

Under CPS, an attacker cannot forge a code pointer

Is this enough? In practice, yes!

Contrived example of an attack on a CPS-protected program

```
① int *q = p + input;
② *q = input2;
  …
③ func_ptr = struct_ptr->f;
④ (*func_ptr)();
```



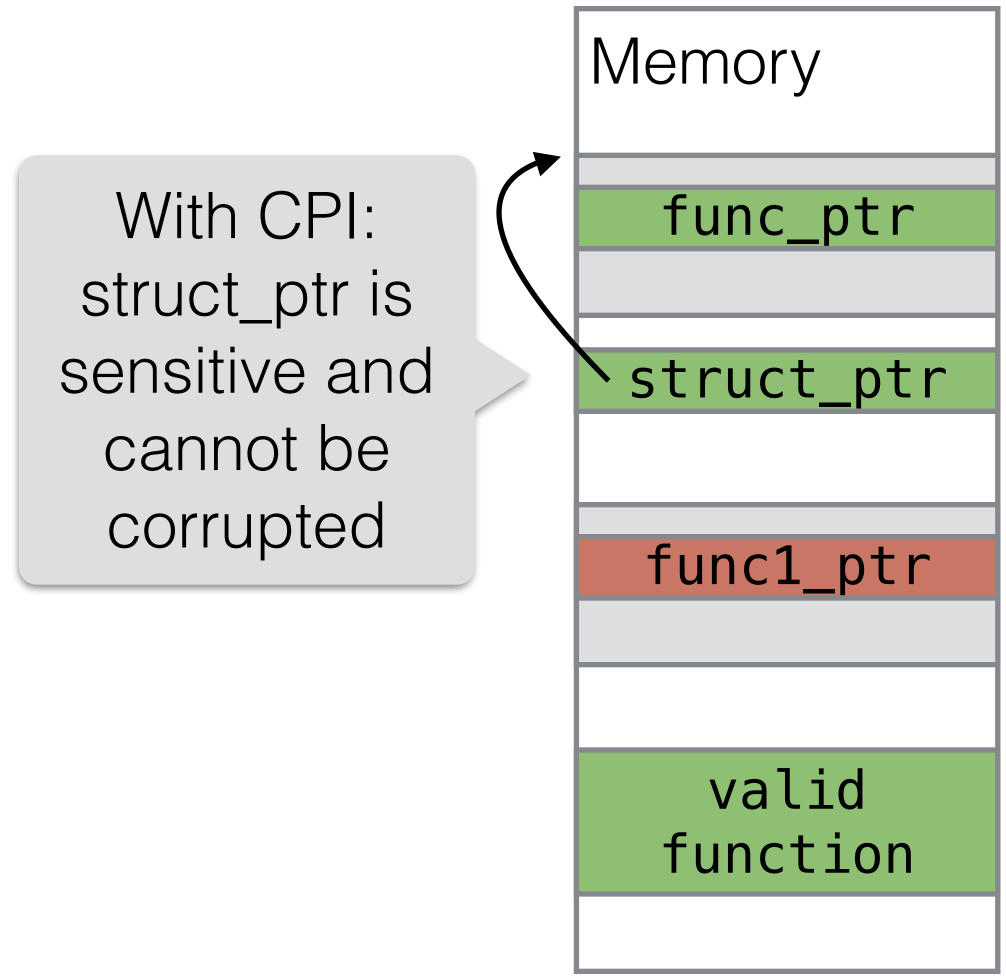
① Attacker corrupts a data pointer
② Attacker uses it to corrupt a struct pointer
③ Program loads a function pointer from wrong location in the safe memory
④ Control-flow is transferred to different function whose address was previously stored in the safe memory

Under CPS, an attacker cannot forge a code pointer

Is this enough?
In practice, yes!

