Preventing control-flow hijacks with Code Pointer Integrity

László Szekeres

Stony Brook University

Joint work with Volodymyr Kuznetsov, Mathias Payer, George Candea, R. Sekar, Dawn Song

Problem

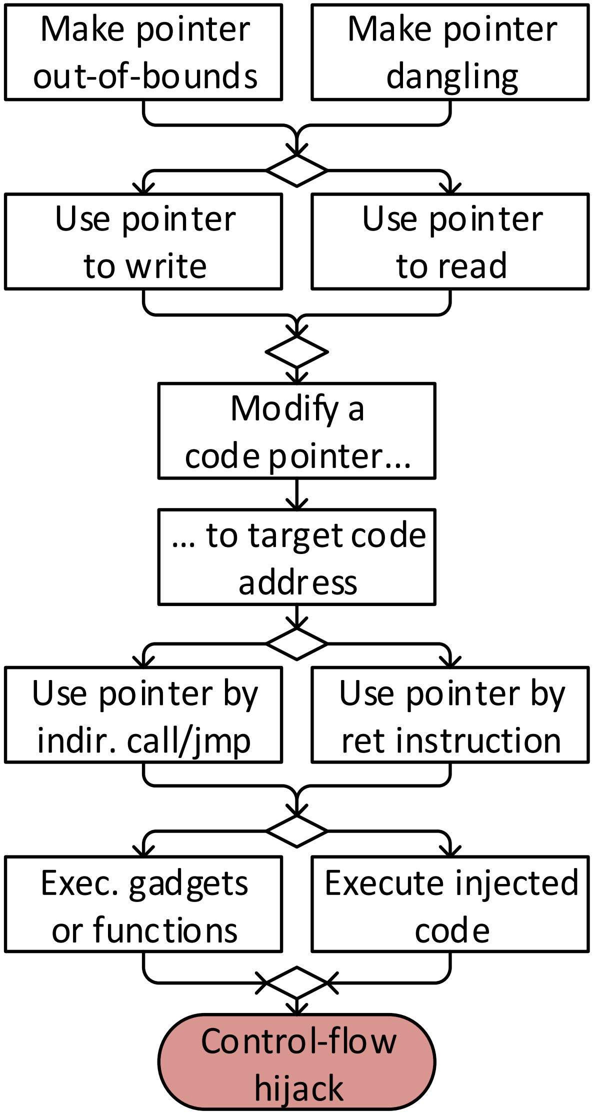
- C/C++ is unsafe and unavoidable today
- All of our systems have C/C++ parts
- All of them have exploitable vulnerabilities
- They all can be compromised

PWNED!
Pwn2Own 2013
PWNED!
PWNED!

1:30 - Java (James Forshaw) PWNED
2:30 - Java (Joshua Drake) PWNED
3:30 - IE 10 (VUPEN Security) PWNED
4:30 - Chrome (Nils & Jon) PWNED
5:30 - Firefox (VUPEN Security) PWNED
5:31 - Java (VUPEN Security) PWNED

Control-flow hijack attack

[Eternal War in Memory, IEEE S&P '13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

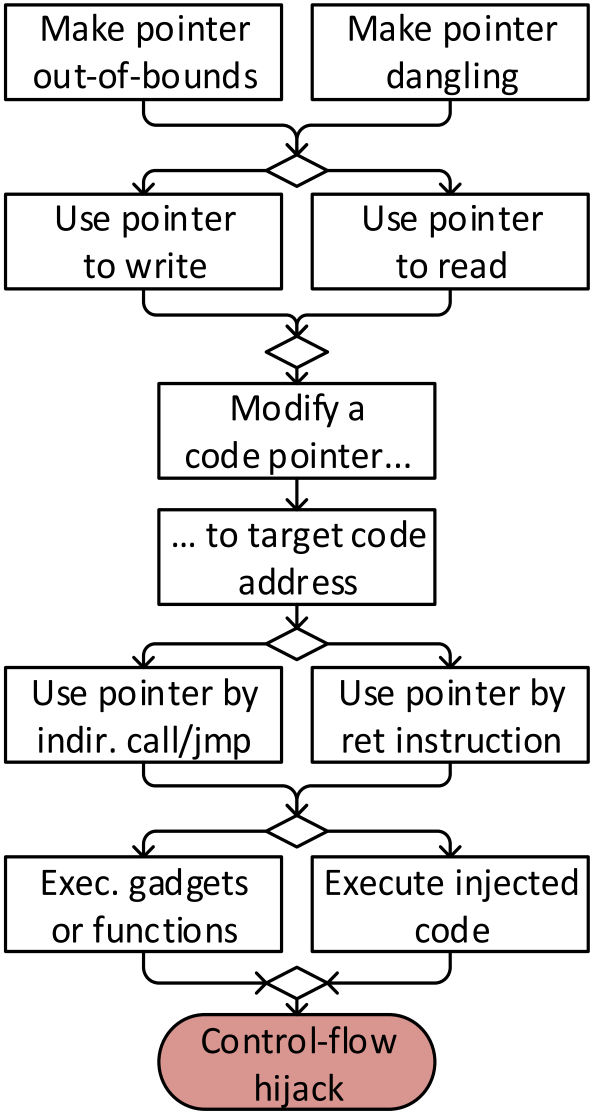
Exec. gadgets or functions

Execute injected code

Control-flow hijack

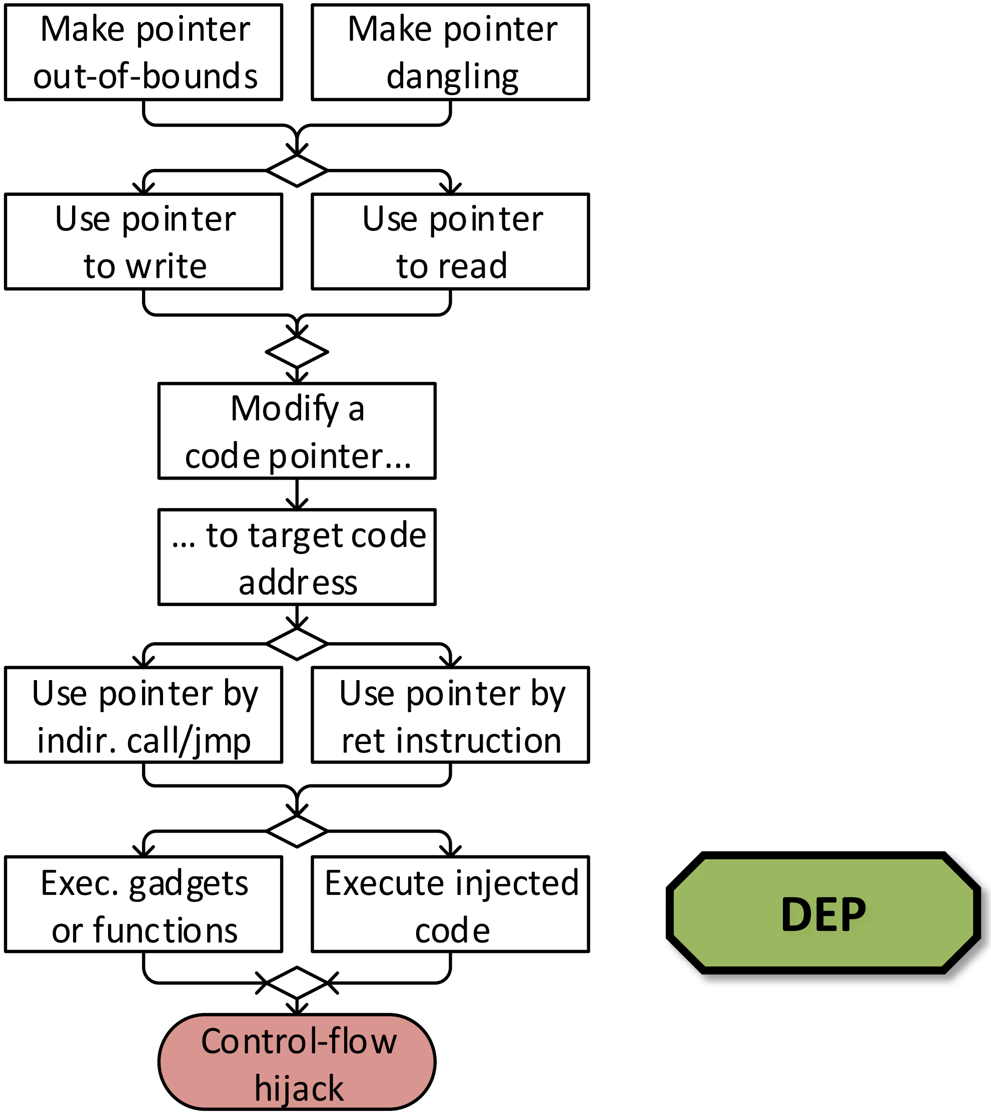
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P '13]


Make pointer out-of-bounds
Make pointer dangling
Use pointer to write
Use pointer to read
Modify a code pointer...
... to target code address
Use pointer by indir. call/jmp
Use pointer by ret instruction
Exec. gadgets or functions
Execute injected code
Control-flow hijack

Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

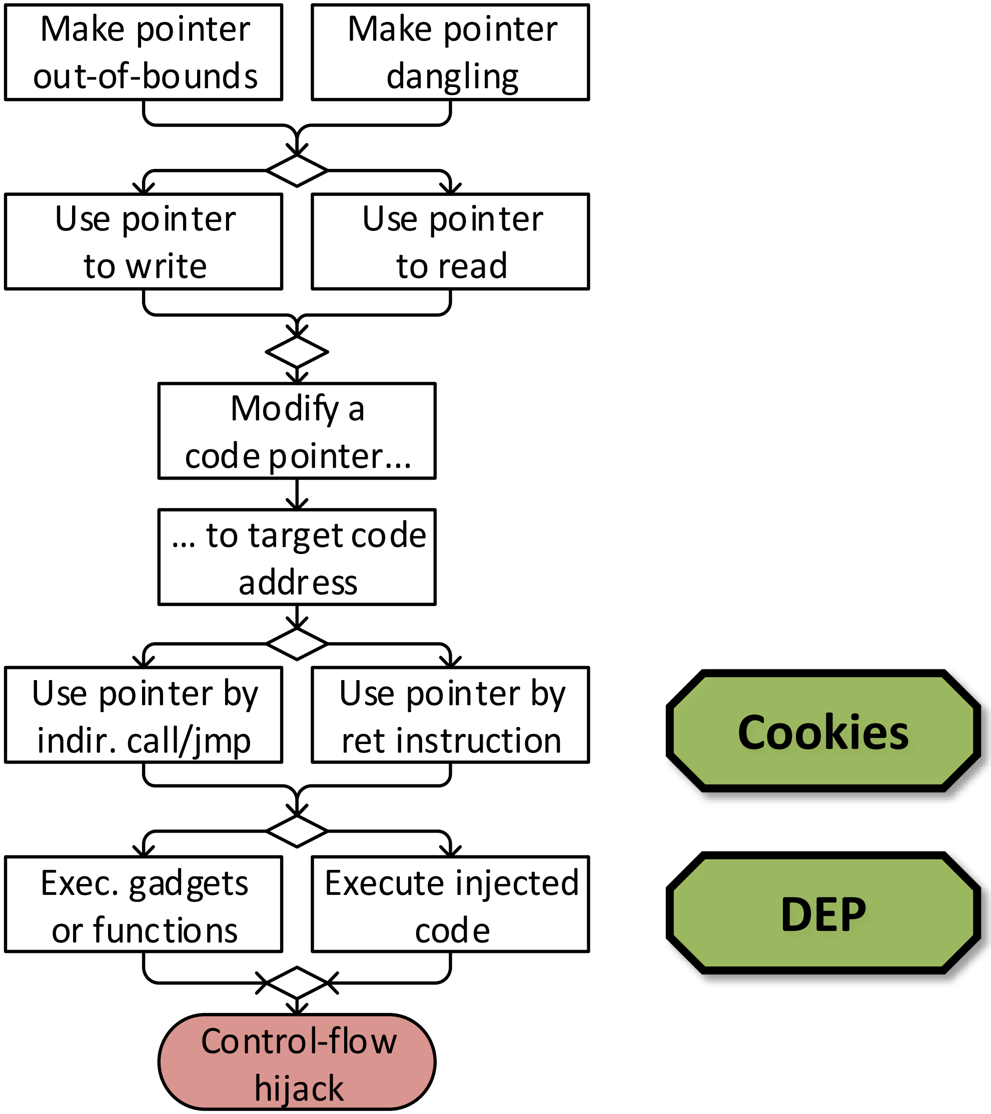
Execute injected code

Control-flow hijack

DEP

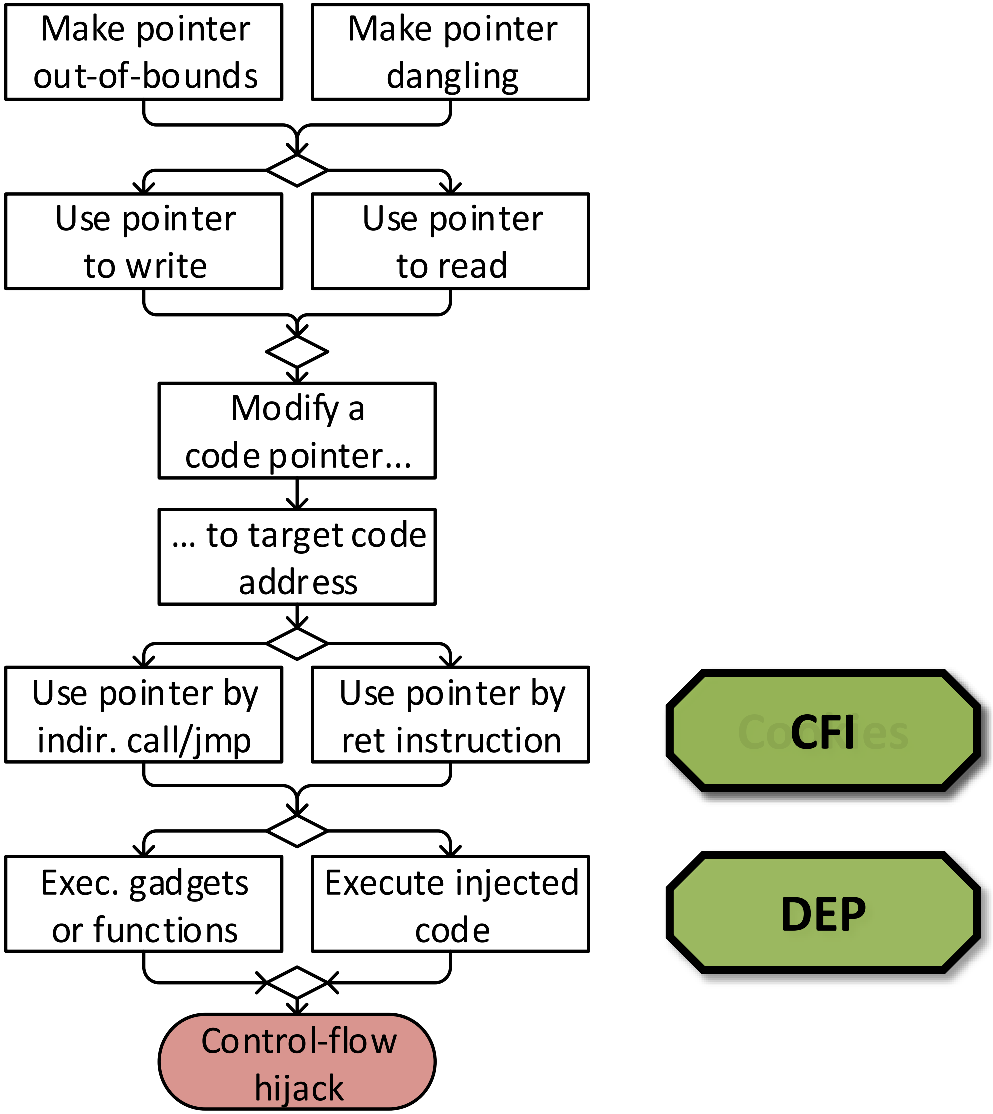
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P '13]



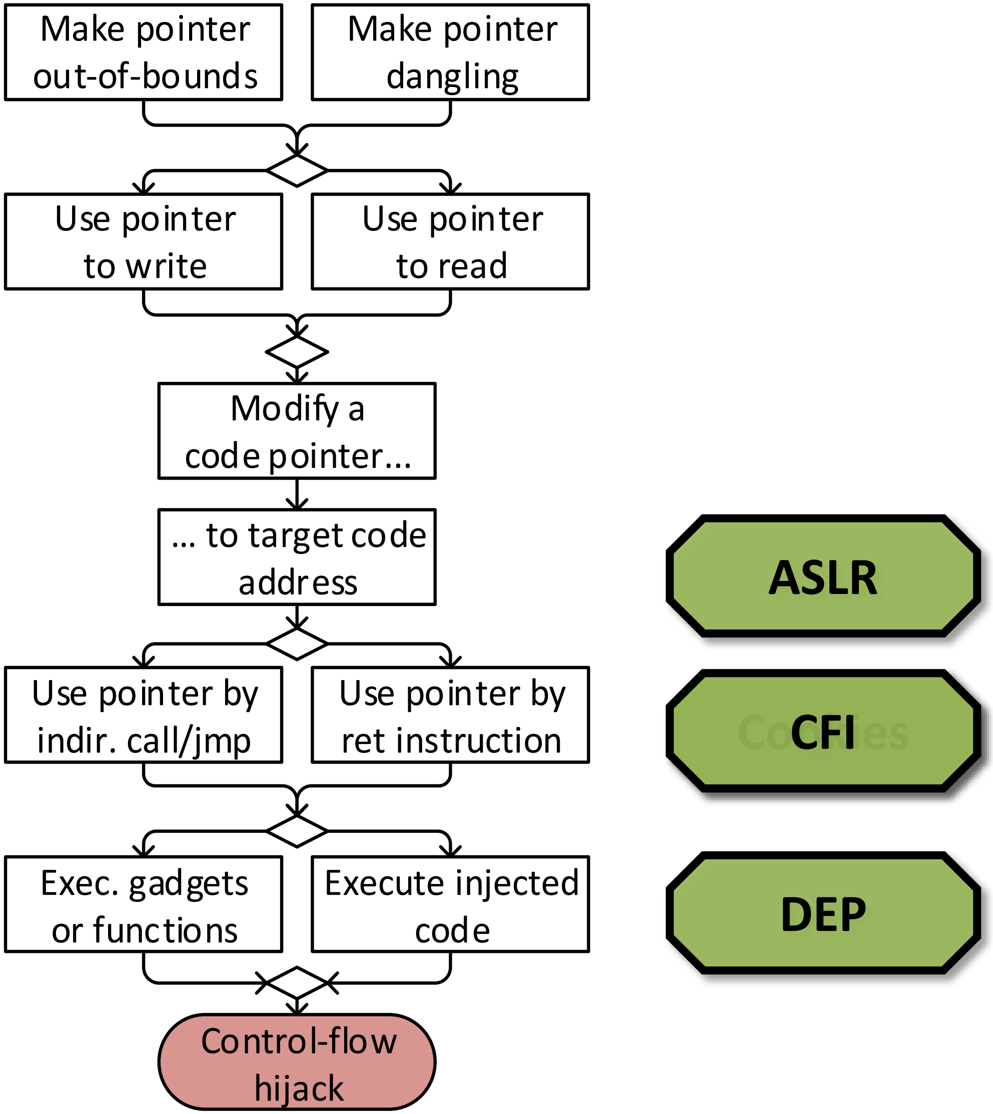
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]



Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

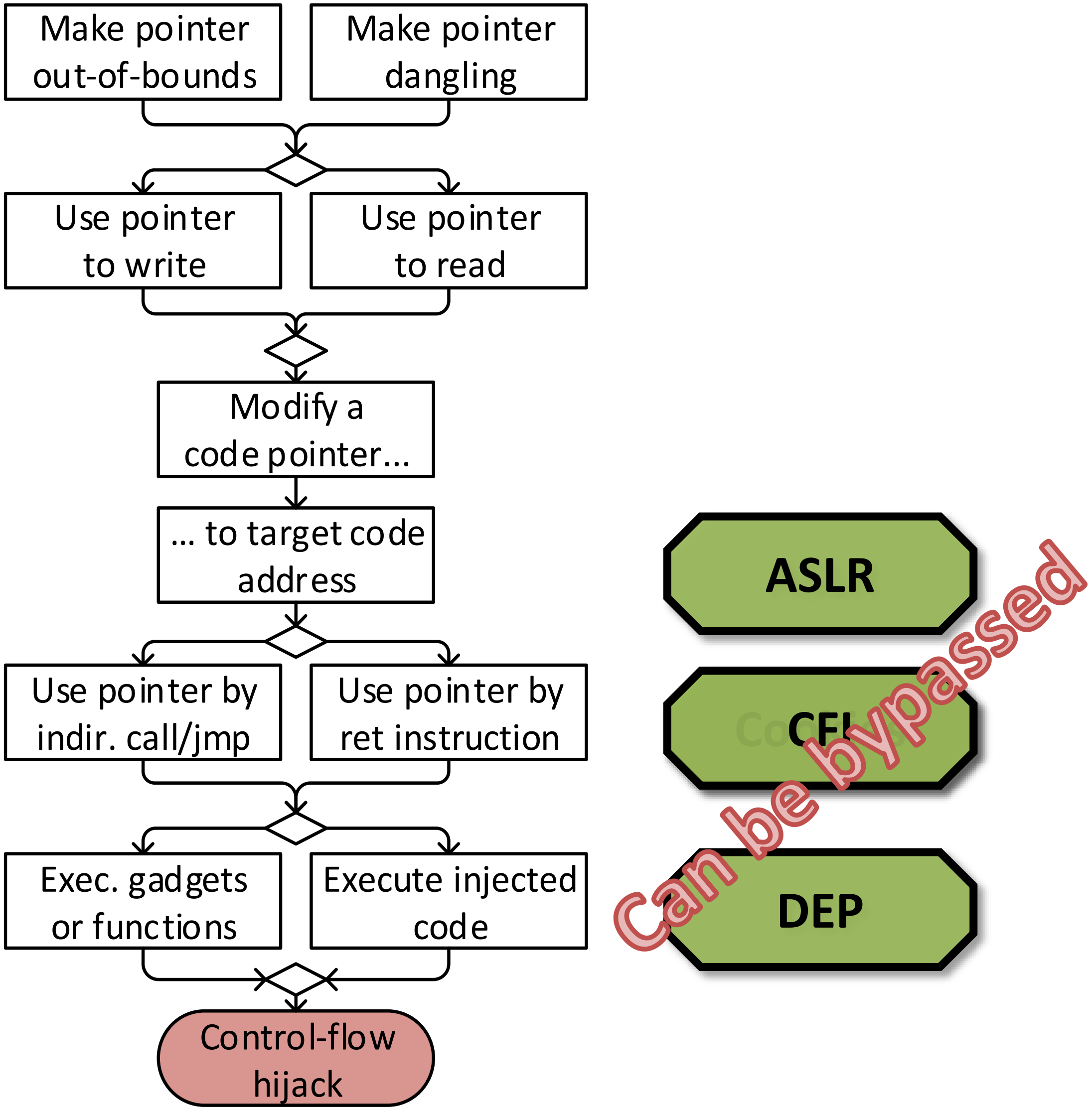
ASLR

CFI

DEP

Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P '13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

ASLR

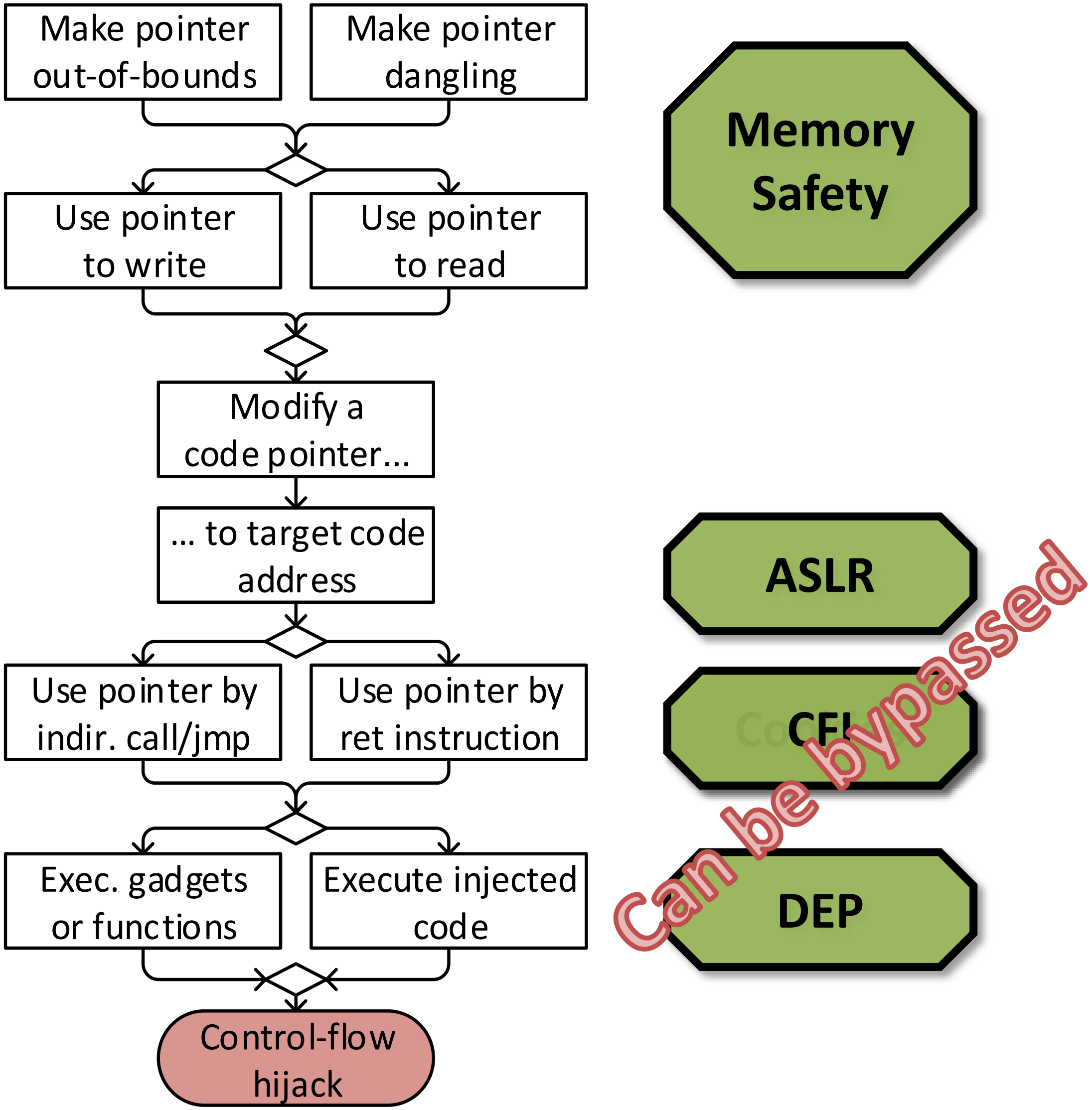
CFI

DEP

Can be bypassed

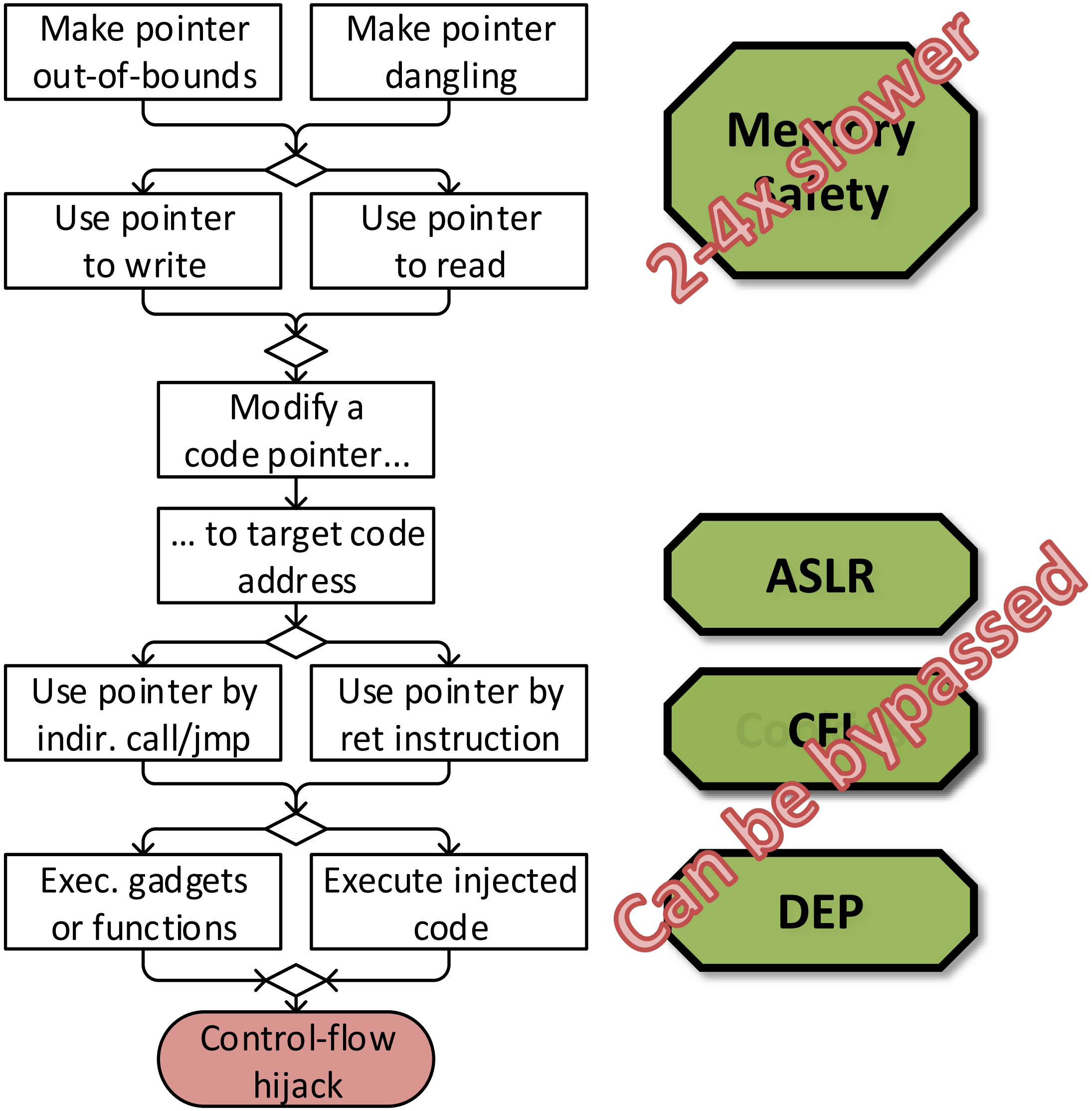
Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]



Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P '13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

Memory Safety

2-4x slower

ASLR

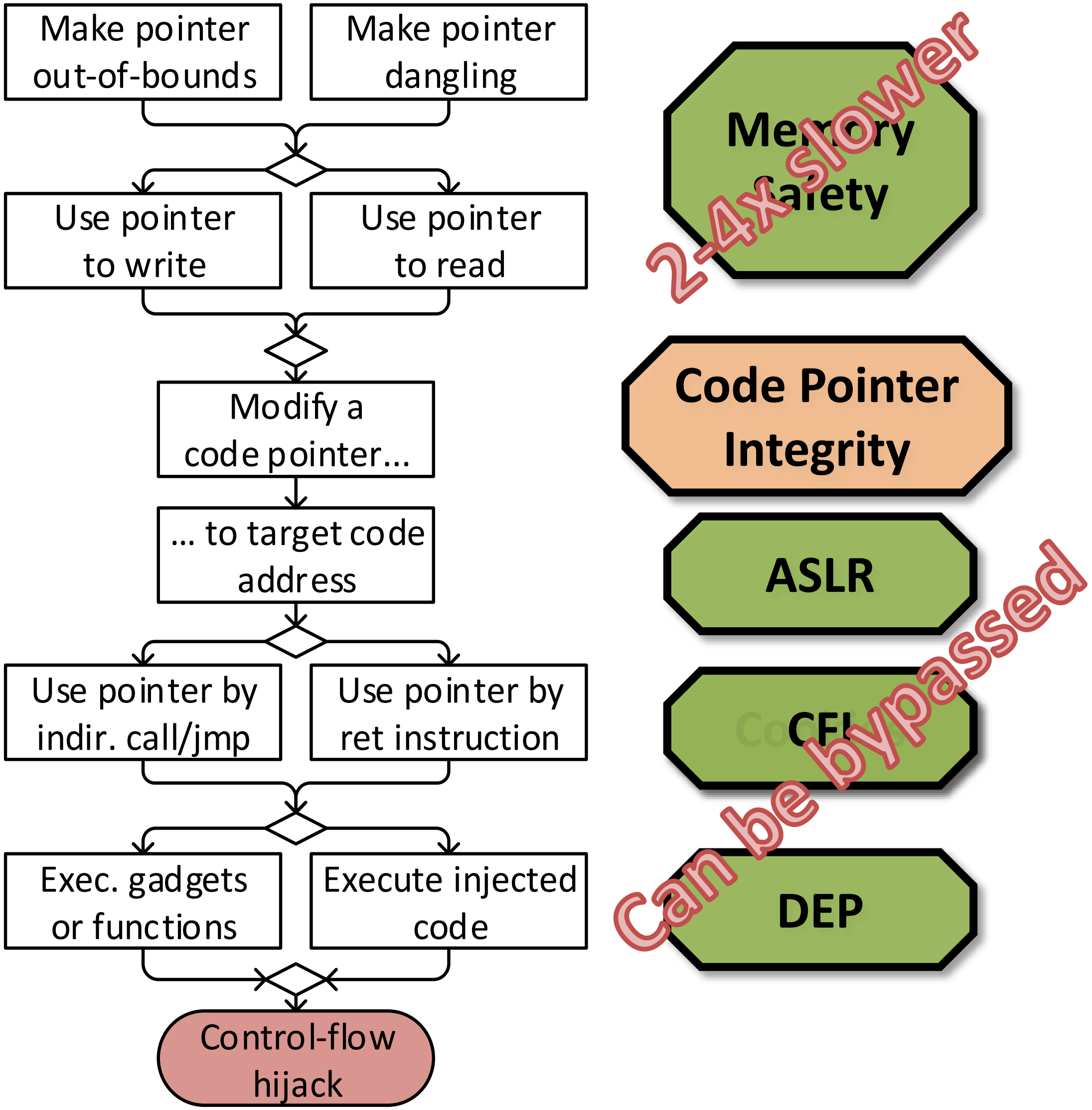
CFI

DEP

Can be bypassed

Control-flow hijack defenses

[Eternal War in Memory, IEEE S&P ‘13]



Make pointer out-of-bounds

Make pointer dangling

Use pointer to write

Use pointer to read

Modify a code pointer…

… to target code address

Use pointer by indir. call/jmp

Use pointer by ret instruction

Exec. gadgets or functions

Execute injected code

Control-flow hijack

Memory Safety

2-4x slower

Code Pointer Integrity

ASLR

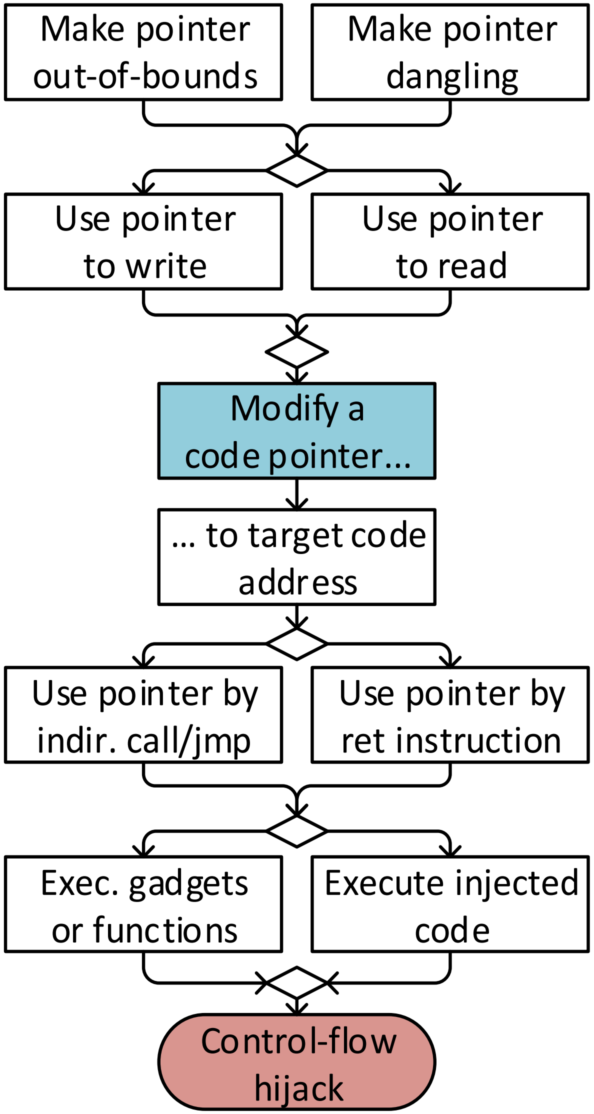
CFI

DEP

Can be bypassed

Code Pointer Integrity?

