Code-Pointer Integrity

Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea

R. Sekar, Dawn Song

Outline

- Problem Statement
- Existing solutions and their weaknesses
- Code-Pointer Integrity
- Implementation-dependant weakness (Related Paper)
- Discussion

Problem Statement

Problem Statement

- Attackers exploit bugs to cause memory corruption

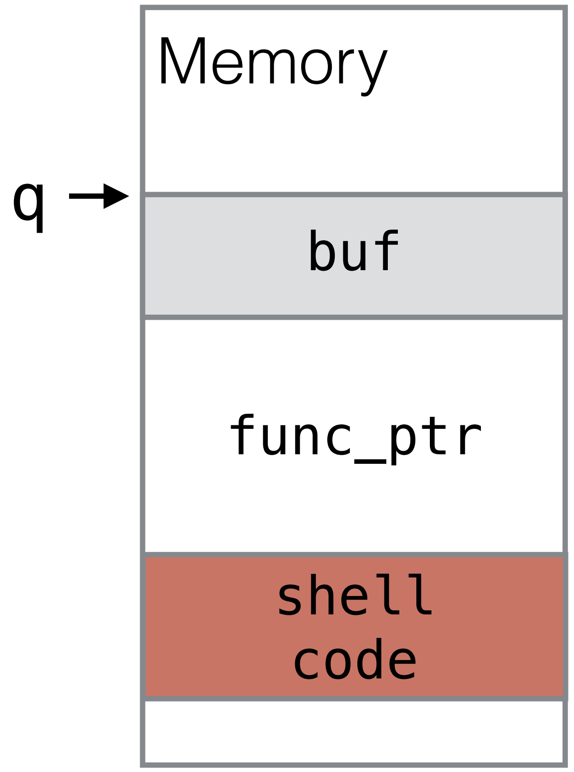
Problem Statement

- Attackers exploit bugs to cause memory corruption
- Steal sensitive data and/or execute code on the system

Problem Statement

- Attackers exploit bugs to cause memory corruption
- Steal sensitive data and/or execute code on the system

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

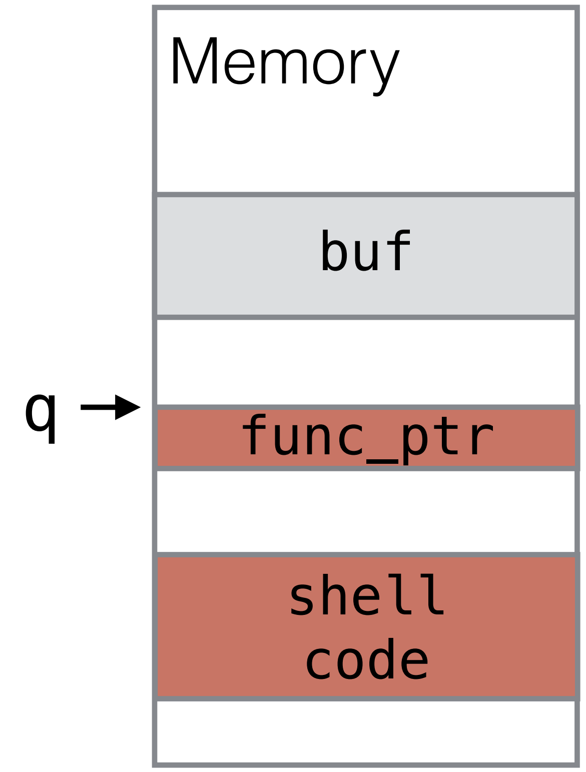


Problem Statement

- Attackers exploit bugs to cause memory corruption

- Steal sensitive data and/or execute code on the system

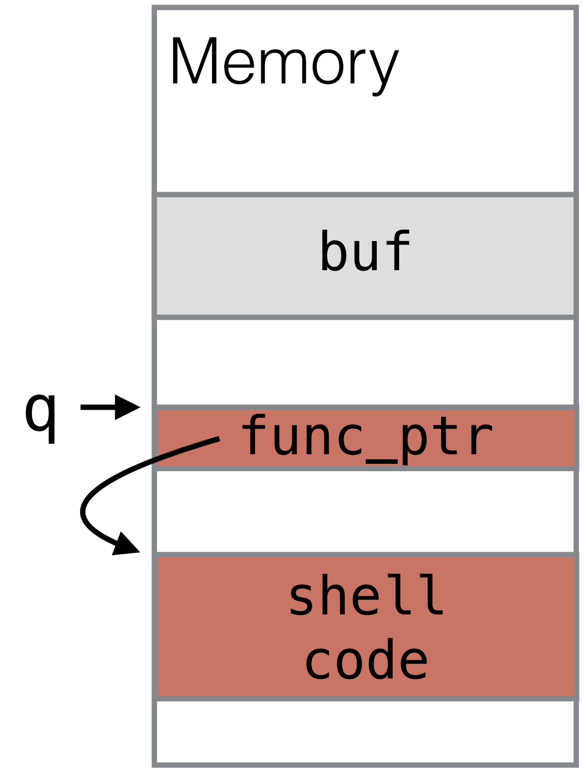
```
int *q = buf + input;
*q = input2;
…
(*func_ptr)();
```



Problem Statement

- Attackers exploit bugs to cause memory corruption
- Steal sensitive data and/or execute code on the system

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



Existing Solutions

Existing Solutions

- Adress Space Layout Randomisation (ASLR)

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
  - Places code and data segments at random addresses
  - Complicates code-reuse (ROP)
  - Defeated by pointer leaks and side channel attacks

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
  - Protect return addresses on the stack
  - Only protect against continuous buffer overflows

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
- Data Execution Prevention (DEP)

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
- Data Execution Prevention (DEP)
- Control-Flow Integrity

Existing Solutions

- Adress Space Layout Randomisation (ASLR)
- Stack Cookies
- Data Execution Prevention (DEP)
- Control-Flow Integrity
- Memory Safety

Control-Flow Integrity

Control-Flow Integrity

- Limit the set of functions that can be called at each call site

Control-Flow Integrity

- Limit the set of functions that can be called at each call site
- Coarse-grained CFI can be bypassed

Control-Flow Integrity

- Limit the set of functions that can be called at each call site
- Coarse-grained CFI can be bypassed
- Finest-grained CFI has 10-21% performance overhead