Contrived example of an attack on a CPS-protected program

```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Precise solution: protect all *sensitive*[1] pointers

[1]*Sensitive* pointers = code pointers and **pointers used to access sensitive pointers**

Guaranteed Protection (CPI)

Sensitive pointers = code pointers and
pointers used to access sensitive pointers

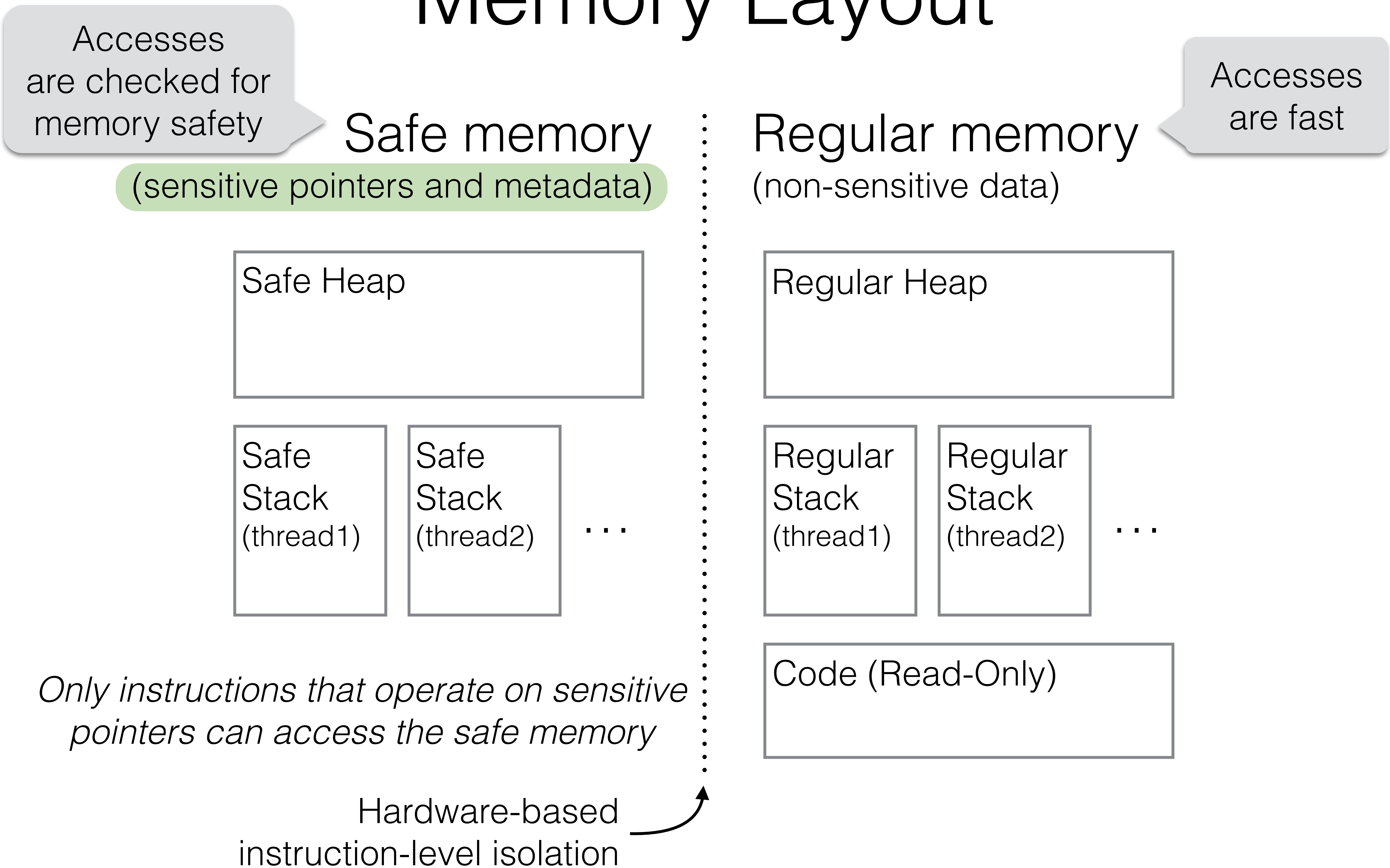
- CPI identifies all sensitive pointers using over-approximate type-based static analysis:

  `is_sensitive(v) = is_sensitive_type(type of v)`

- Over-approximation doesn't hurt security, it only affects performance:

  On SPEC2006 ≤6.5% memory accesses are sensitive

Guaranteed Protection (CPI): Memory Layout



Guaranteed Protection (CPI)

Guaranteed memory safety for all sensitive[1] pointers

$\Downarrow$

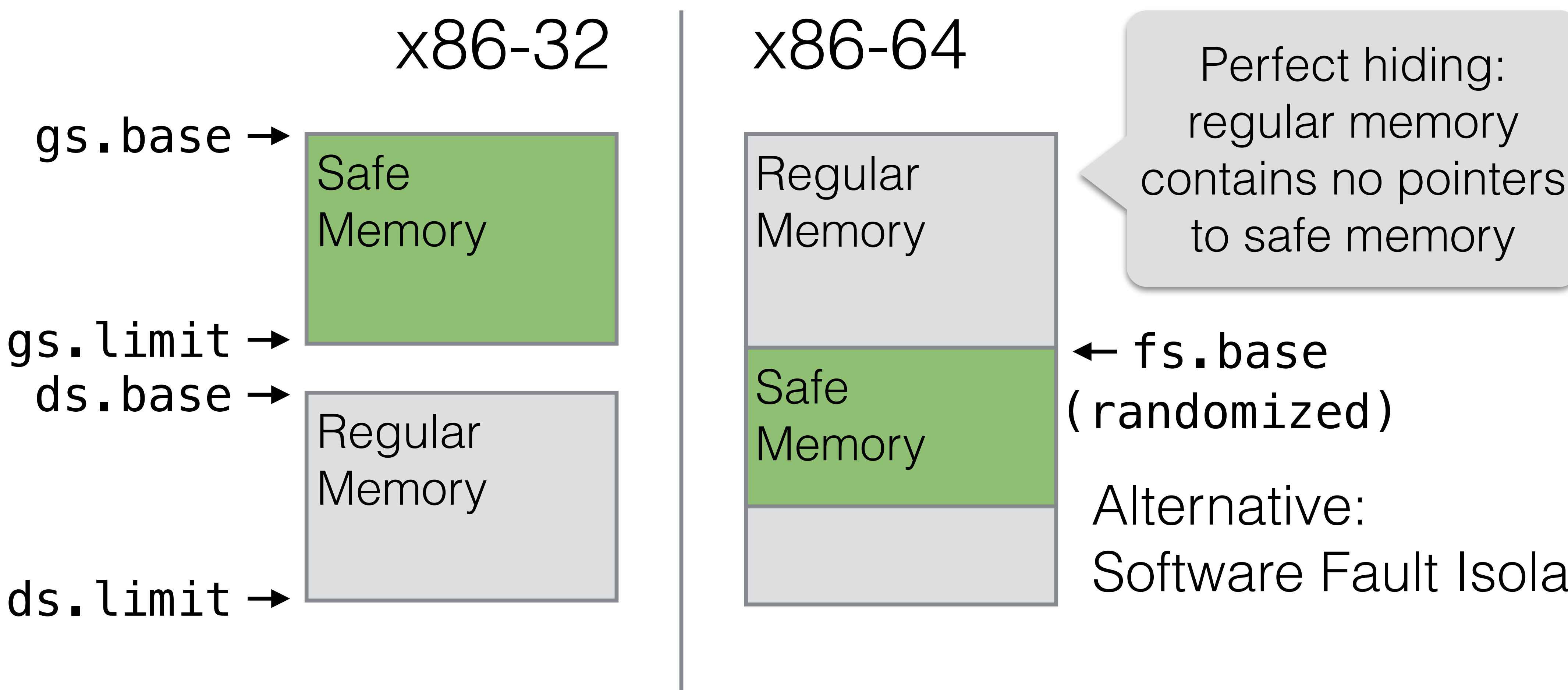
Guaranteed protection against control-flow hijack attacks enabled by memory bugs

[1]Sensitive pointers = code pointers and **pointers used to access sensitive pointers**

Instruction-Level Isolation

```
int *q = ptr + input;
*q = input2;                movl input2, q
…
(*func_ptr)();              call *%gs:func_ptr
```

Dedicated segment register, used only to access the safe memory



Alternative:
Software Fault Isolation

CPS | CPI

- Separate sensitive pointers and regular data

CPS	CPI
Sensitive pointers = code pointers	*Sensitive pointers = code pointers +* ***indirect pointers to sensitive pointers***

- Accessing sensitive pointers is safe

CPS	CPI
Separation	*Separation +* ***runtime checks***

- Accessing regular data is fast

Instruction-level safe region isolation

Overview

- Does it solve a real problem?