[Eternal War in Memory, IEEE S&P '13]



Code Pointer Integrity?

Code Pointer Integrity

[OSDI ’14]

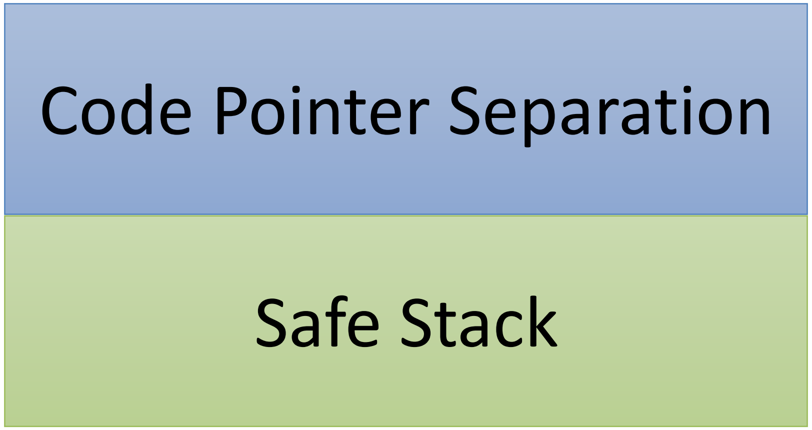
- Joint work with Volodymyr Kuznetsov, Mathias Payer, George Candea, R. Sekar, Dawn Song
- It prevents **all control-flow hijacks**
- It has only **8% runtime overhead** in average

Outline

Outline

Safe Stack

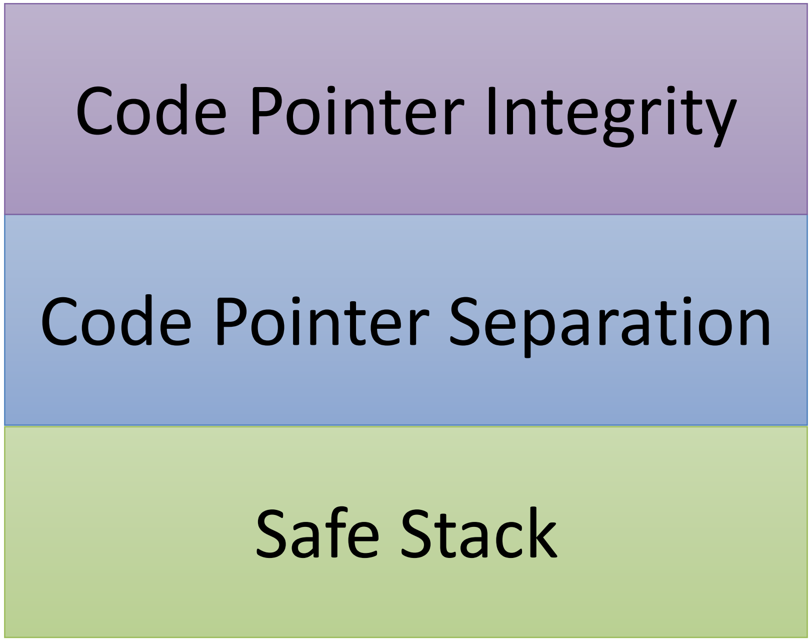
Outline



Code Pointer Separation

Safe Stack

Outline



Code Pointer Integrity

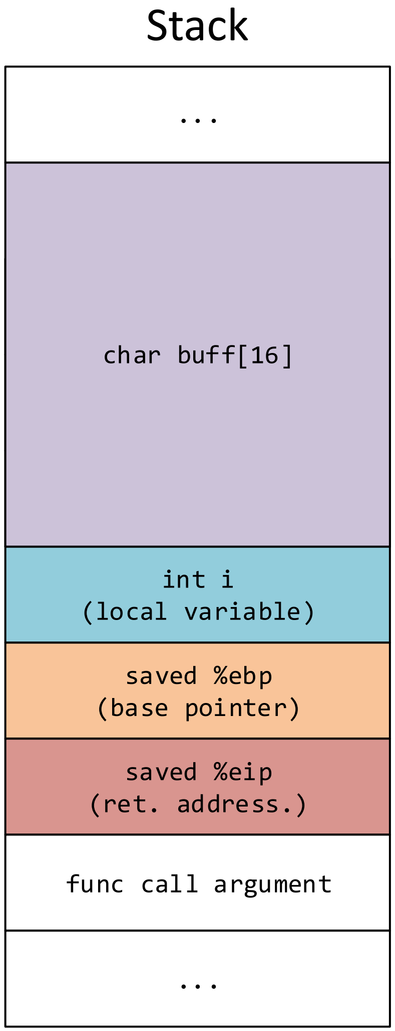
Code Pointer Separation

Safe Stack

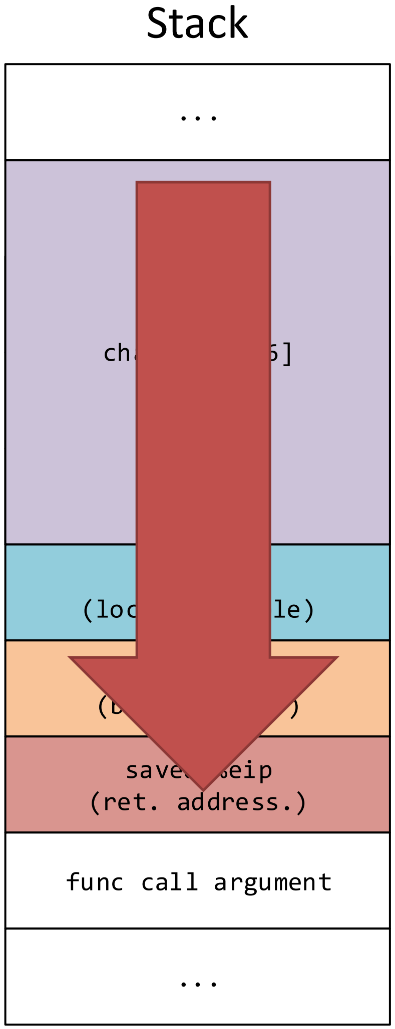
Safe Stack

Enforcing the integrity of return addresses

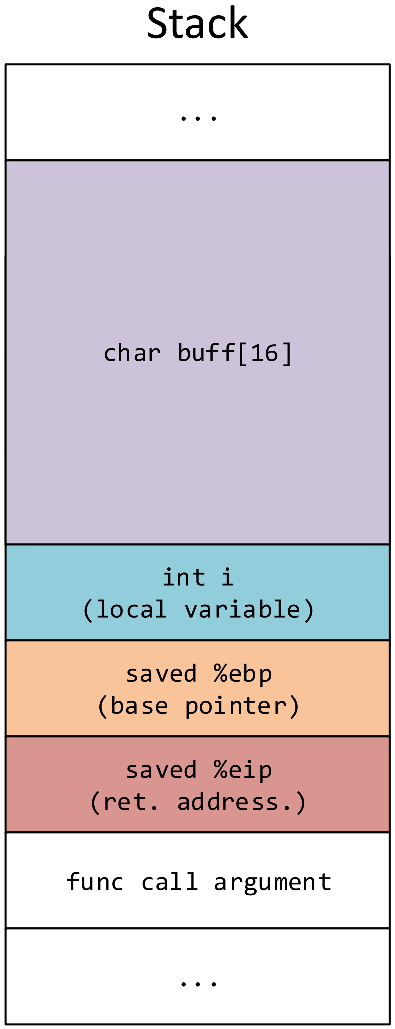
Integrity of return addresses



Integrity of return addresses

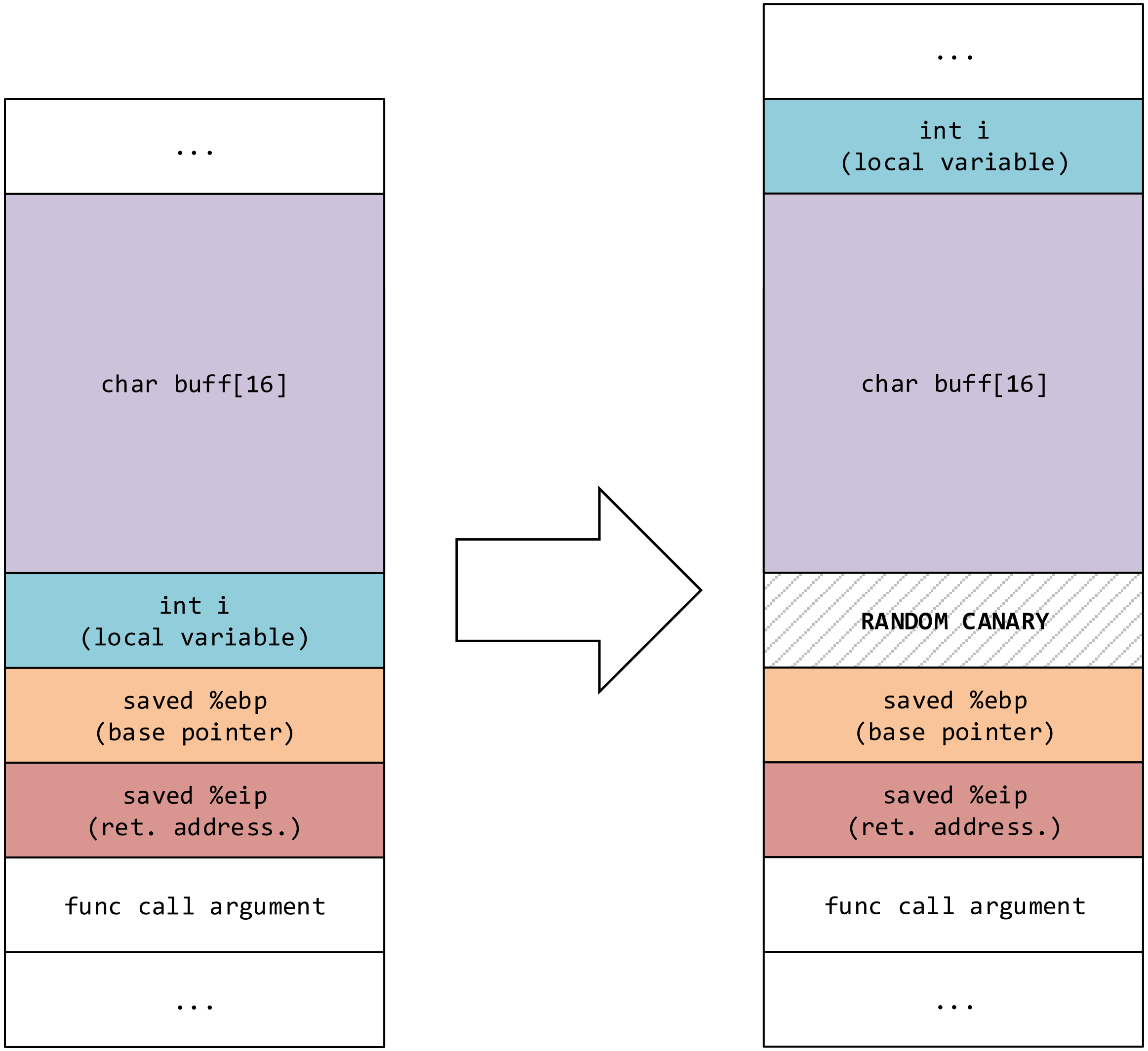


Integrity of return addresses

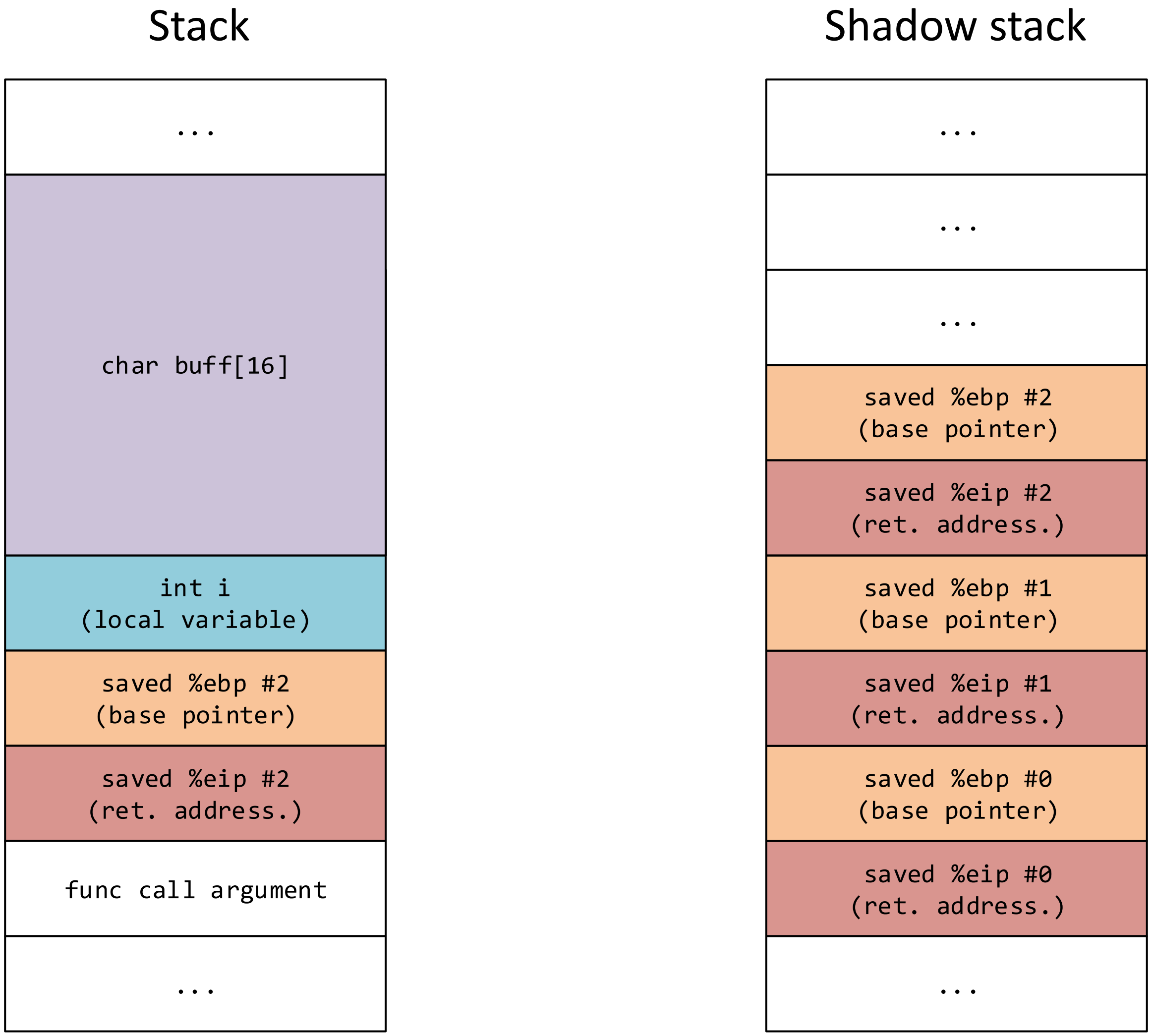


```
p[idx]=val;
```

Stack cookies



Shadow stack



Stack

...

char buff[16]

int i
(local variable)

saved %ebp #2
(base pointer)

saved %eip #2
(ret. address.)

func call argument

...

Shadow stack

...

...