Memory Safety

Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object

Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object

  ➡ Completely prevents control-flow hijacks

Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object

  ➡ Completely prevents control-flow hijacks

- Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code

Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object

  ➡ Completely prevents control-flow hijacks

- Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code

- Requires runtime checks to verify correctness of pointer computations

Memory Safety

- Guarantees memory objects can only be accessed by pointers properly based on the specific object
  - ➡ Completely prevents control-flow hijacks
- Requires rewriting code in memory-safe languages or retrofitting memory safety onto existing code
- Requires runtime checks to verify correctness of pointer computations
  - ➡ Introduces significant performance overhead (≥2x when retrofitted)

Code-Pointer Integrity

Code-Pointer Integrity

- Goals:
  - Prevent all control-flow hijack attacks
  - Significantly less performance overhead than state-of-the-art

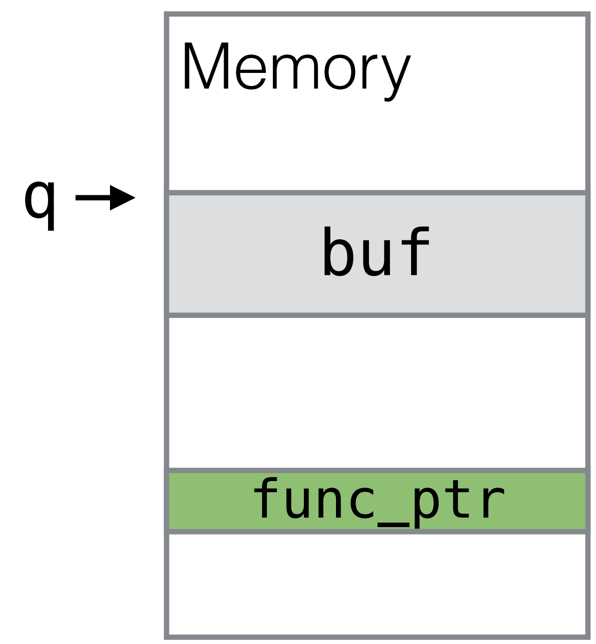
Code-Pointer Integrity

- Goals:
  - Prevent all control-flow hijack attacks
  - Significantly less performance overhead than state-of-the-art
- Idea:
  - Use memory-safety but only protect code-pointers

Code-Pointer Separation

Code-Pointer Separation

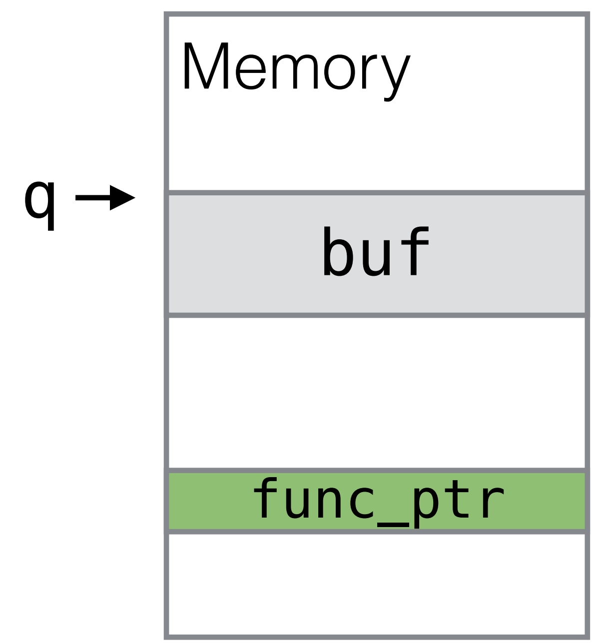
```
int *q = buf + input;
*q = input2;
…
(*func_ptr)();
```