- ➡ How does it work?
    - Threat model & background
    - Practical protection: CPS
    - Guaranteed protection: CPI
- How secure is it?
- How practical is it?

Overview

- Does it solve a real problem?
- How does it work?
- ➡ How secure is it?
- How practical is it?

How secure is it?

- RIPE[1] runtime intrusion prevention evaluator:
    - **Both CPS and CPI prevent all attacks from RIPE**
- Future attacks:
    - **CPI correctness proof in the paper**

[1]Wilander at al., ACSAC 2011

Protects Against	Technique	Security Guarantees	Average Overhead
Memory corruption vulnerabilities	Memory Safety	Precise	116%
Control-flow hijack vulnerabilities	**CPI** (Guaranteed protection)	Precise	8.4-10.5%
	CPS (Practical protection)	Strong	0.5-1.9%
	Finest-grained CFI	Medium (attacks may exist) Göktaş el., IEEE S&P 2014	10-21%
	Coarse-grained CFI	Weak (known attacks) Göktaş el., IEEE S&P 2014 and USENIX Security 2014, Davi et al, USENIX Security 2014 Carlini et al., USENIX Security 2014	4.2-16%
	ASLR DEP Stack cookies	Weakest (bypassable + widespread attacks)	~0%

Overview

Does it solve a real problem?

How does it work?

➡ How secure is it?

How practical is it?

Overview

- Does it solve a real problem?
- How does it work?
- How secure is it?
- ➡ How practical is it?
    - Implementation
    - Is it practical?
    - Is it fast enough?

Implementation

```
cc -fcpi foo.c
```

- LLVM-based prototype at http://levee.epfl.ch
- Plan to integrate upstream into LLVM

Implementation

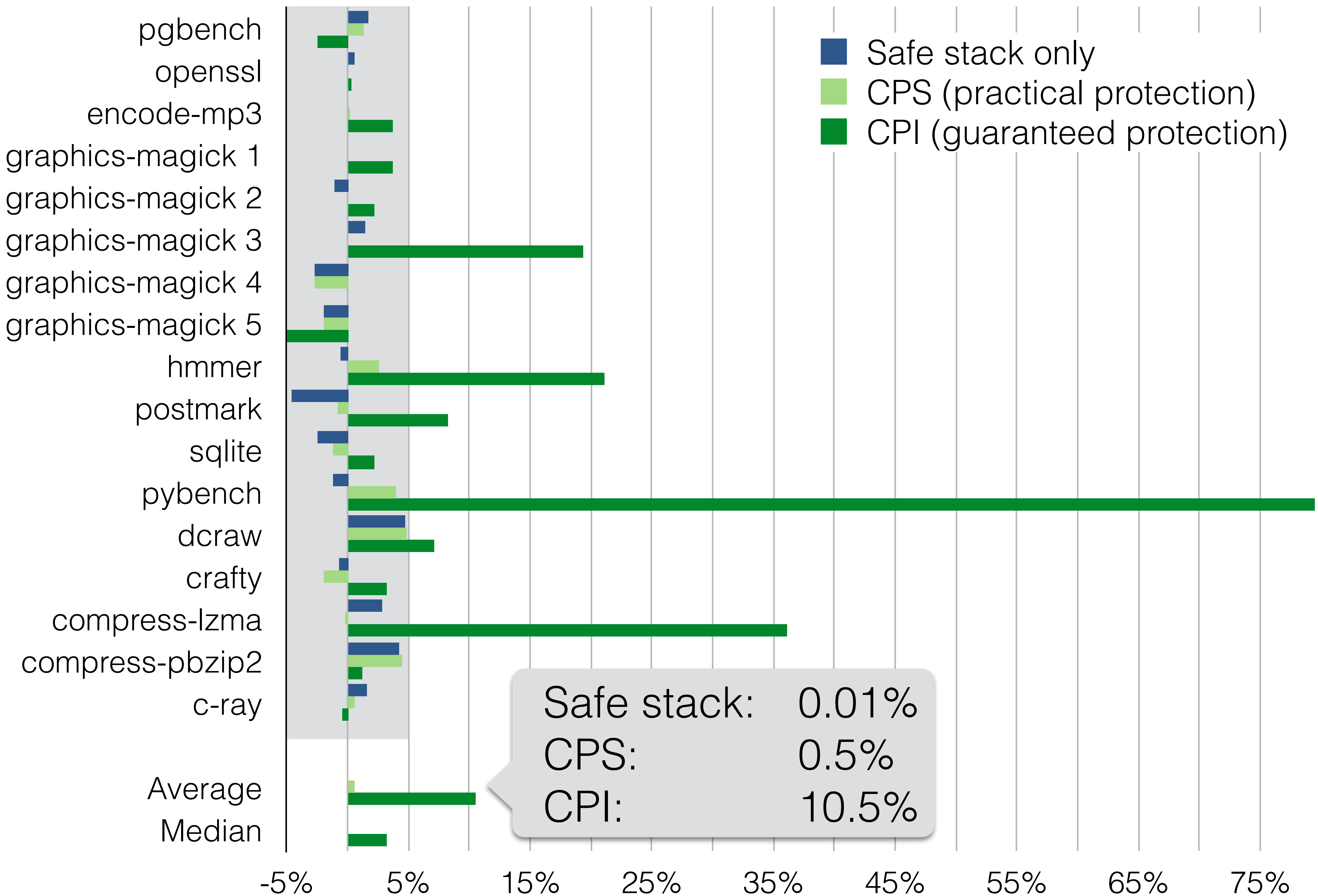
- LLVM-based prototype at http://levee.epfl.ch
  - Front-end (clang):
    Collect type information
  - Back-end (LLVM):
    CPI/CPS and SafeStack instrumentation passes