...

saved %ebp #2
(base pointer)

saved %eip #2
(ret. address.)

saved %ebp #1
(base pointer)

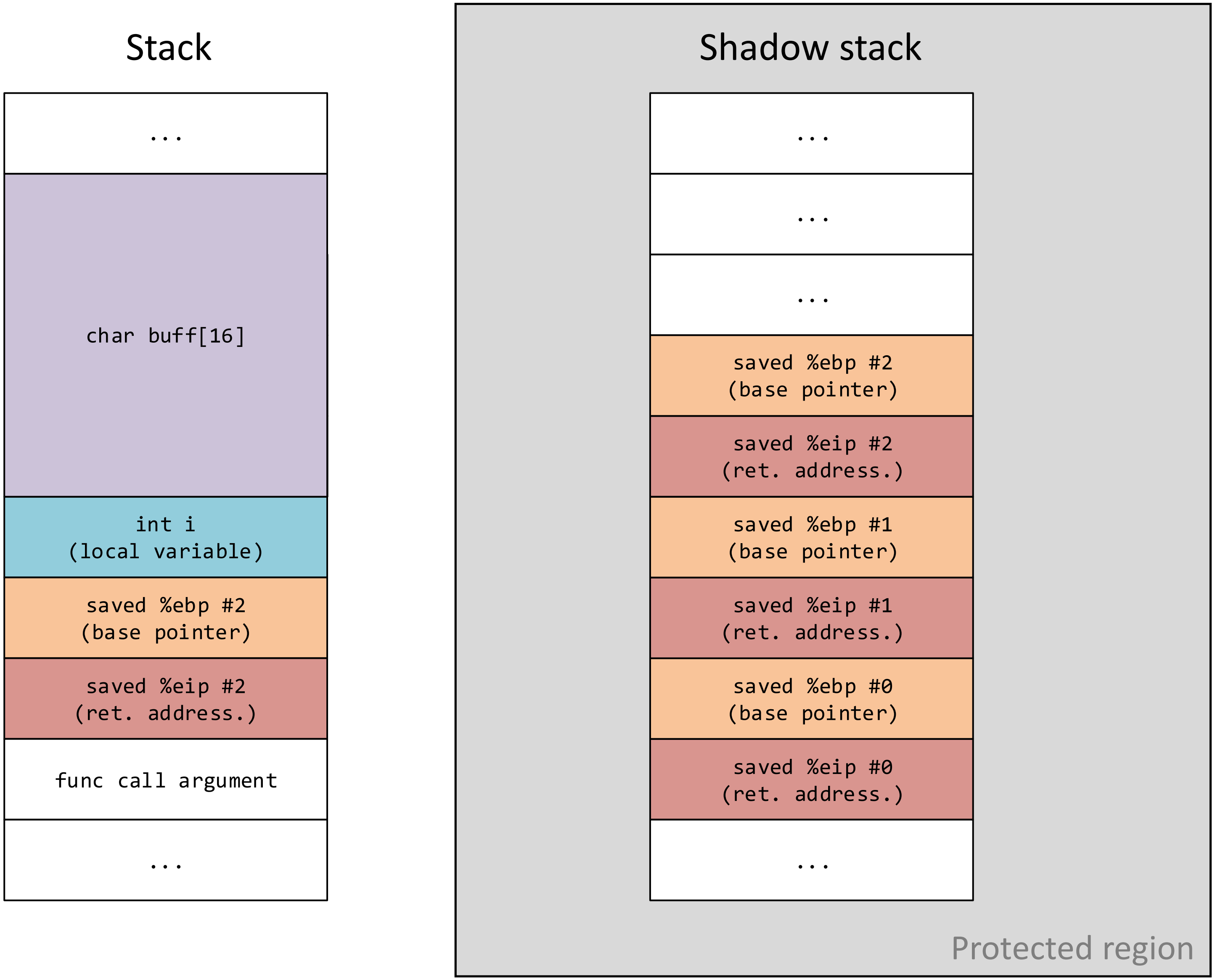
saved %eip #1
(ret. address.)

saved %ebp #0
(base pointer)

saved %eip #0
(ret. address.)

...

Shadow stack



Stack

...

char buff[16]

int i
(local variable)

saved %ebp #2
(base pointer)

saved %eip #2
(ret. address.)

func call argument

...

Shadow stack

...

...

...

saved %ebp #2
(base pointer)

saved %eip #2
(ret. address.)

saved %ebp #1
(base pointer)

saved %eip #1
(ret. address.)

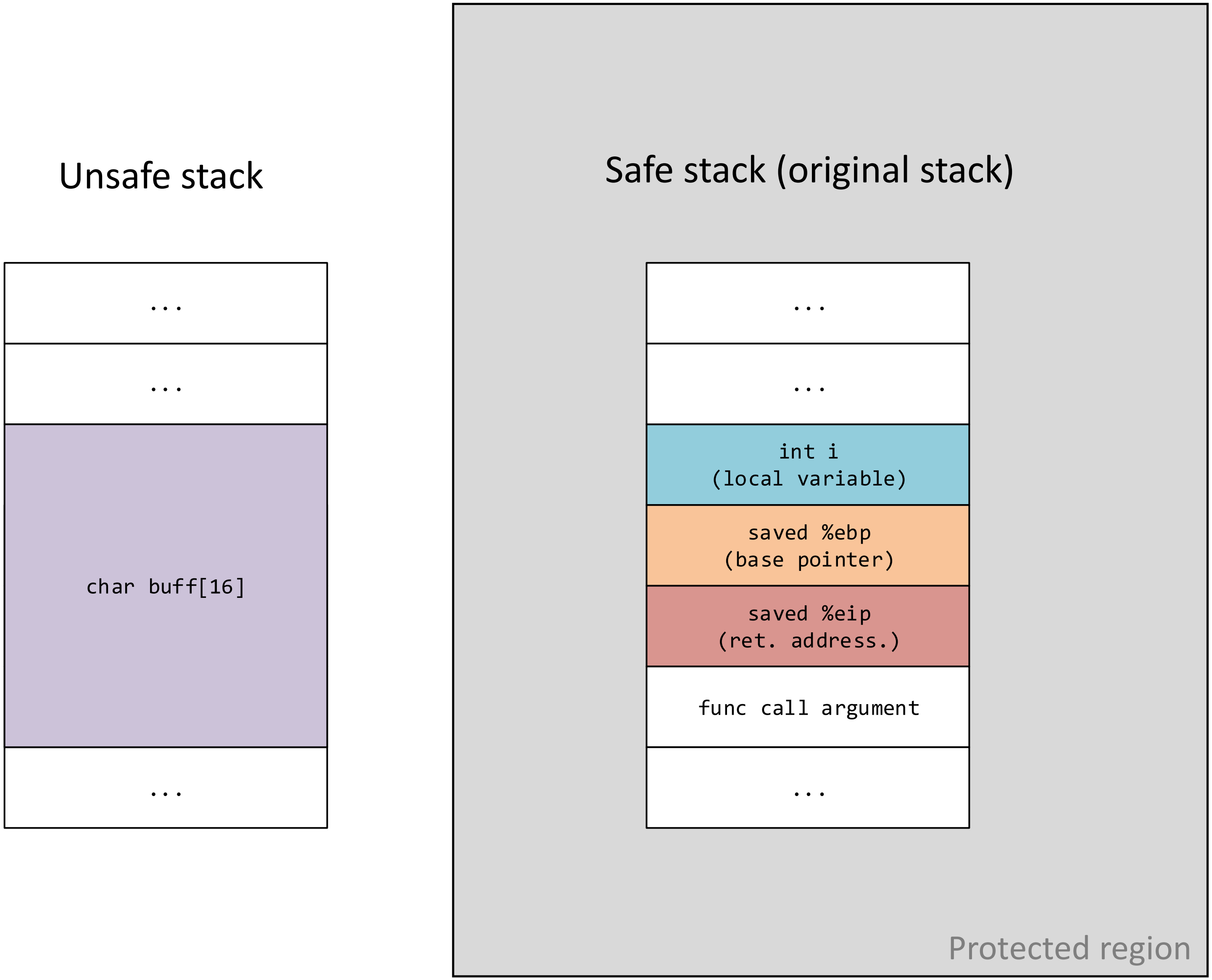
saved %ebp #0
(base pointer)

saved %eip #0
(ret. address.)

...

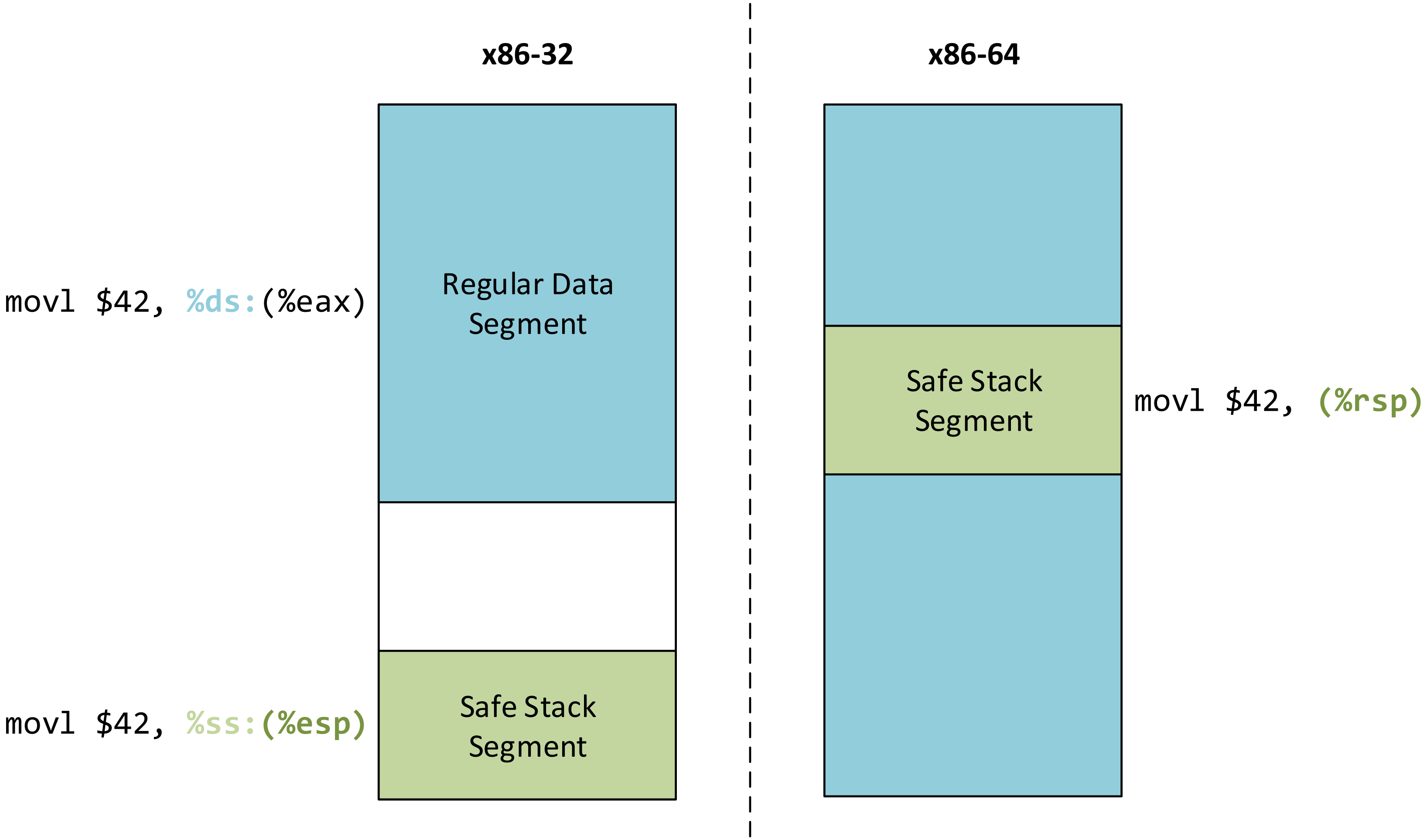
Protected region

Safe Stack



Unsafe stack
...
...
char buff[16]
...
Safe stack (original stack)
...
...
int i
(local variable)
saved %ebp
(base pointer)
saved %eip
(ret. address.)
func call argument
...
Protected region

Protecting the Safe Stack



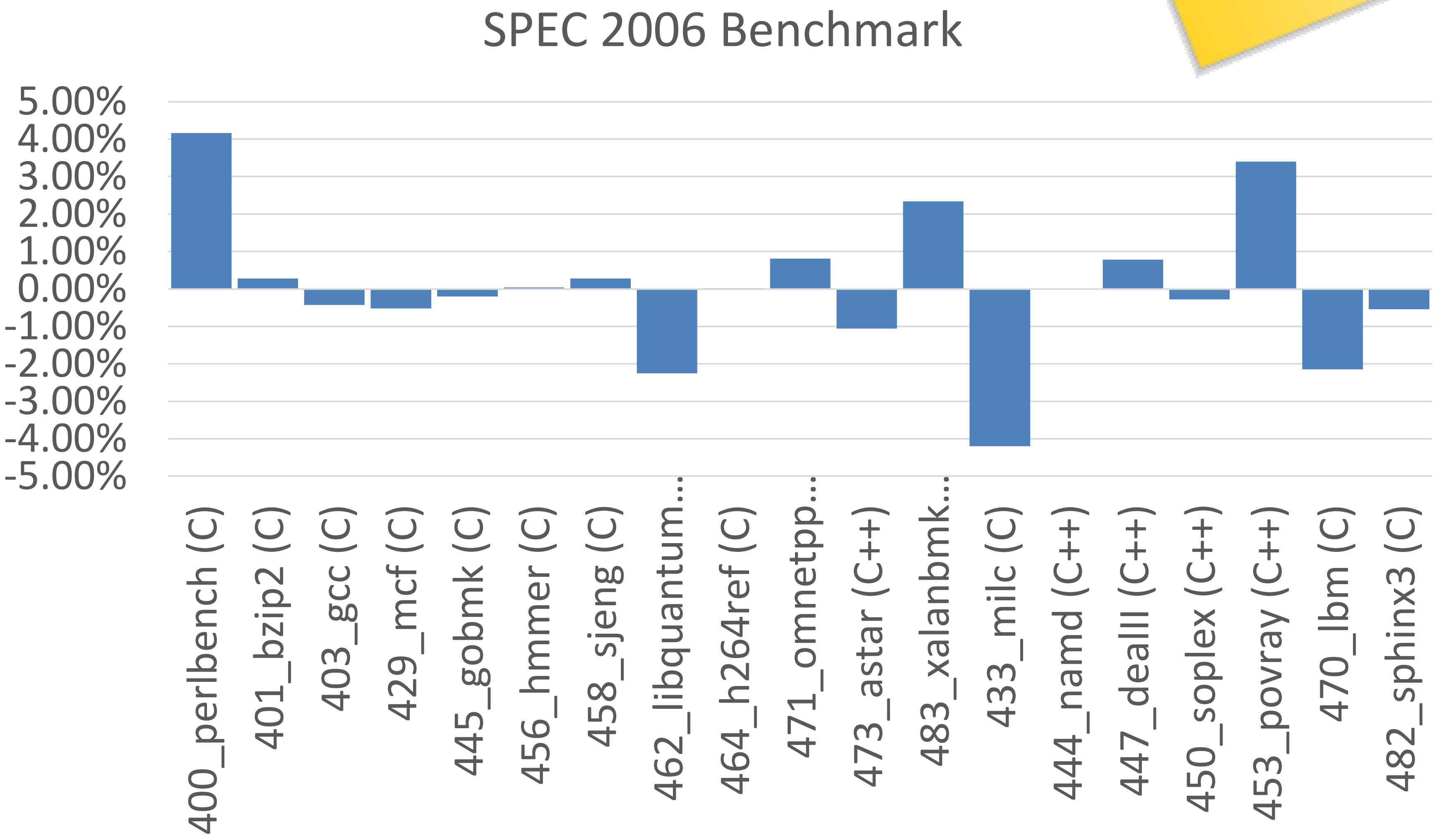
```
movl $42, %ds:(%eax)
movl $42, %ss:(%esp)
movl $42, (%rsp)
```

How effective is the Safe Stack?

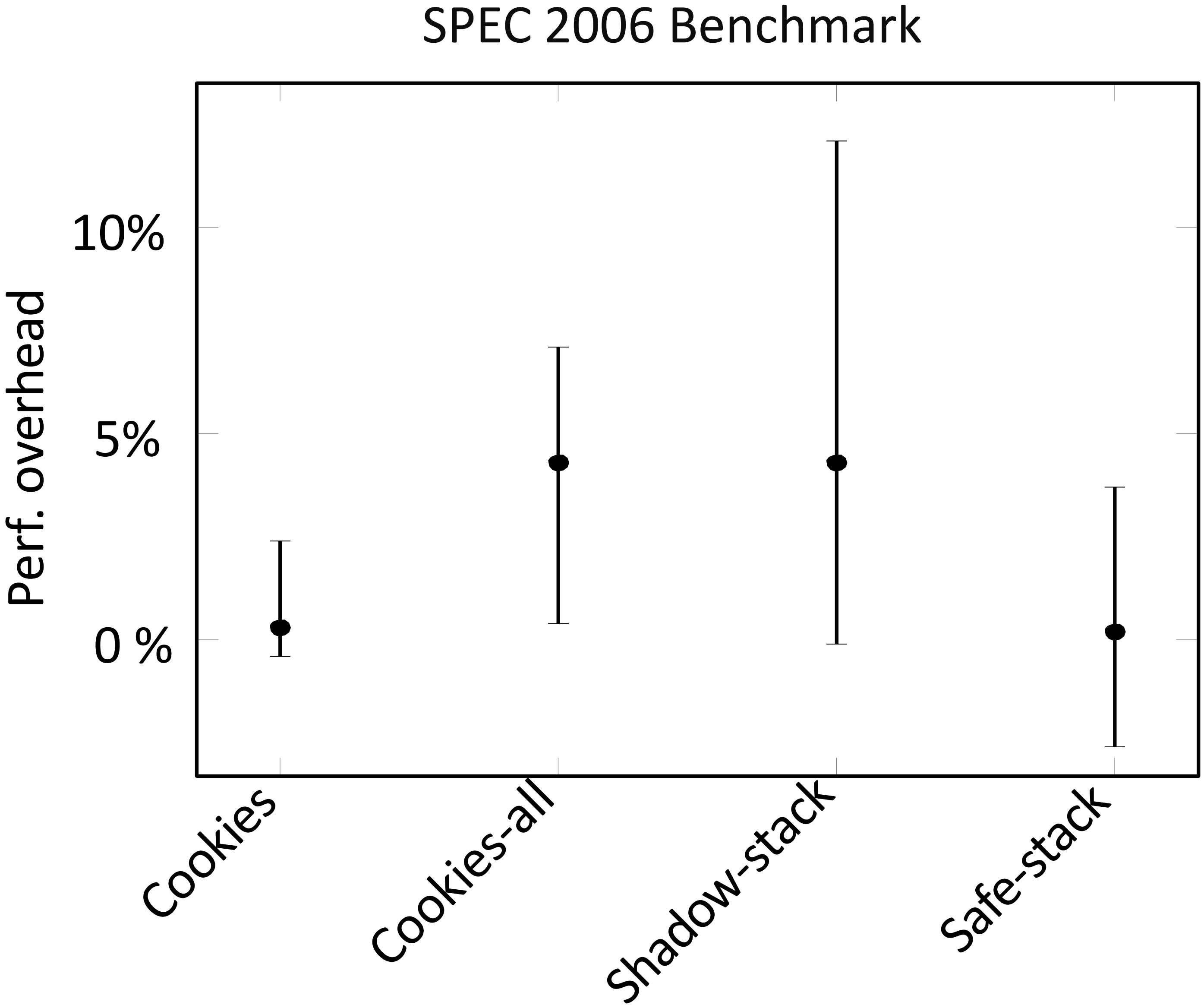
- **Strictly stronger** protection than stack cookies or shadow stack
- Only the Safe Stack provides **guaranteed** protection against return address corruption
- Stops **all ROP attacks** alone!