Code-Pointer Separation

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

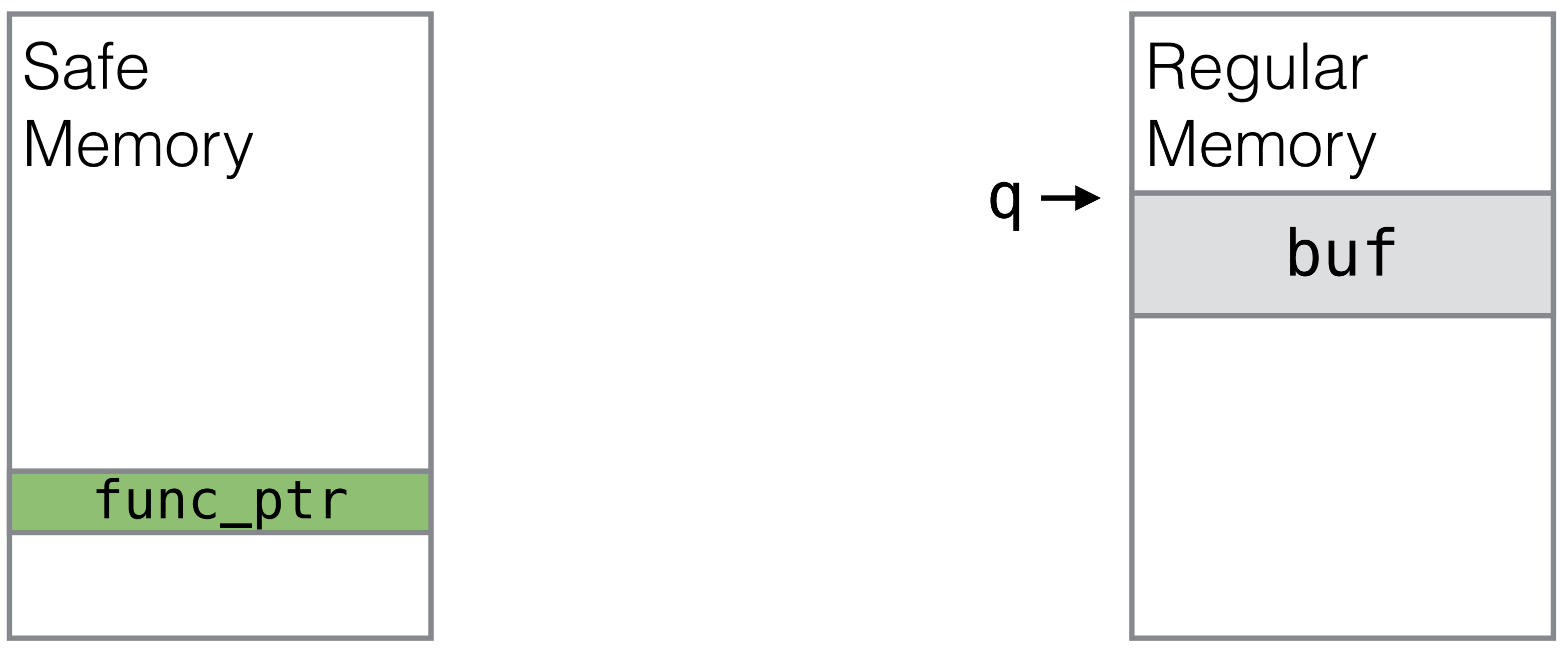
- Type-based static analysis



Code-Pointer Separation

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

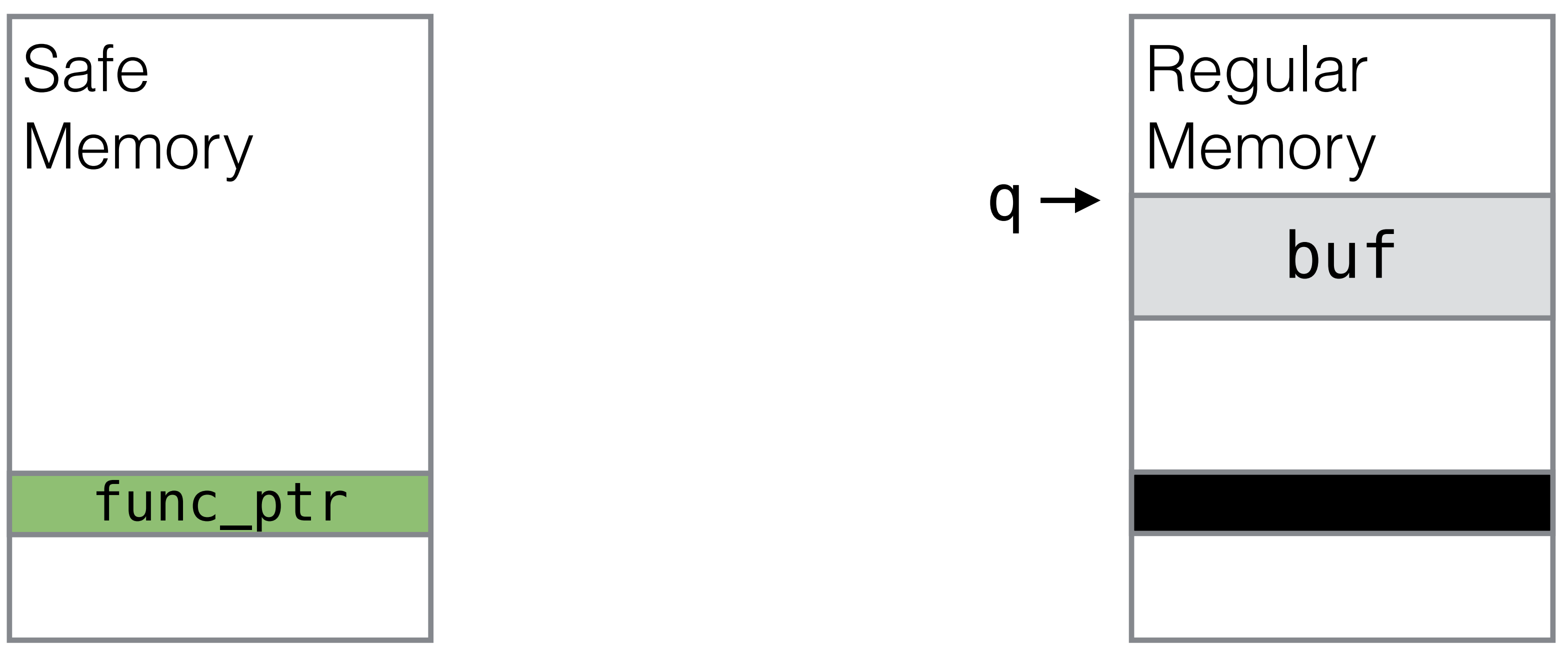
- Type-based static analysis
- Move only code pointers to safe memory
  - ➡ Isolate safe memory on instruction level



Code-Pointer Separation

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

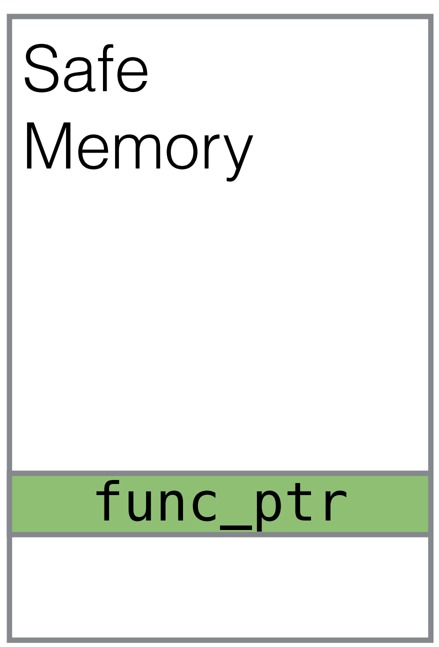
- Type-based static analysis
- Move only code pointers to safe memory
  - ➡ Isolate safe memory on instruction level
- Keep memory layout unchanged



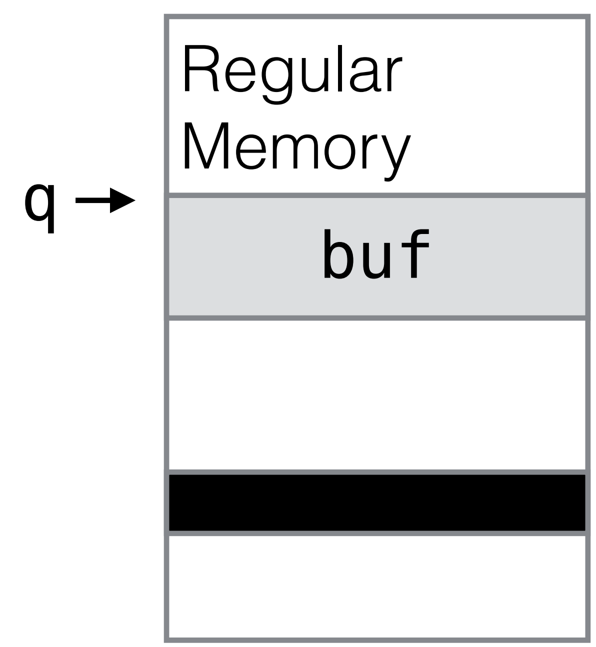
Code-Pointer Separation

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

- Type-based static analysis
- Move only code pointers to safe memory
  - ➡ Isolate safe memory on instruction level
- Keep memory layout unchanged