  - Runtime support (compiler-rt or libc):
    Safe heap and stacks management

Full OS Distribution

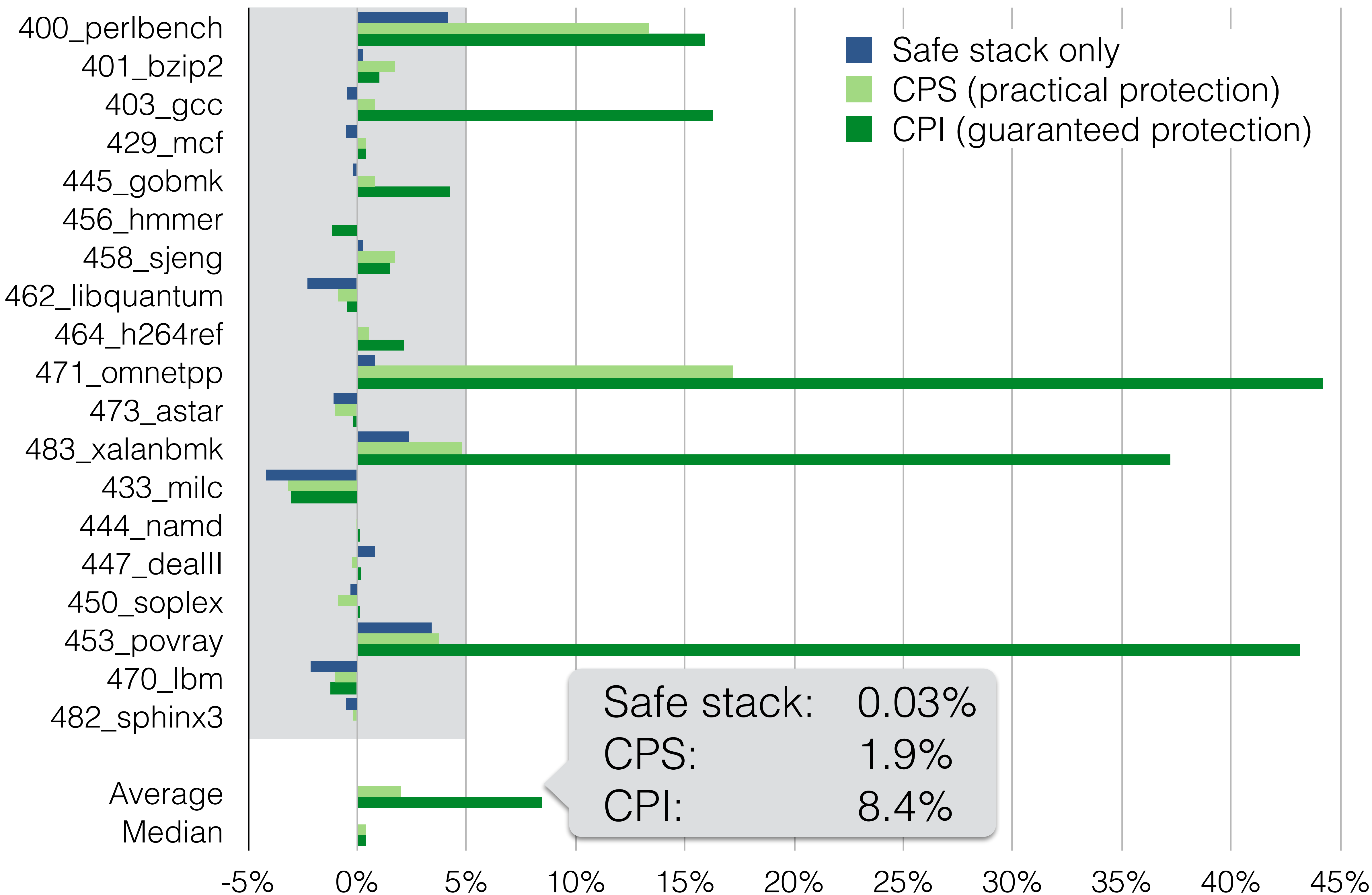
with CPS/CPI protection

- Recompiled the entire FreeBSD userspace…
- … and more than 100 packages

Performance overhead on Phoronix



Performance overhead on SPEC2006 CPU



Overview

Does it solve a real problem?

How does it work?

How secure is it?

➡ How practical is it?

- Implementation
- Is it fast enough?
- Is it practical?

Code-Pointer Integrity

Control-flow hijack protection		**Unmodified C/C++**
Practical protection	**and**	**0.5 - 1.9% overhead**
Guaranteed protection		**8.4 - 10.5% overhead**

Key insight: memory safety for code pointers only

FreeBSD hardened

python

SQLite

lame

OpenSSL

PostgreSQL

GraphicsMagick

Apache

http://levee.epfl.ch