Safe Stack overhead

0% avg.



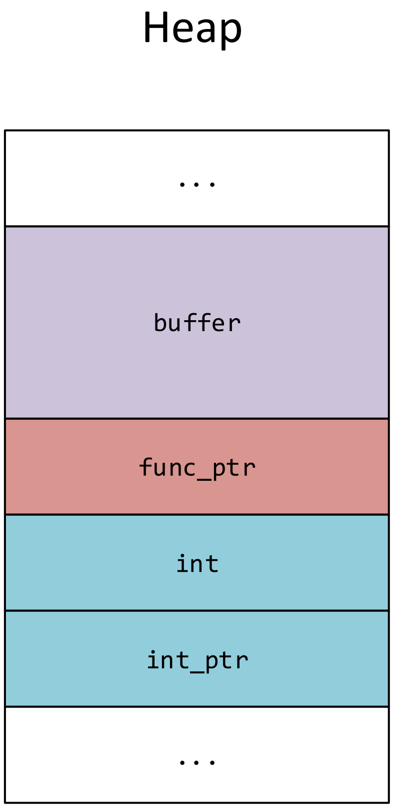
Safe Stack overhead



Code Pointer Separation

Protecting function pointers

Integrity of function pointers



Heap

...

buffer

func_ptr

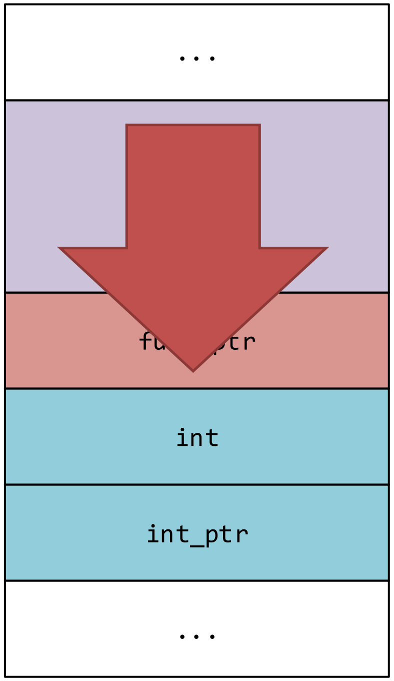
int

int_ptr

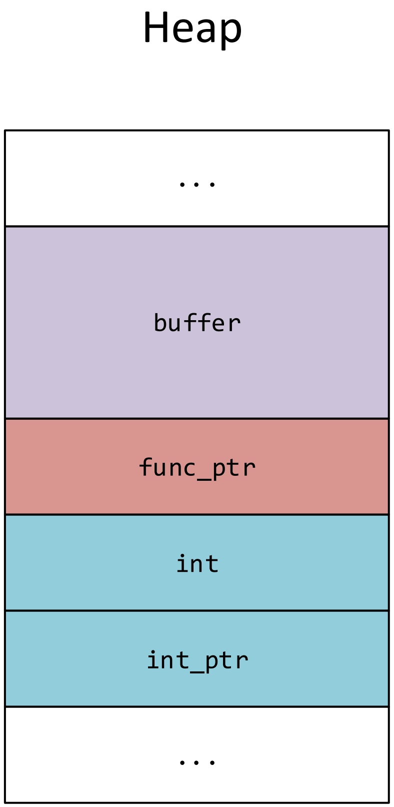
...

Integrity of function pointers

Heap

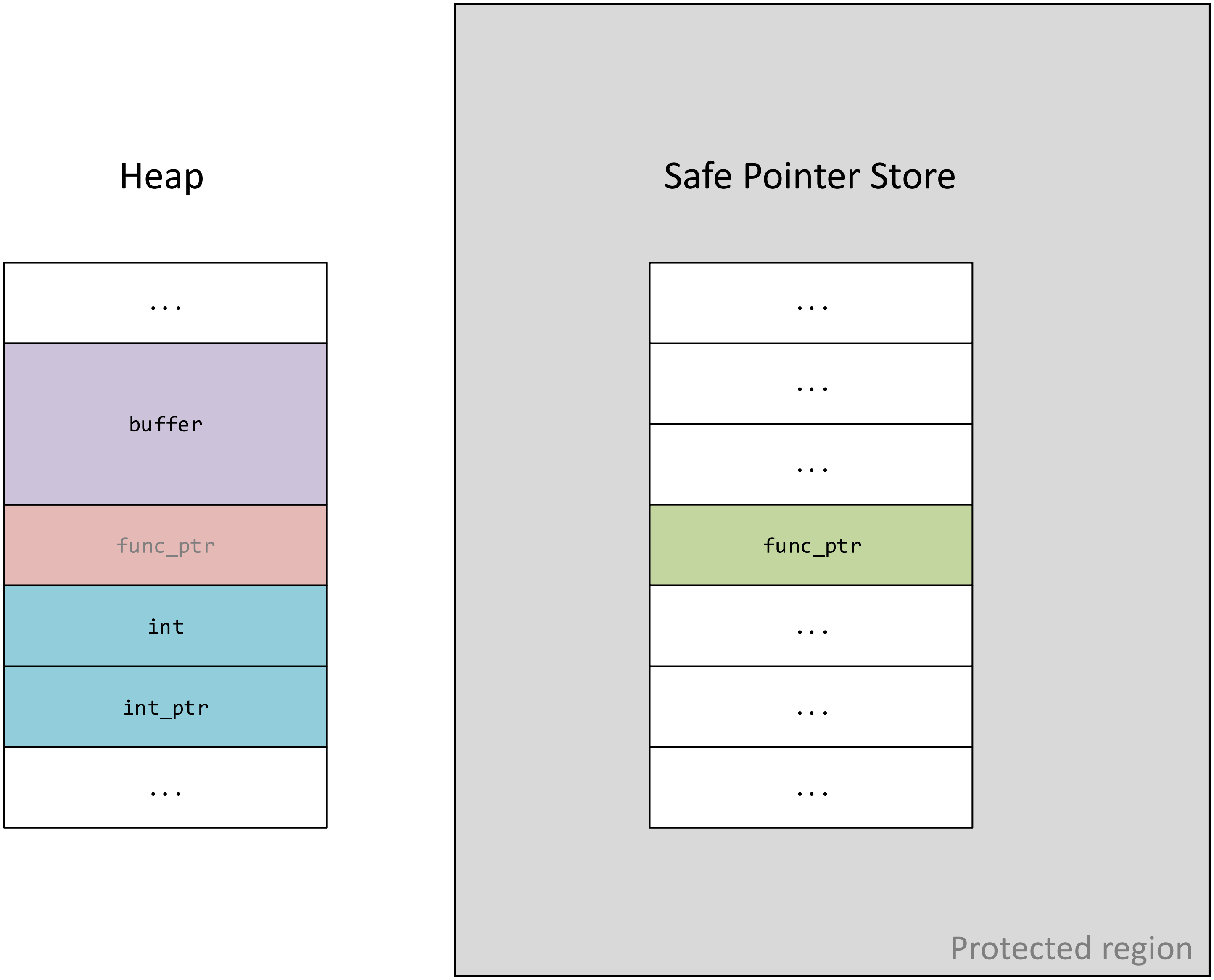


Integrity of function pointers



```
p[idx]=val;
```

Code Pointer Separation (CPS)



Heap

...

buffer

func_ptr

int

int_ptr

...

Safe Pointer Store

...

...

...

func_ptr

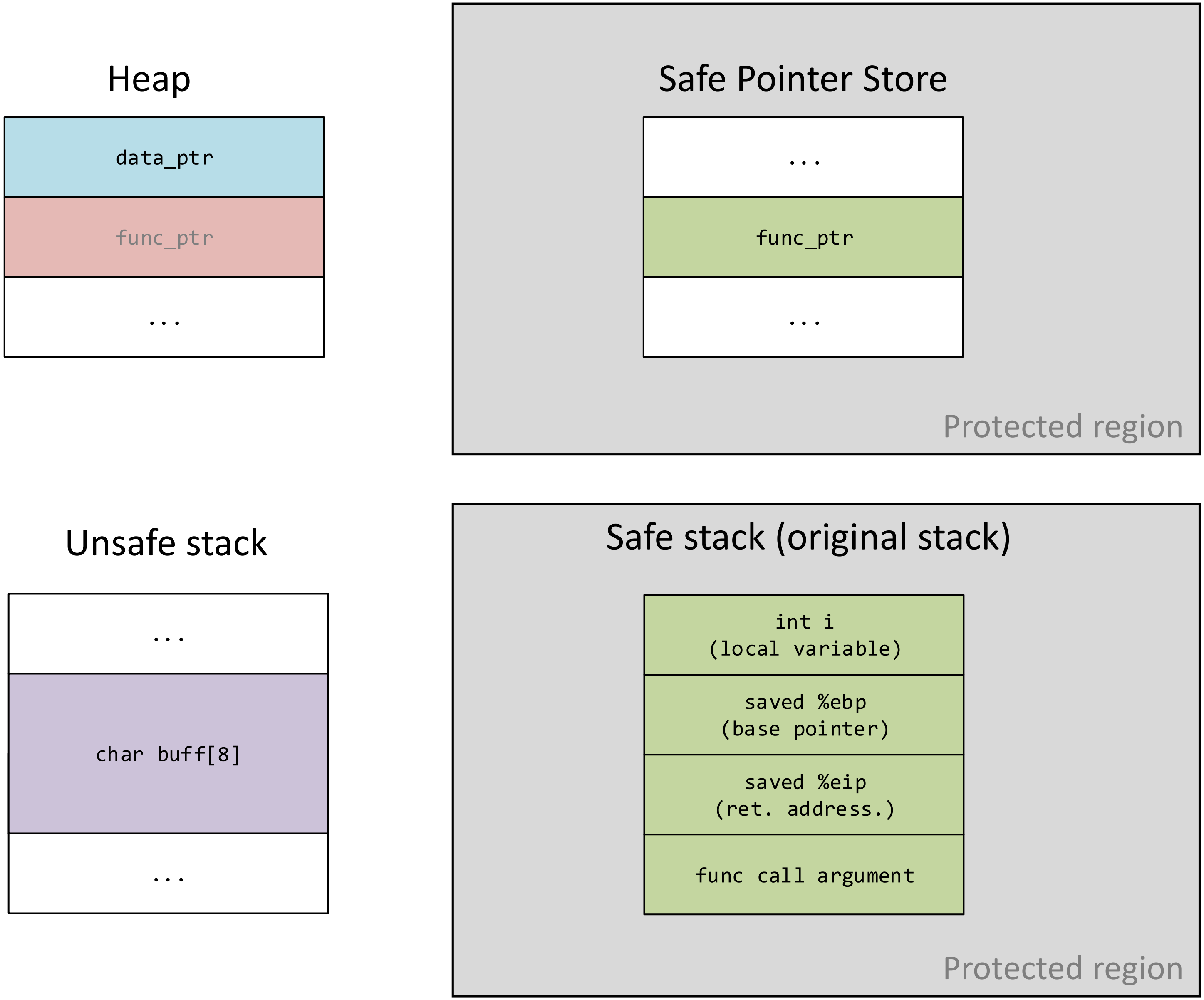
...

...

...

Protected region

Code Pointer Separation (CPS)



Heap

data_ptr

func_ptr

...

Safe Pointer Store

...

func_ptr

...

Protected region

Unsafe stack

...

char buff[8]

...

Safe stack (original stack)

int i
(local variable)

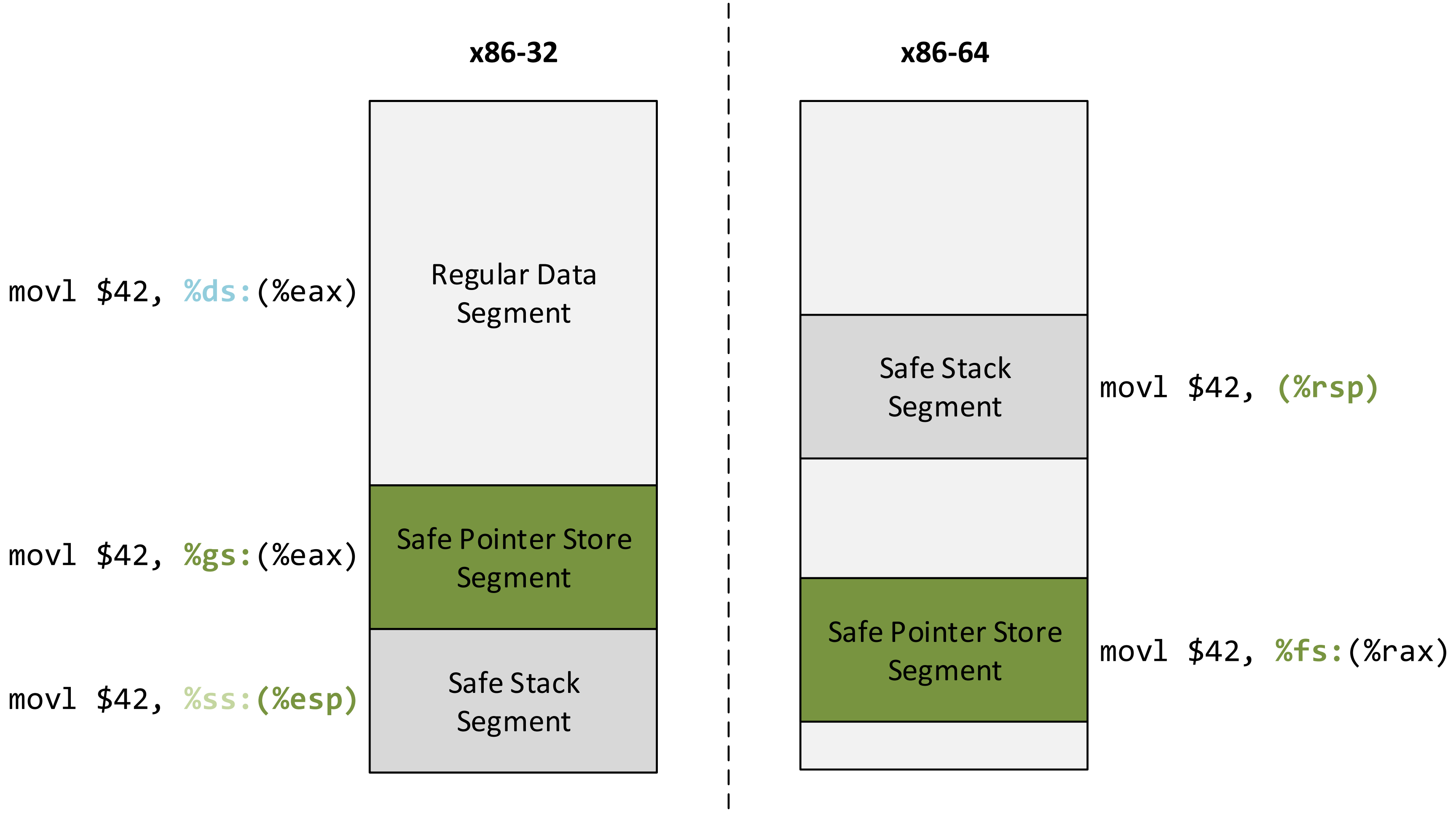
saved %ebp
(base pointer)

saved %eip
(ret. address.)