2.5%
memory accesses

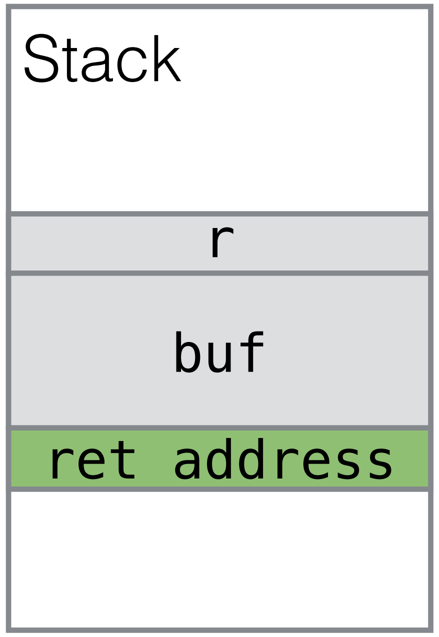


97.5%
memory accesses

Safestack

Safestack

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



Safestack

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

- Split into regular and safe stack

Safestack

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

- Split into regular and safe stack
- Statical check during compile which objects are safe

Safestack

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

- Split into regular and safe stack
- Statical check during compile which objects are safe

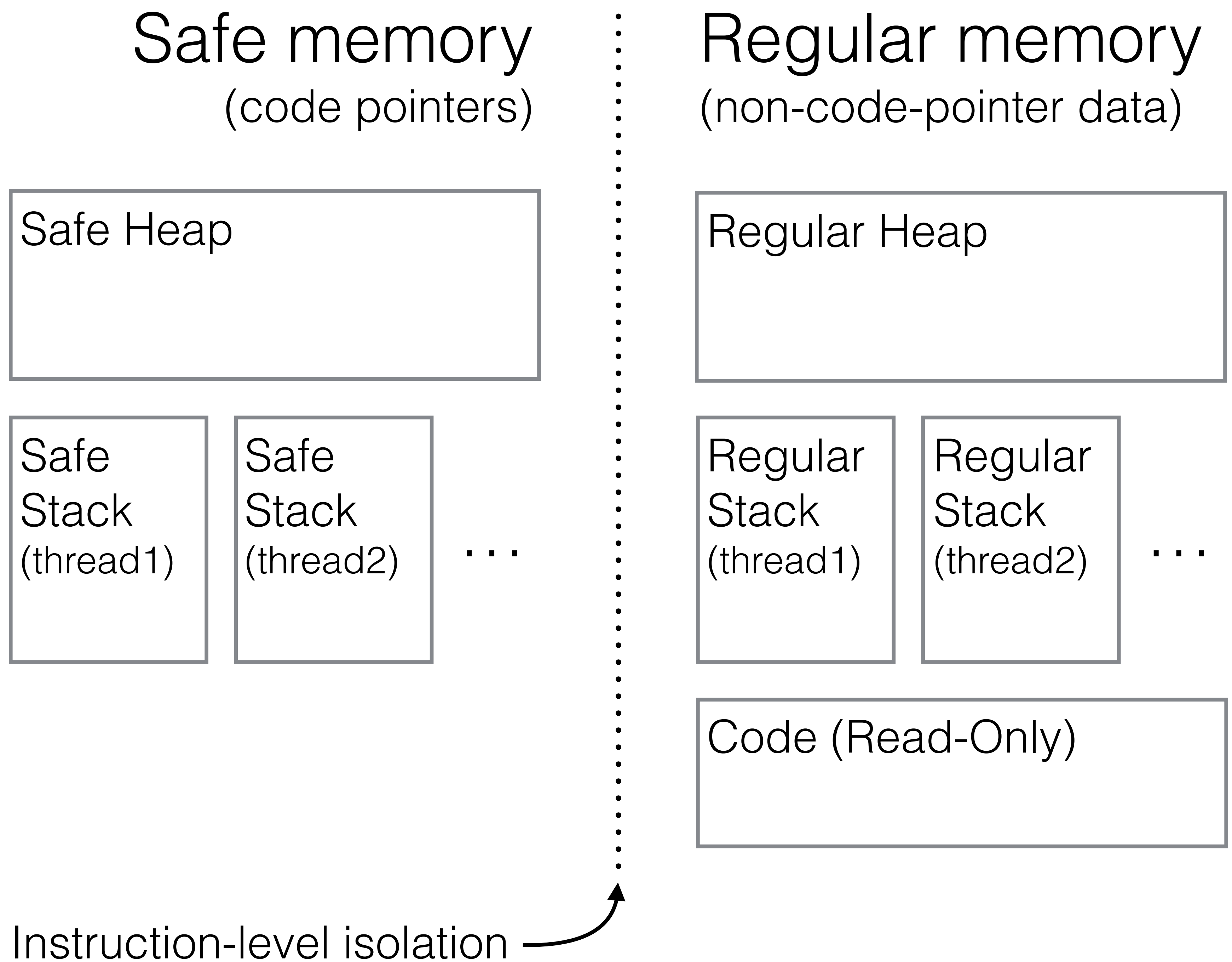
Safestack

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

- Split into regular and safe stack
- Statical check during compile which objects are safe
- Only keep unsafe objects on the regular stack (e.g. arrays)

CPS Memory Layout

CPS Memory Layout



Code-Pointer Integrity

Code-Pointer Integrity

Protecting only code pointers is not enough:

Code-Pointer Integrity

<u>Protecting only code pointers is not enough:</u>

```
int *q = p + input;
*q = input2;
…

(*func_ptr)();
```