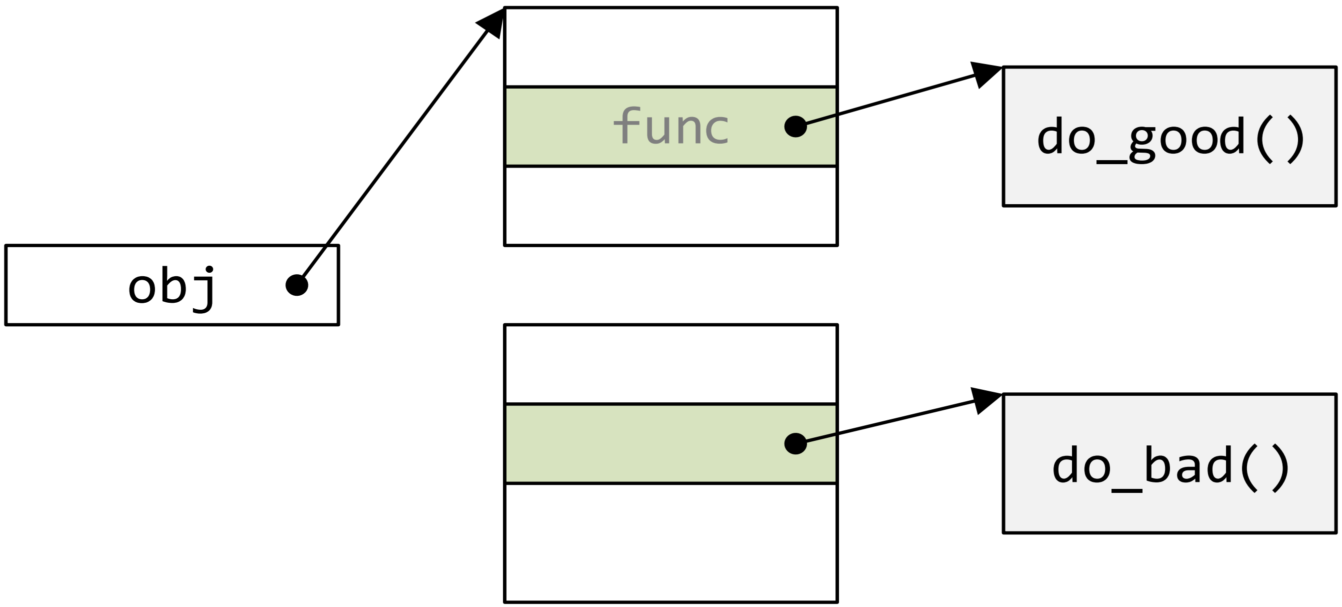
func call argument

Protected region

Protecting the Safe Pointer Store

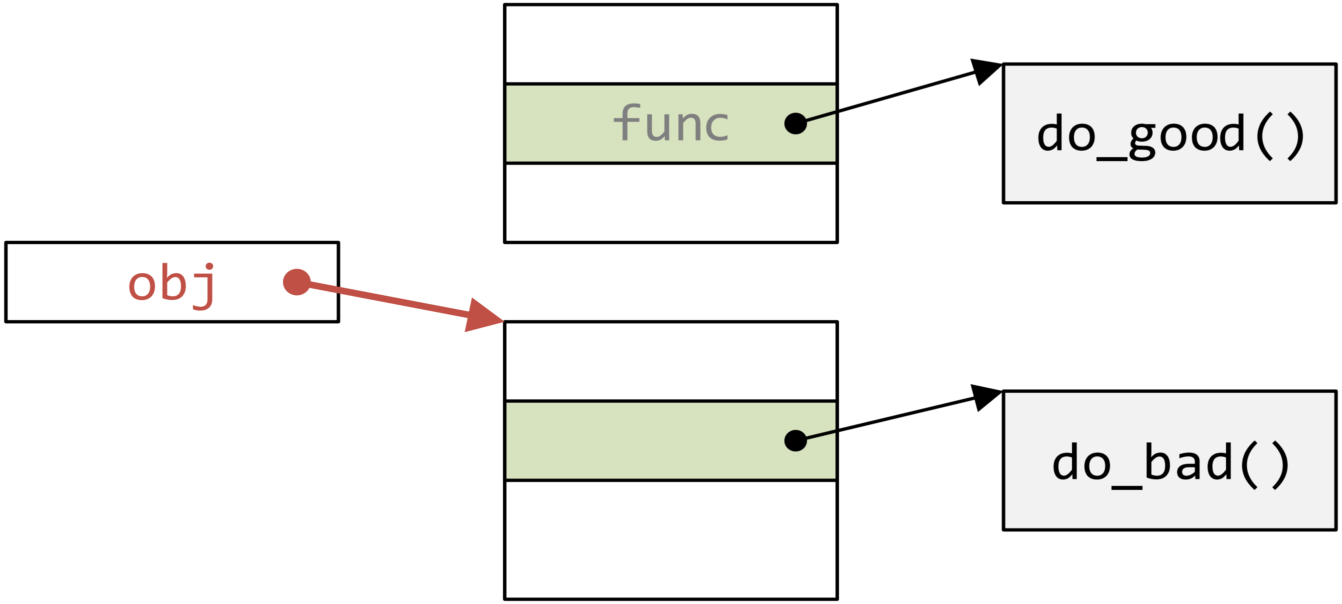


How effective is CPS?



```
obj->func();
```

How effective is CPS?



```
obj->func();
```

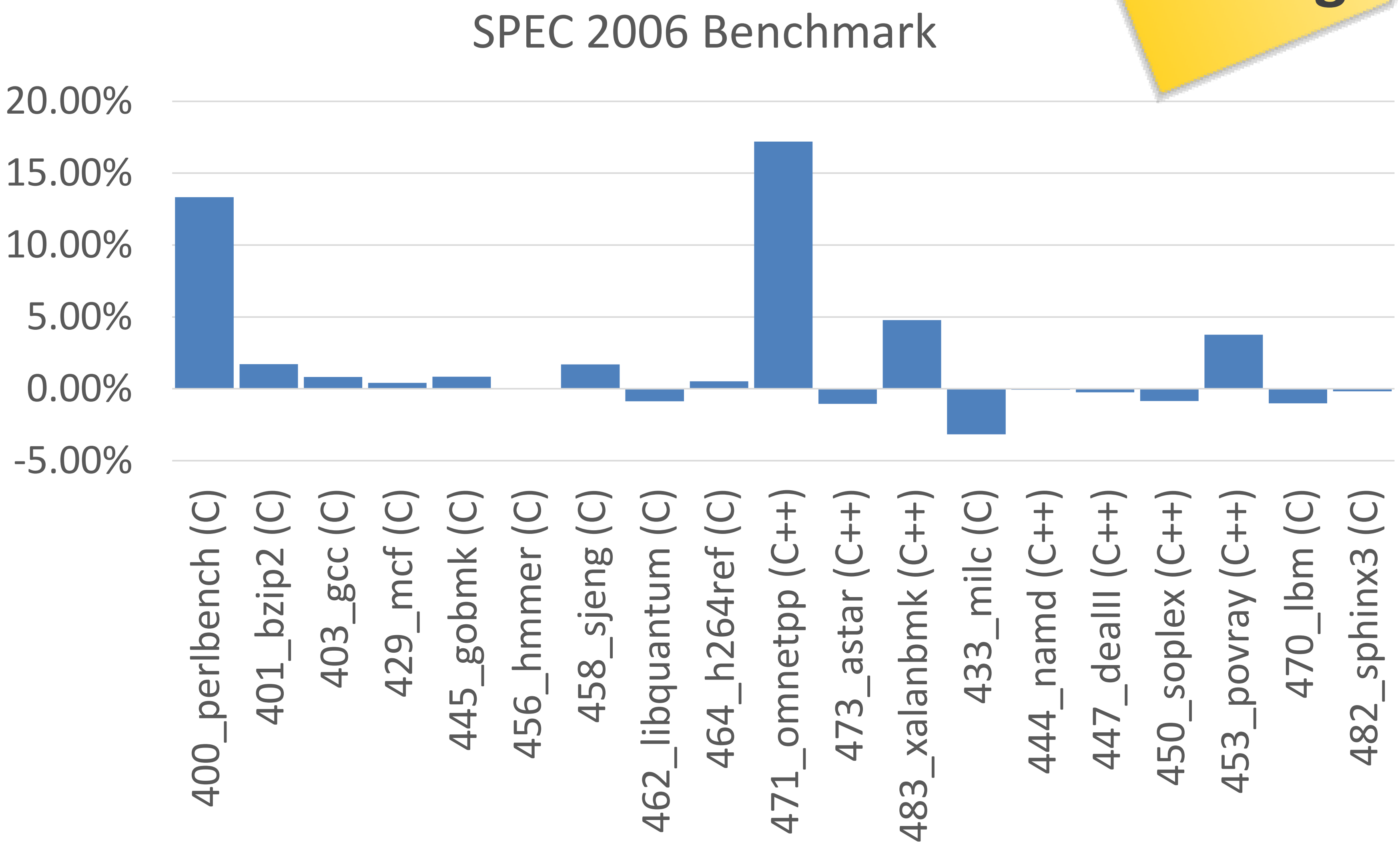
CPS vs. CFI

Practical CFI solutions
Classic CFI, CCS ‘05
CCFIR, IEEE S&P ‘13
binCFI, Usenix Sec ‘13
kBouncer, Usenix Sec ‘13

CFI attacks
Göktaş et al., IEEE S&P ‘14
Göktaş et al., Usenix Sec ‘14
Davi et al., Usenix Sec ‘14
Carlini et al., Usenix Sec ‘14

	CFI	**CPS**
Calls can go to	**any** function whose address is taken	any function whose address is taken **and** *stored in memory at the current point of execution*
Return can go to	**any** call site	**only** their actual caller

CPS overhead

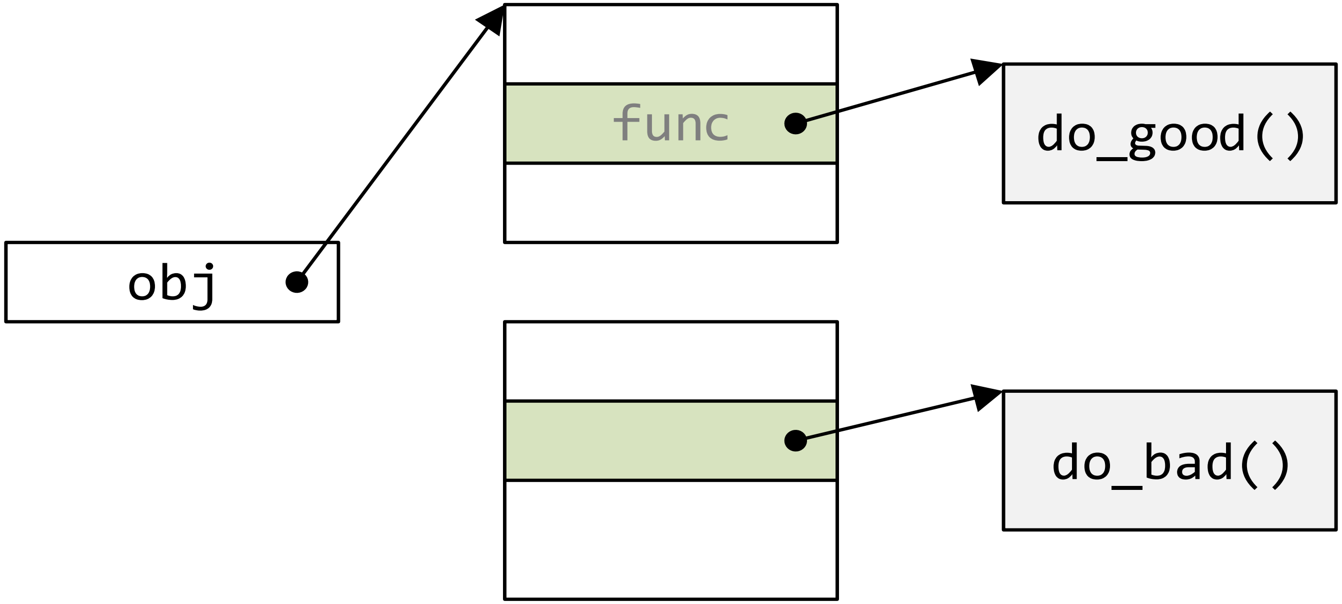


2% avg.

Code Pointer Integrity

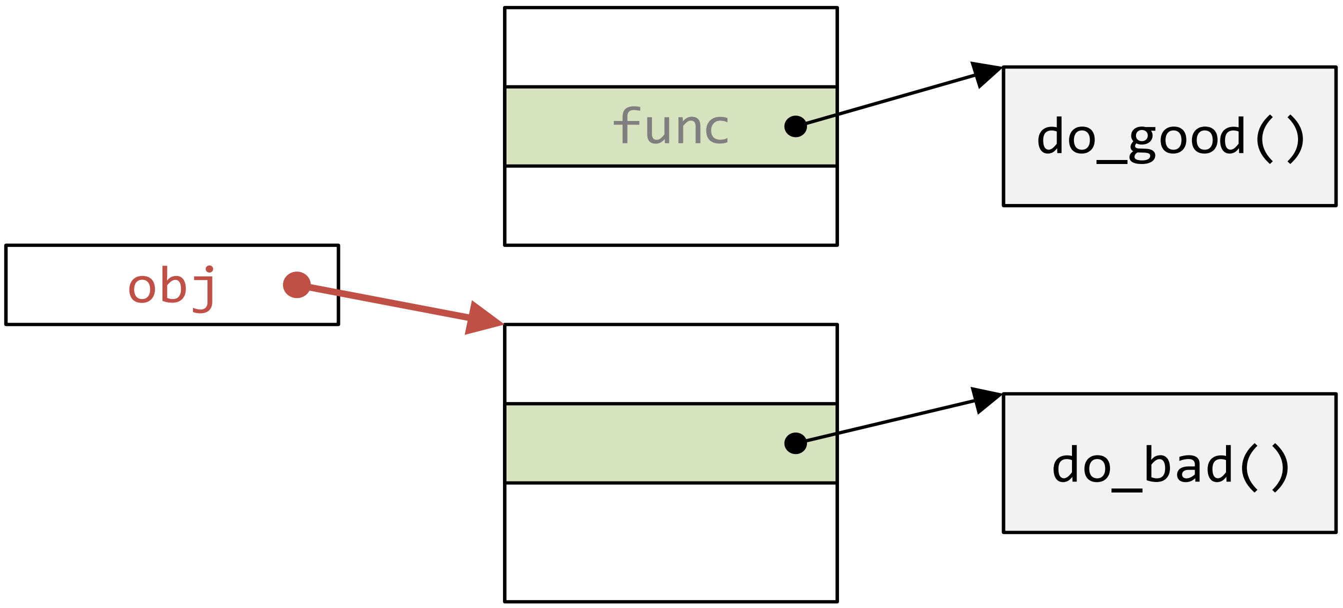
Guaranteed protection of *all* code pointers

Issue #1



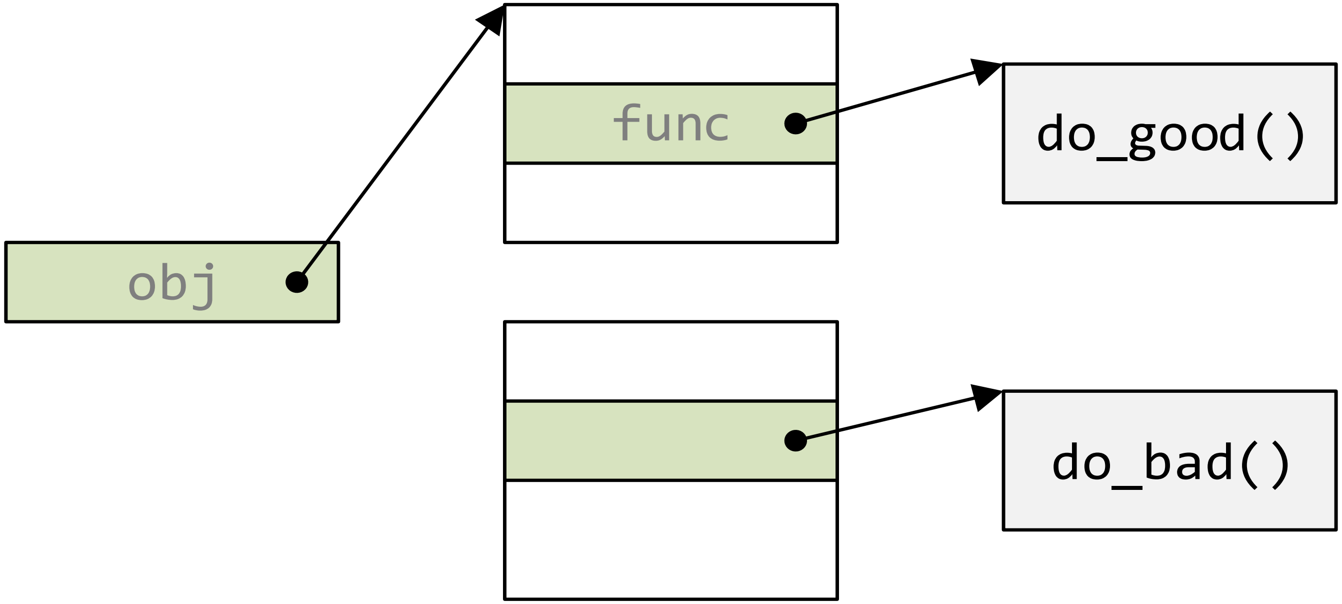
```
obj->func();
```

Issue #1: pointer coverage



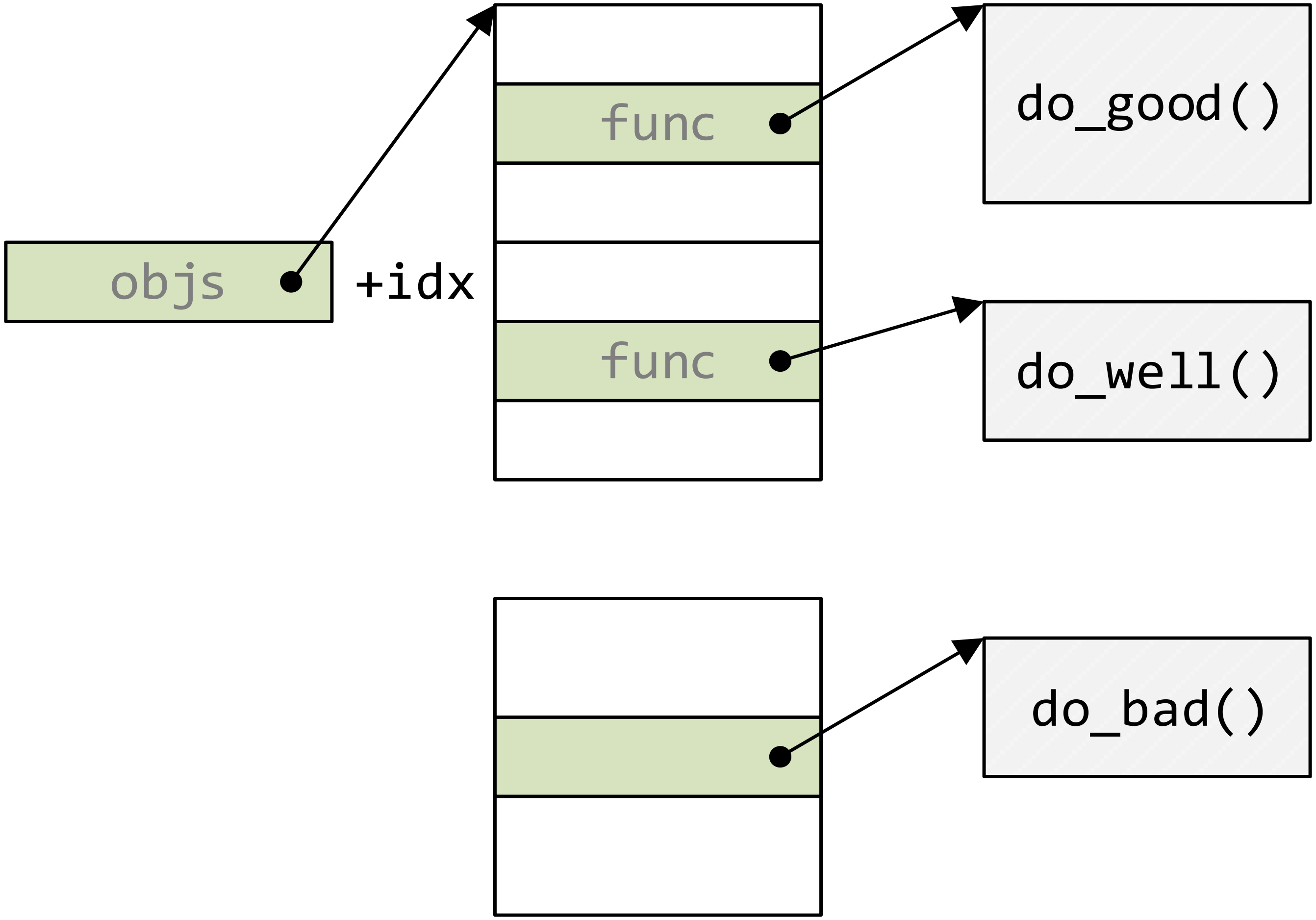
```
obj->func();
```

Issue #1: pointer coverage



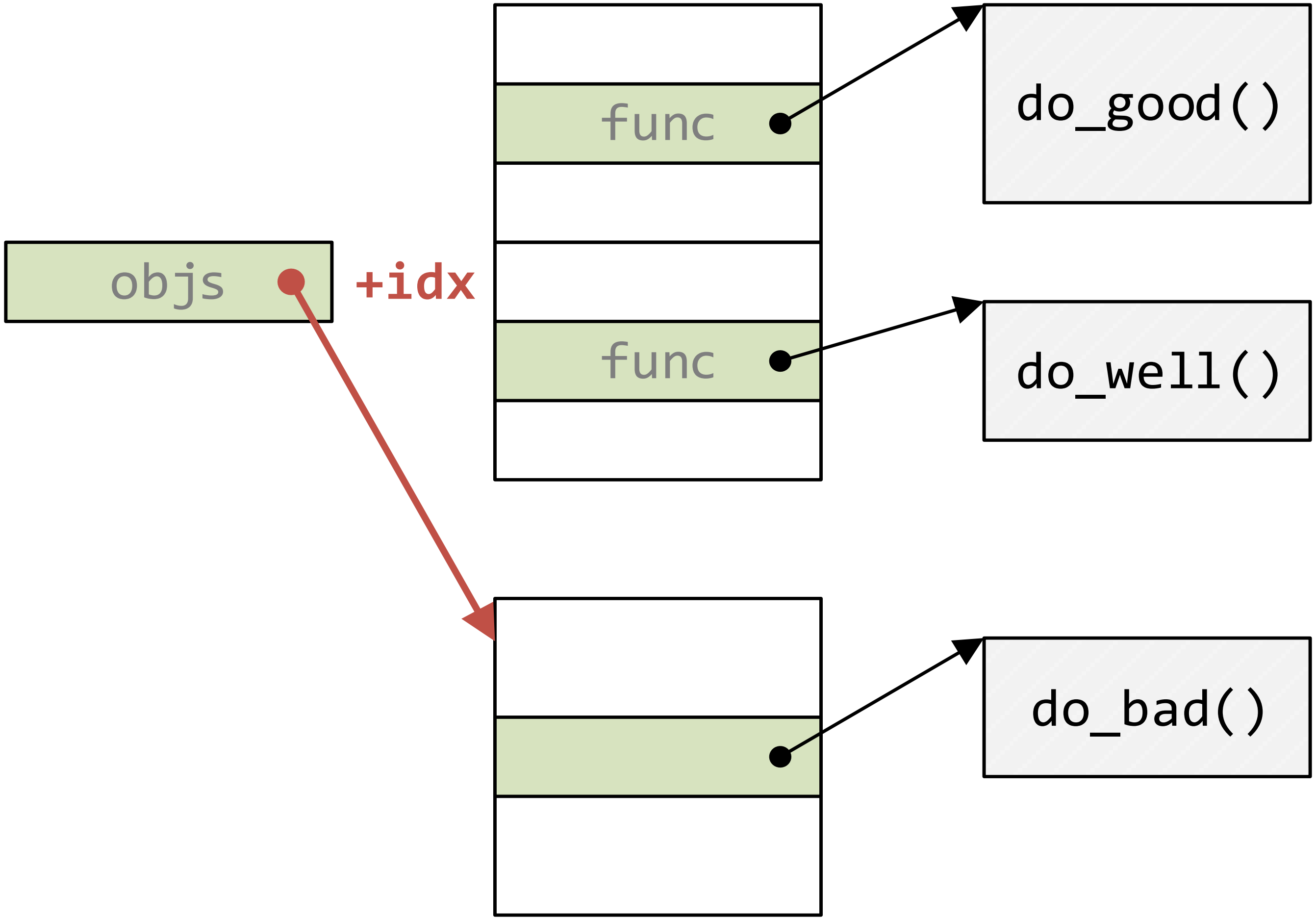
```
obj->func();
```

Issue #2



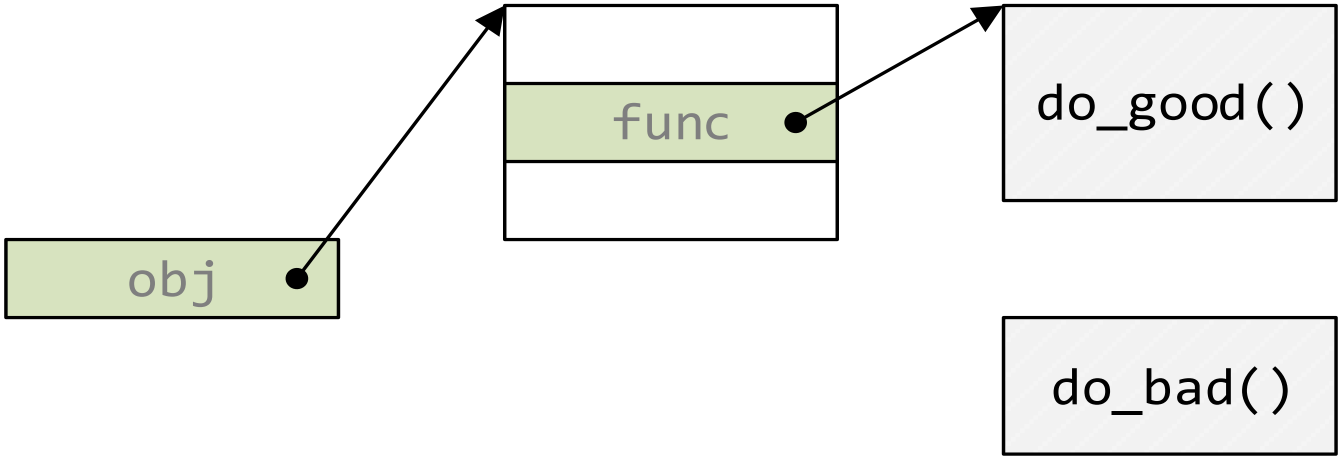
```
obj=&objs[idx]
obj->func();
```

Issue #2: spatial safety



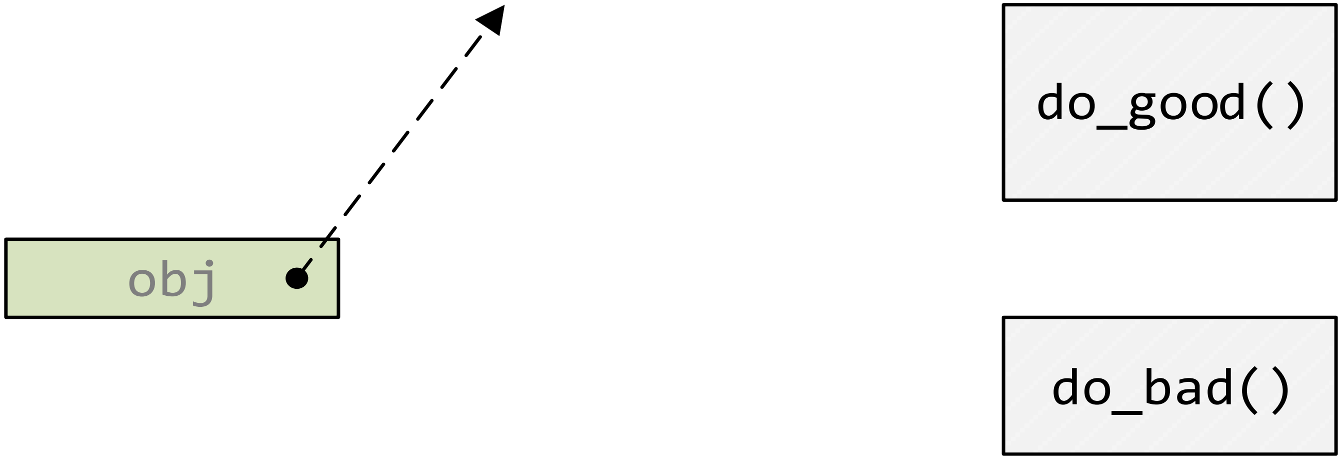
```
obj=&objs[idx]
obj->func();
```

Issue #3



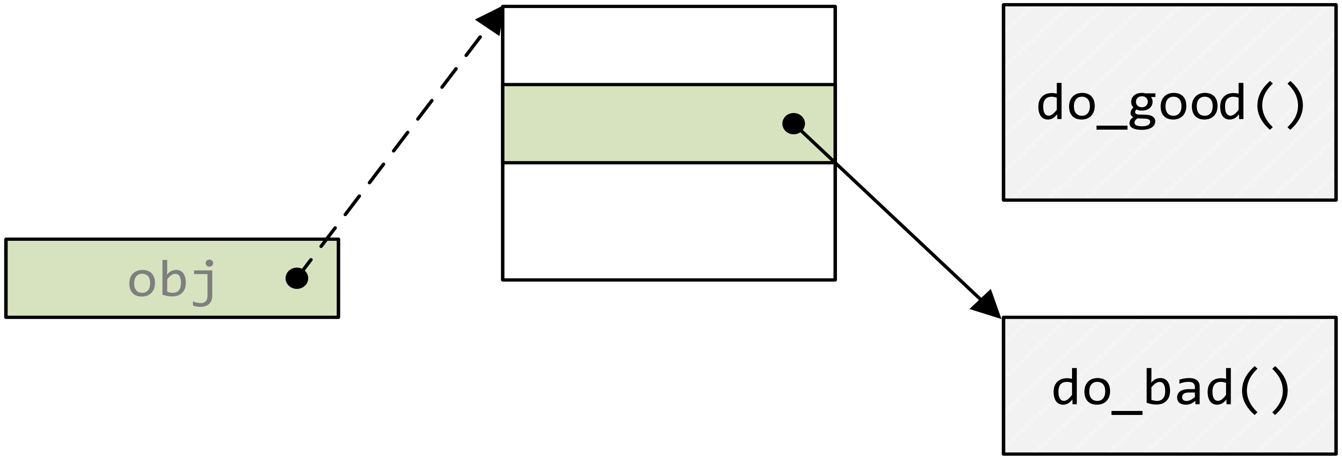
```
⇨ delete obj;
  …
  obj->func();
```

Issue #3



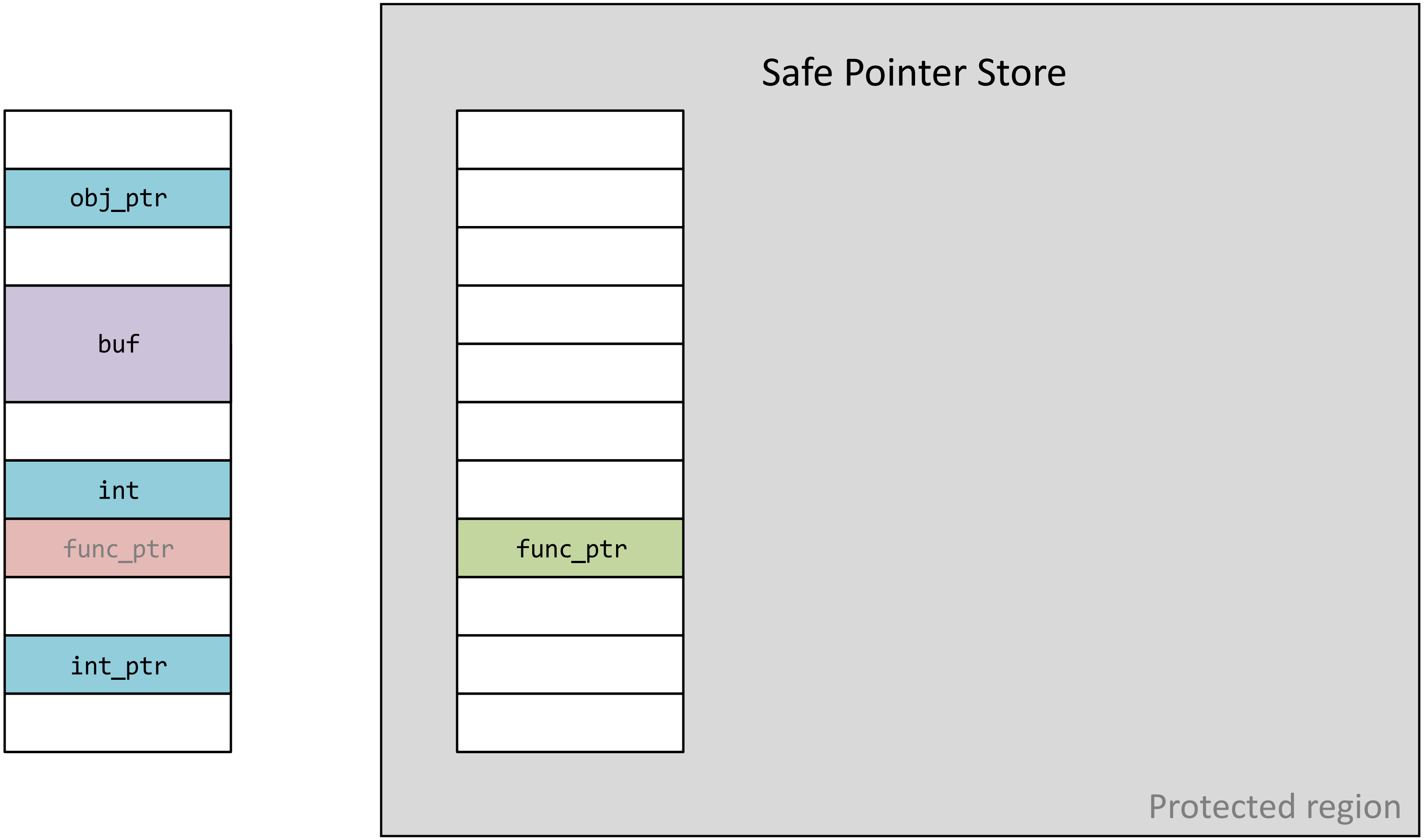
```
  delete obj;
⇨ …
  obj->func();
```