Code-Pointer Integrity

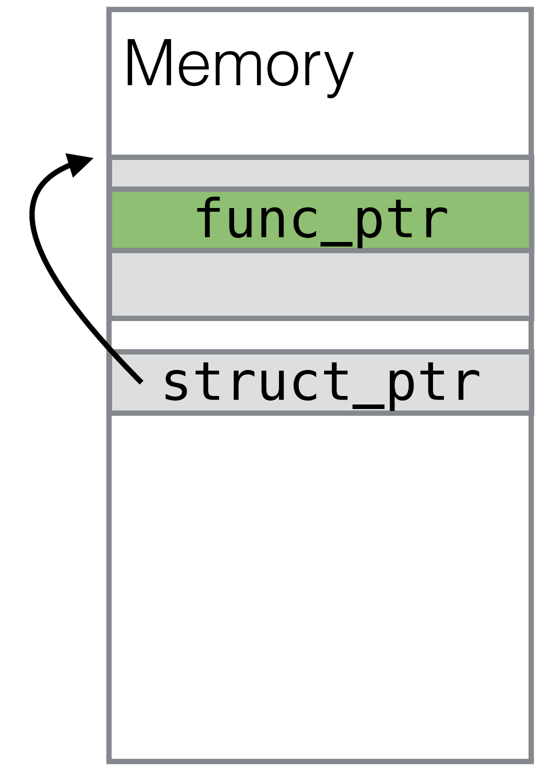
Protecting only code pointers is not enough:

```
int *q = p + input;
*q = input2;
…

(*func_ptr)();
```

Code-Pointer Integrity

Protecting only code pointers is not enough:

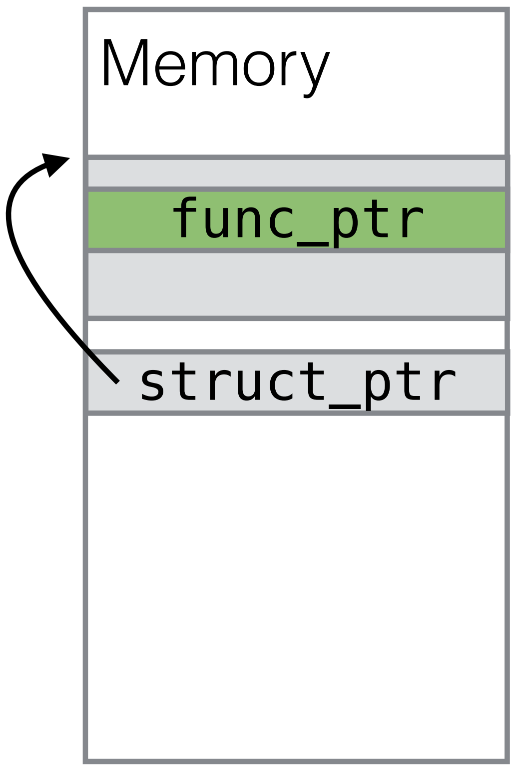
```
int *q = p + input;
*q = input2;
…

(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

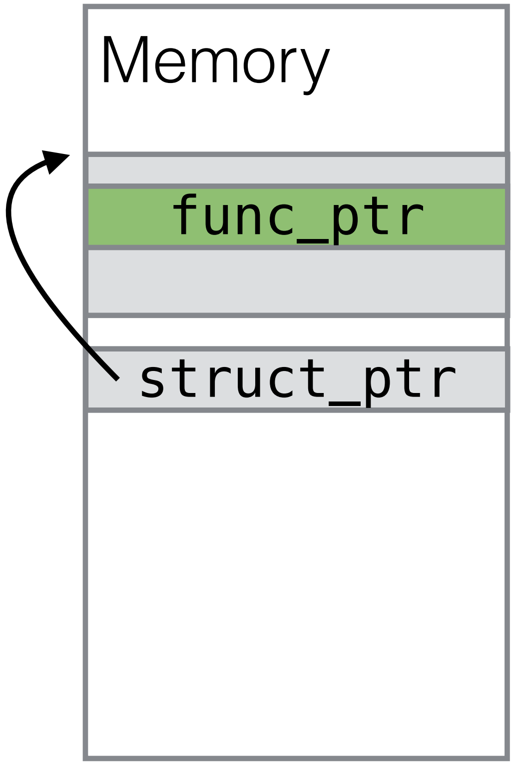
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

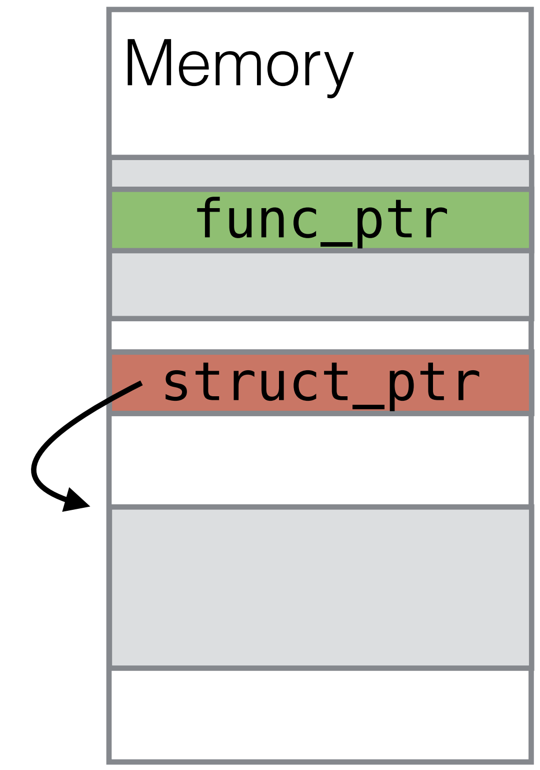
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

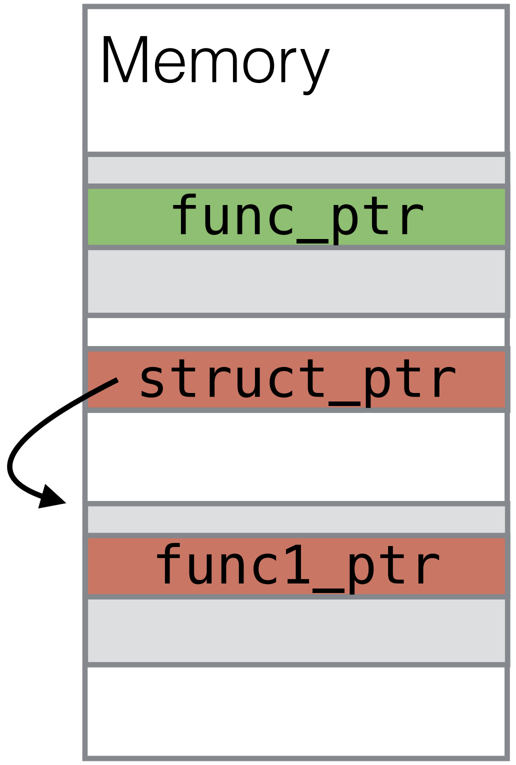
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

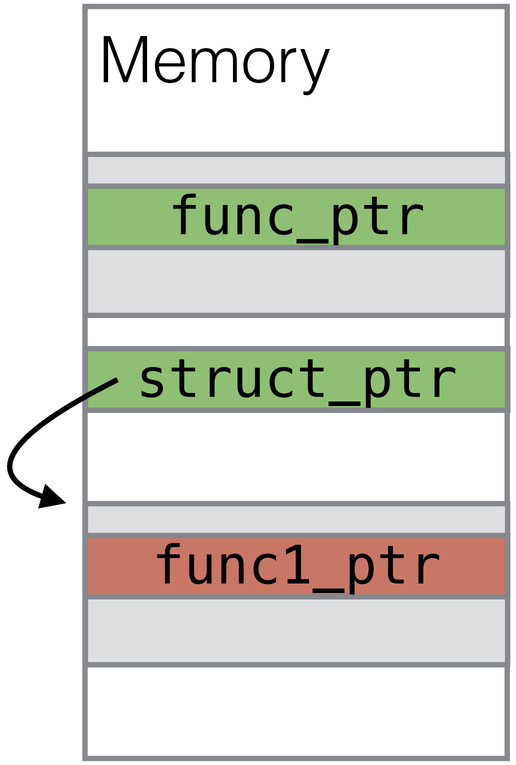
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

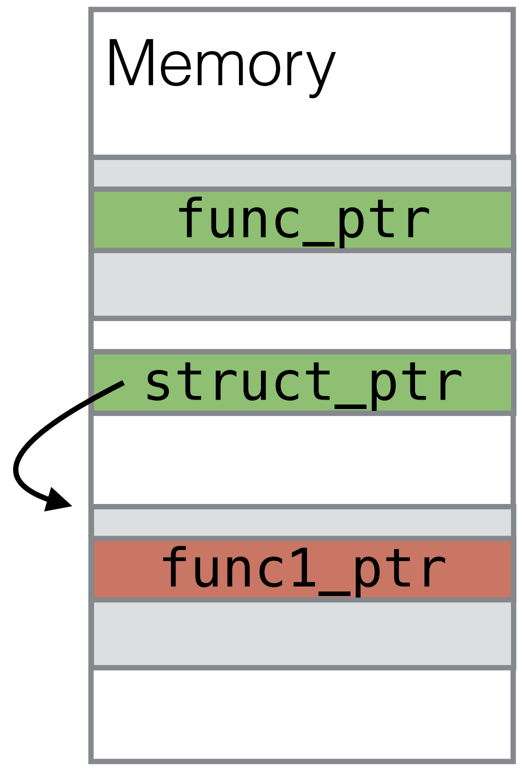
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



Code-Pointer Integrity

Protecting only code pointers is not enough:

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

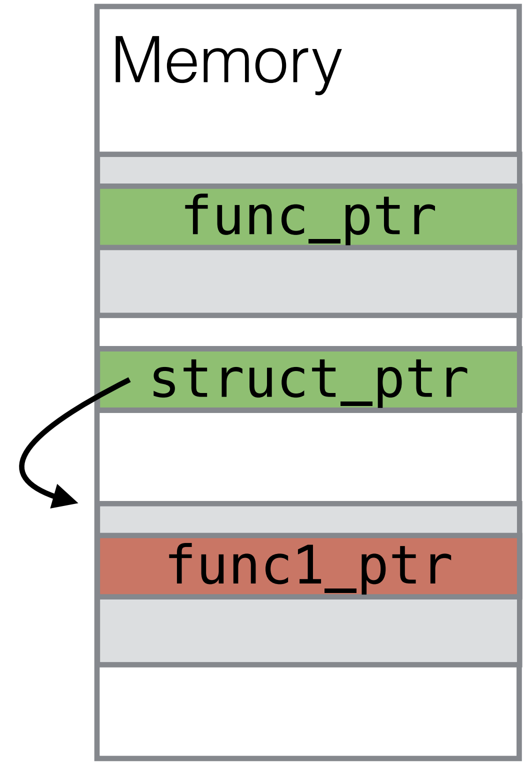


➡ **Indirect Pointers have to be protected as well**

Code-Pointer Integrity

Protecting only code pointers is not enough:

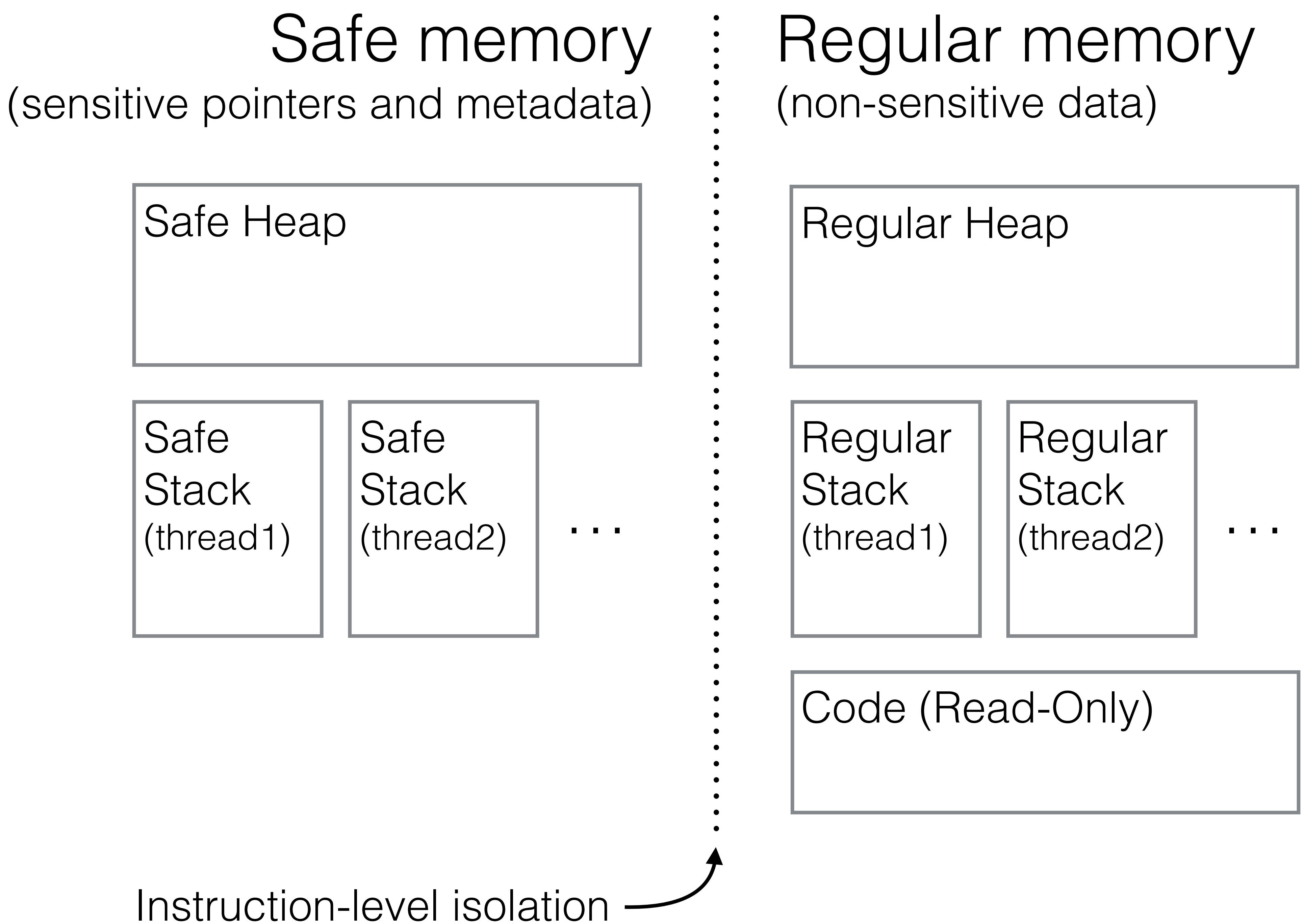
```
int *q = p + input;
*q = input2;
…
func_ptr = struct_ptr->f;
(*func_ptr)();
```



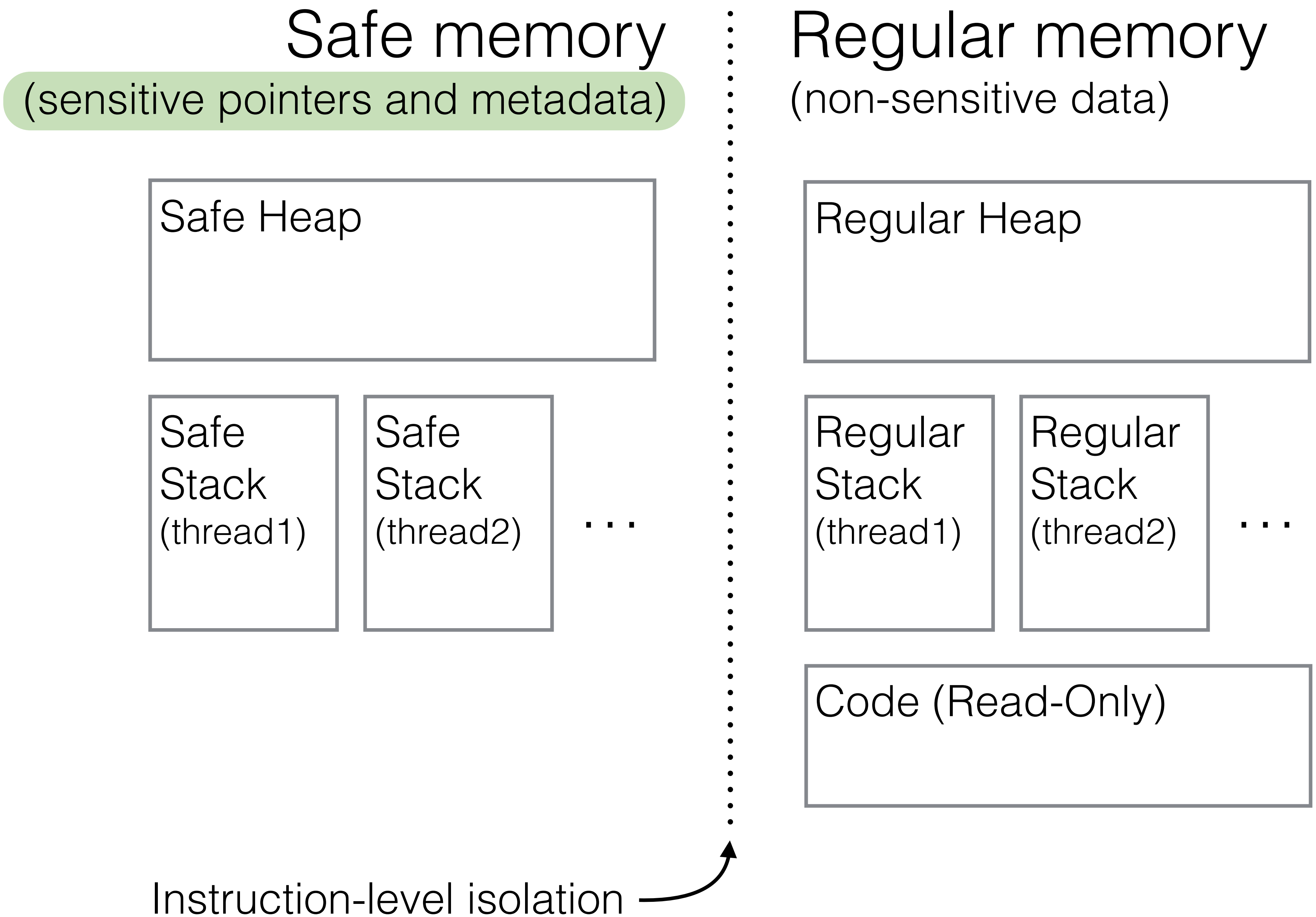
➡ **Indirect Pointers have to be protected as well**

➡ **Extend static analysis to include indirect pointers**

CPI Memory Layout



CPI Memory Layout



Summary

Summary

- CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)

Summary

- CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)
  - ➡ Guaranteed protection against control-flow hijack attacks enabled by memory bugs

Summary

- CPI guarantees memory safety for all sensitive pointers (code pointers and pointers to sensitive pointers)

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

- Keeps performance overhead low by not protecting data pointers