Issue #3: temporal safety



```
  delete obj;
  …
⇨ obj->func();
```

CPS → Code Pointer Integrity



obj_ptr

buf

int

func_ptr

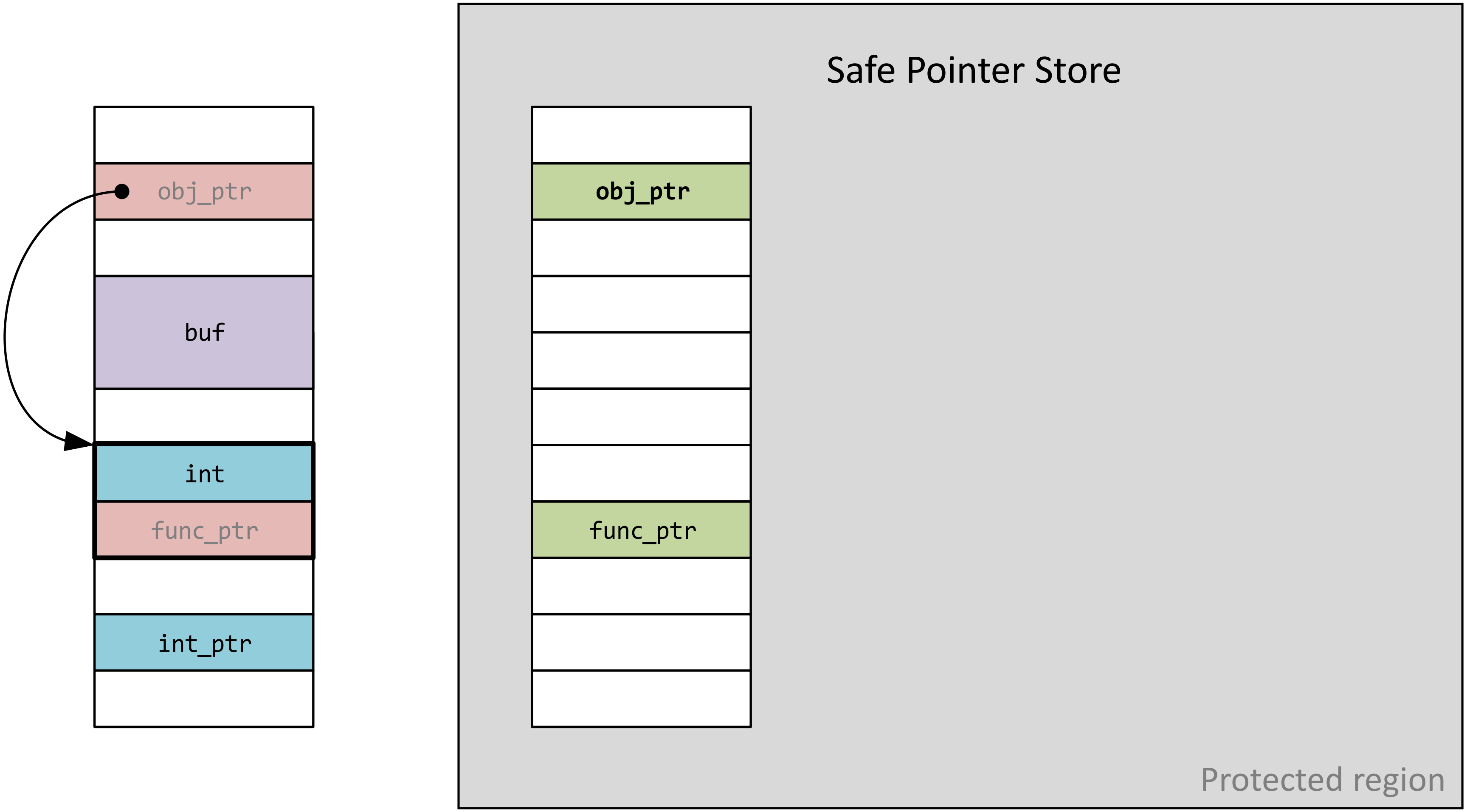
int_ptr

Safe Pointer Store

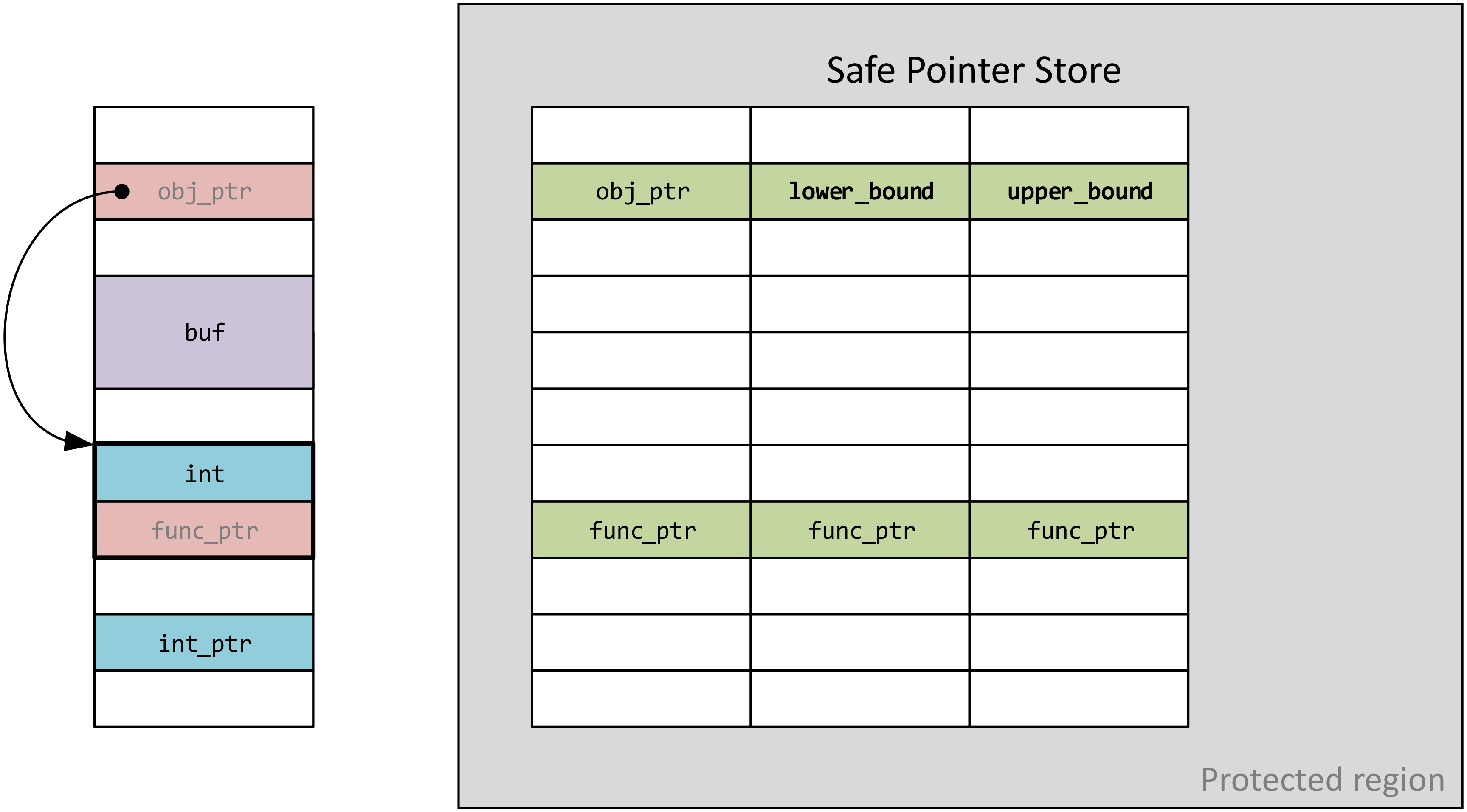
func_ptr

Protected region

Issue #1: pointer coverage



Issue #2: spatial safety



obj_ptr

buf

int

func_ptr

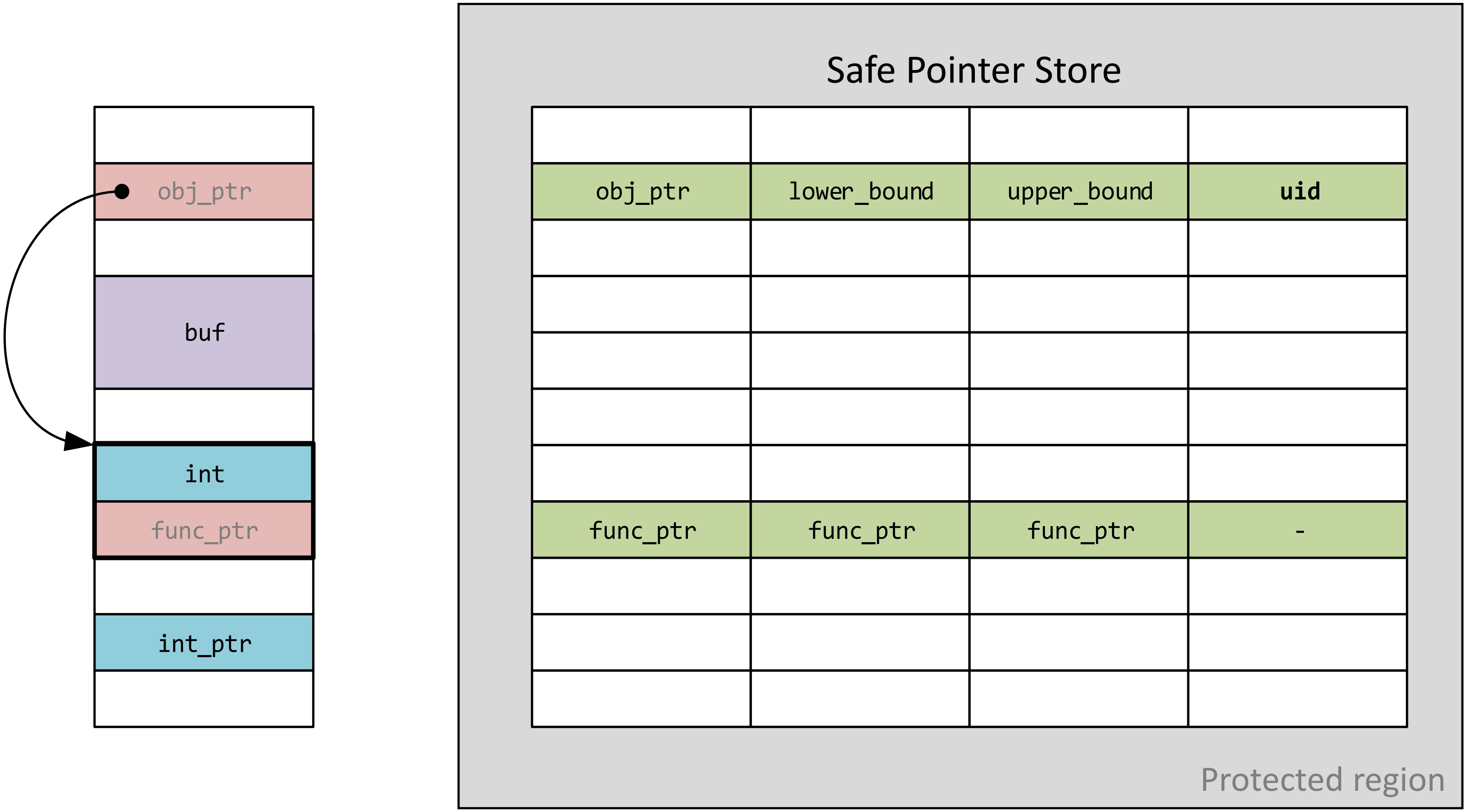
int_ptr

Safe Pointer Store

obj_ptr	**lower_bound**	**upper_bound**
func_ptr	func_ptr	func_ptr

Protected region

Issue #3: temporal safety



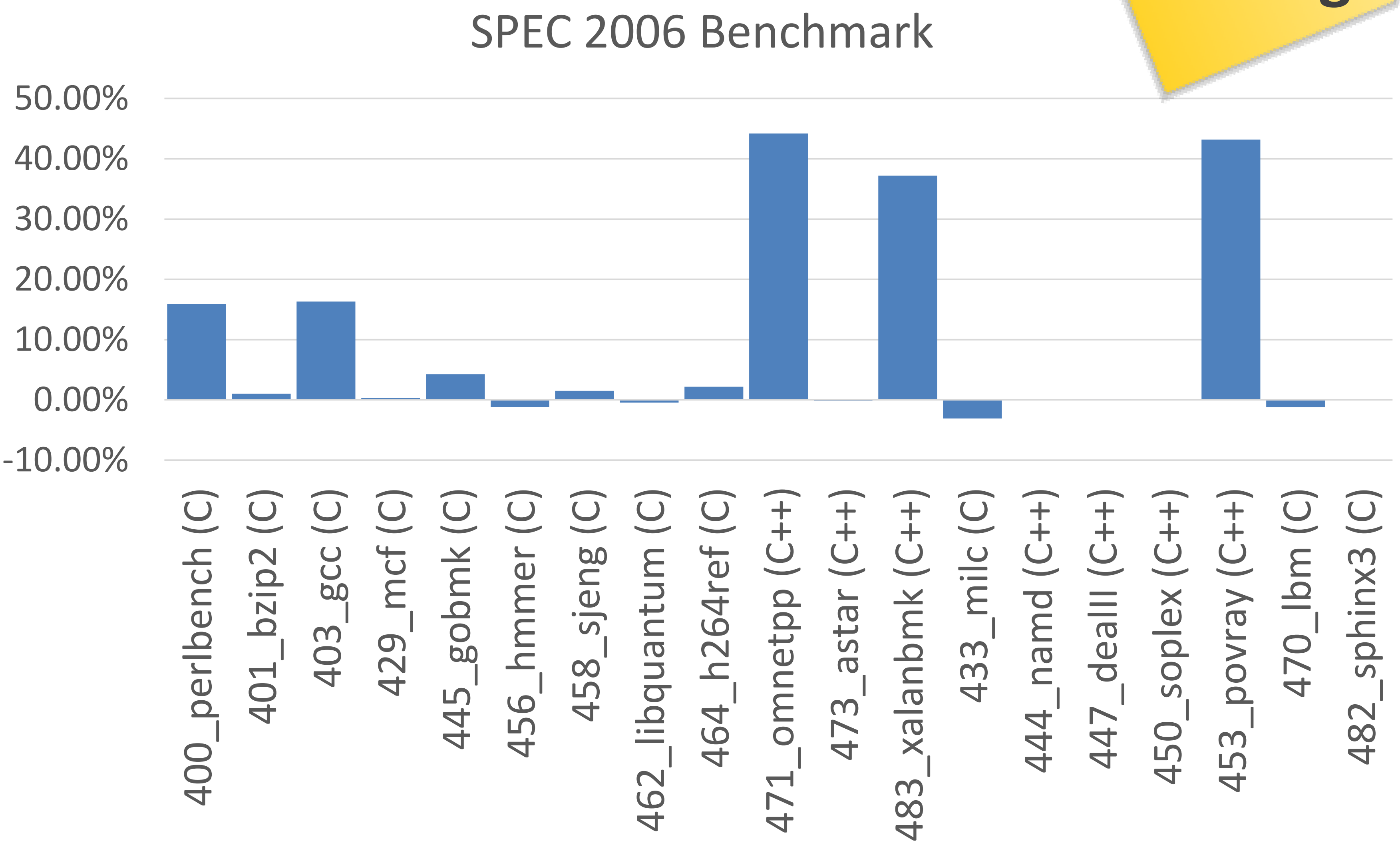
Safe Pointer Store

obj_ptr	lower_bound	upper_bound	**uid**
func_ptr	func_ptr	func_ptr	-

Protected region

CPI overhead

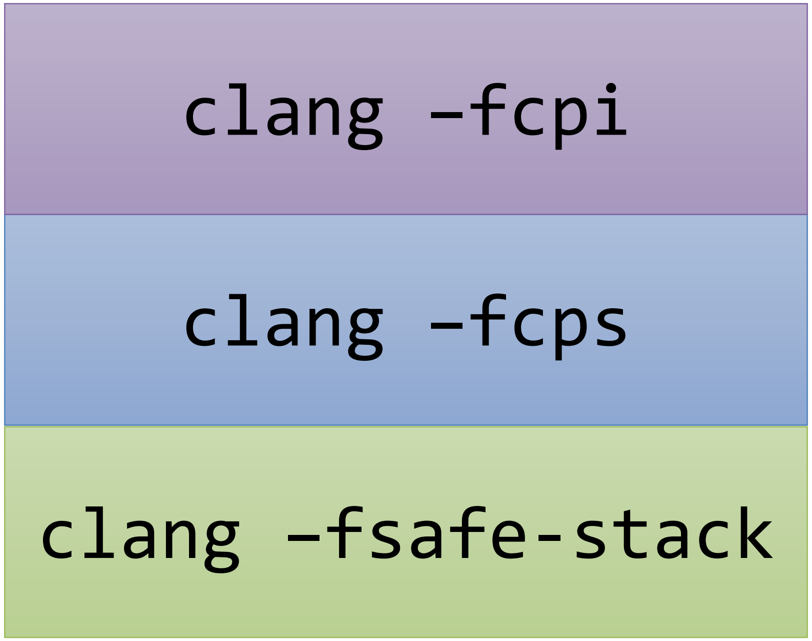
8% avg.



Implementation

and case studies

Levee in LLVM/Clang



```
clang –fcpi
clang –fcps
clang –fsafe-stack
```

Get the prototype from: http://levee.epfl.ch

Control-flow hijack protected FeeBSD

- Complete FreeBSD distribution (modulo kernel)
- >100 extra packages

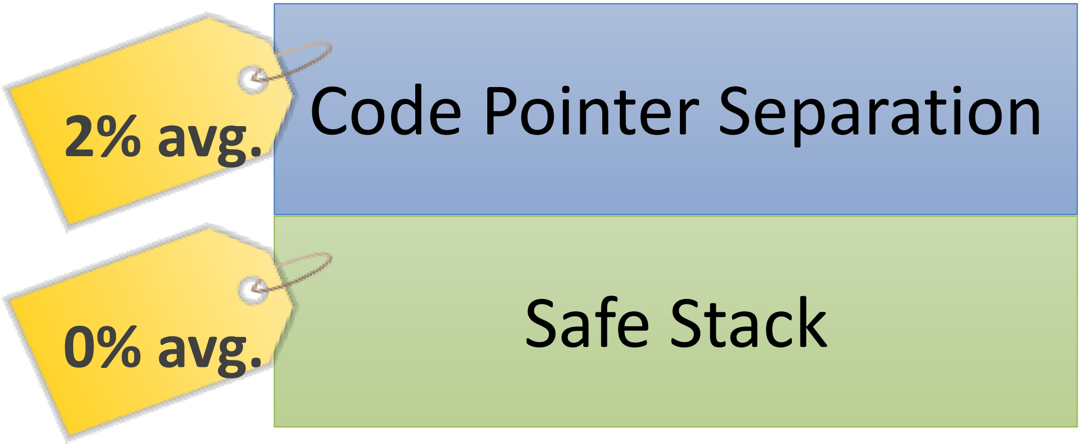
Summary

Summary

0% avg.

Safe Stack

Summary



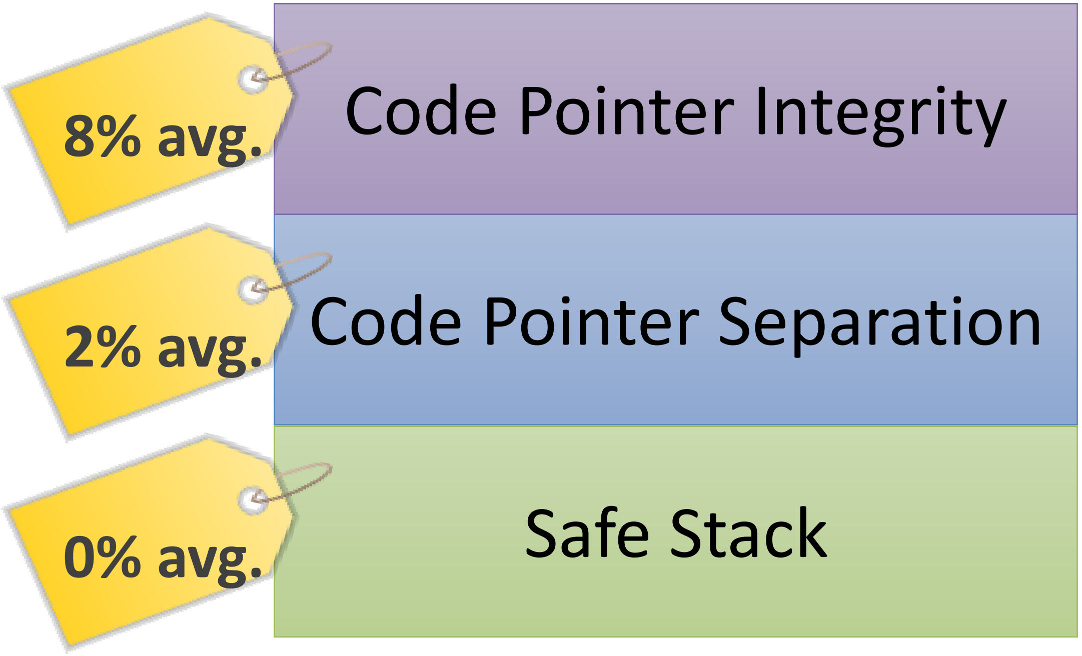
2% avg.

Code Pointer Separation

0% avg.

Safe Stack

Summary



8% avg. — Code Pointer Integrity

2% avg. — Code Pointer Separation

0% avg. — Safe Stack

Thank you!

Questions?