Design

Design

- Static analysis on source code during compilation

Design

- Static analysis on source code during compilation
- Adding safe memory region while keeping the original memory layout intact

Design

- Static analysis on source code during compilation
- Adding safe memory region while keeping the original memory layout intact
- Separating the safe region from the regular region using instruction level protection:
    - Hardware segment protection on x86-32
    - Information hiding on x86-64 and ARM

Security analysis

Security analysis

- CPI and CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)

Security analysis

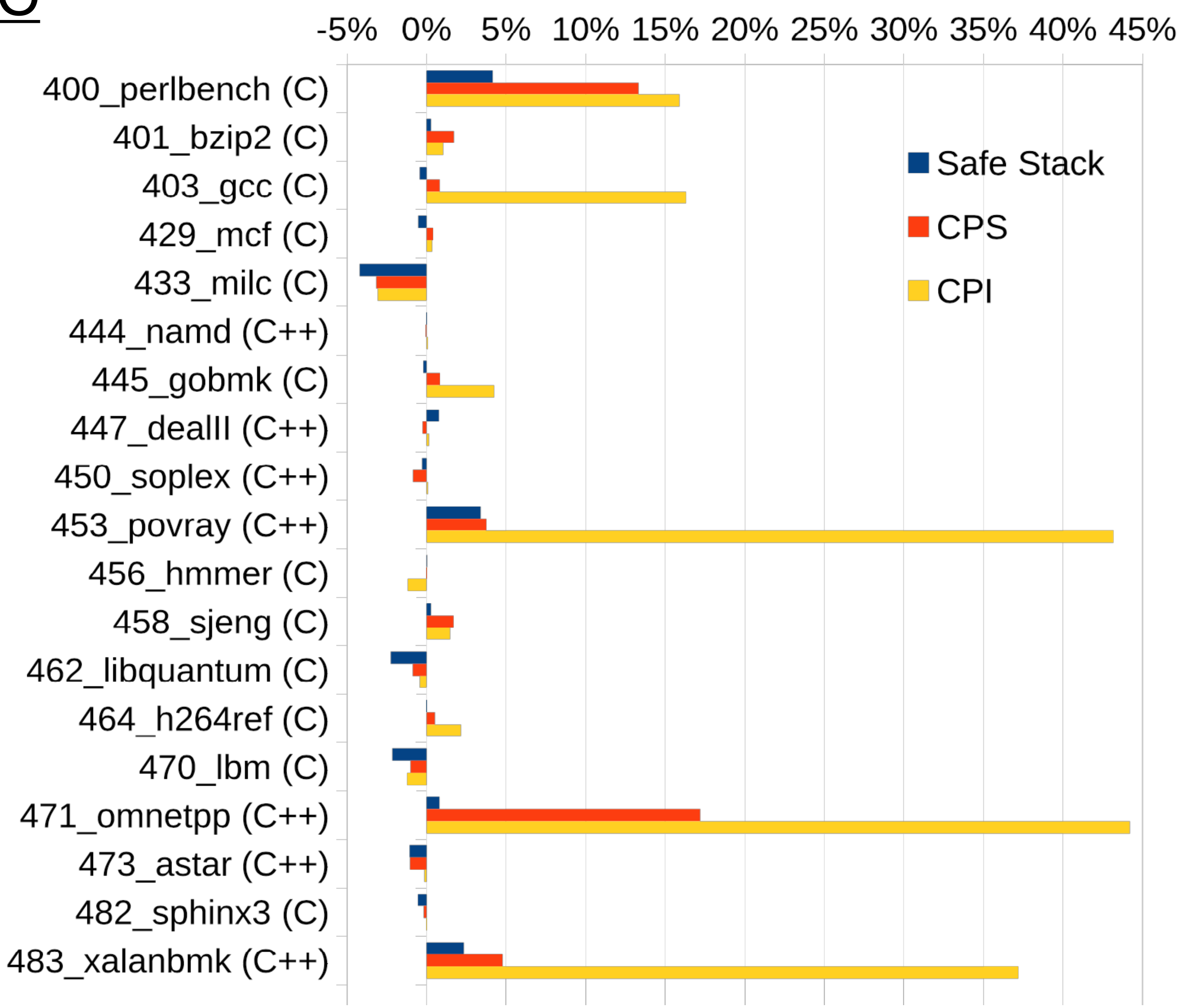
- CPI <u>and</u> CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)
- CPI correctness proof in paper guarantees security against future attacks

Security analysis

- CPI and CPS protect against all attacks from RIPE (Runtime intrusion prevention evaluator)
- CPI correctness proof in paper guarantees security against future attacks
- Does not protect against data-only attacks

Performance Benchmark

Performance in SPEC CPU2006:



Performance summary

	Safe Stack	CPS	CPI
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	-0.4%	1.2%	2.9%
Median (C only)	-0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

Performance numbers from SPEC CPU2006 Benchmark

Security Weakness on x64 and ARM

Security Weakness on x64 and ARM

- Original Paper:

  ➡ Information hiding is secure because no pointer to the safe region exists in unsafe memory

Security Weakness on x64 and ARM

- Original Paper:

  ➡ Information hiding is secure because no pointer to the safe region exists in unsafe memory

- Paper by Evans et. al.:

  ➡ Shows there is a way to find the safe area using side channel attack

Information Hiding Implementation

Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in of the segment registers provided by x64

Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in of the segment registers provided by x64

➡ No pointer to the safe region exists in regular memory

Information Hiding Implementation

1) Randomly choose an address to serve as base address for safe memory region

2) Store address in of the segment registers provided by x64

➡ No pointer to the safe region exists in regular memory

➡ 48 bit address space in x64 CPU makes guessing impractical, most guesses would cause crashing

Attack Description

Attack Description

1) Timing Side-channel Attack

Attack Description

1) Timing Side-channel Attack

2) Data Collection

Attack Description

1) Timing Side-channel Attack

2) Data Collection

3) Locate Safe Region

Attack Description

1) Timing Side-channel Attack

2) Data Collection

3) Locate Safe Region

4) Attack Safe Region

Mitigation of the Weakness

- Implement Hardware Segmentation in x86-64
- Switch to software fault isolation

  ➡ Introduces additional performance overhead of ~5%
- Reduce feasibility of side channel attack by changing implementation of information hiding

  ➡ Replace linear table with hash table or two-level lookup table

Discussion

Questions?

References:

- Code-Pointer Integrity - Kuznetsov et. al. (2014)
- Presentation: Code-Pointer Integrity - Kuznetsov (OSDI 2014)
- Missing the Point(er) - Evans et. al. (2015)
- Getting the Point(er) - Kuznetsov et. al. (2015)