Stack Bounds Protection with Low Fat Pointers

Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro

NDSS 2017

NUS National University of Singapore
School of Computing

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Overview

Heap Bounds Protection with Low Fat Pointers, CC 2016

New method for detecting bounds overflow errors *without explicit metadata*

Pros: Fast (~13% w.o.), near zero memory overheads, highly compatible

Cons: Only protects heap allocation (**`malloc`**) only!

Stack

~~*Heap*~~ *Bounds Protection with Low Fat Pointers*, NDSS 2017

Extend bounds overflow protection to **stack objects**

Requires a whole new bag of tricks

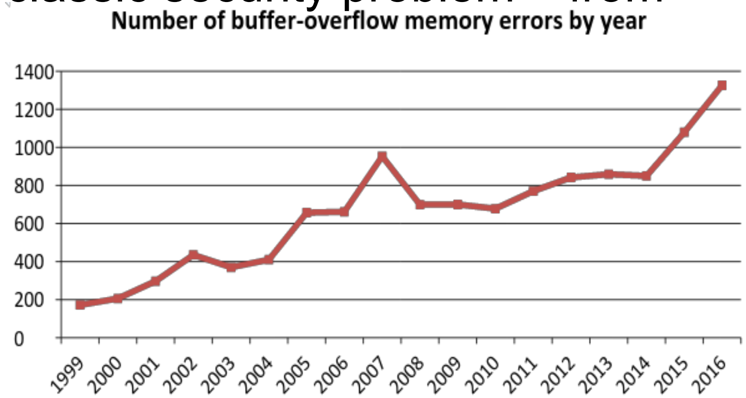
Pros: Fast (~17% w.o.), near zero memory overheads, highly compatible

Motivation

Buffer overflows (spatial memory errors) are classic security problem – from 1970s to present

Continue to be an active threat:

- Heartbleed, Ghost,**Cloudbleed** (Feb 2017), etc.



Common defenses have weaknesses:

- *ASLR^Cache: Practical Cache Attacks on the MMU* (NDSS'17)

Stronger defenses are rarely used

- Overheads
- Compatibility

Countermeasures

Perennial problem, many countermeasures have been proposed.

Indirect methods:

- ASLR and DEP
- Control Flow Integrity (CFI), Code Pointer Integrity, etc.
- Data Flow Integrity (DFI)
- Shadow Stacks, etc.

Direct methods:

- **Many** existing systems: AddressSanitizer, SoftBound, SafeC, CCured, BaggyBounds, PAriCheck, **low-fat-pointers**, etc. etc.
- Most systems track *bounds metadata*

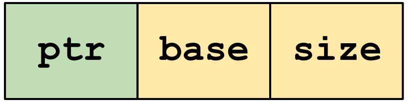
```
if (p < base(O) || p >= base(O)+size(O))
    error();
*p = v;
```

Bounds Checking Approaches

"**Fat pointers**" combine pointers and meta data

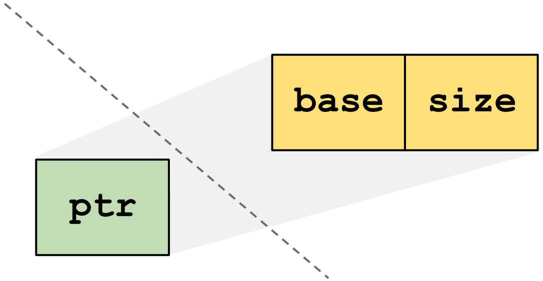
```
struct fat_ptr {
  void *ptr;
  void *base;
  size_t size; };
```

```
size(p) = p.size
base(p) = p.base
```



Shadow space stores metadata in separate memory

```
size(p) = SHADOW[p].size
base(p) = SHADOW[p].base
```

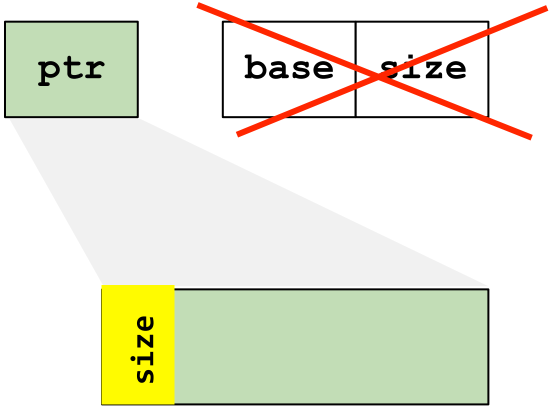


Low Fat Pointers

Low fat pointers are like fat pointers *without the fat*:

```
union low_fat_ptr {
  void *ptr;
  uintptr_t size:10;
};
```

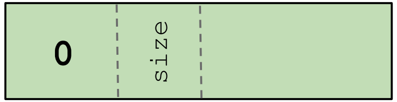
```
size(p) = p.un.size
base(p) = (p / size(p)) * size(p)
```



Compact encoding with no space overheads.

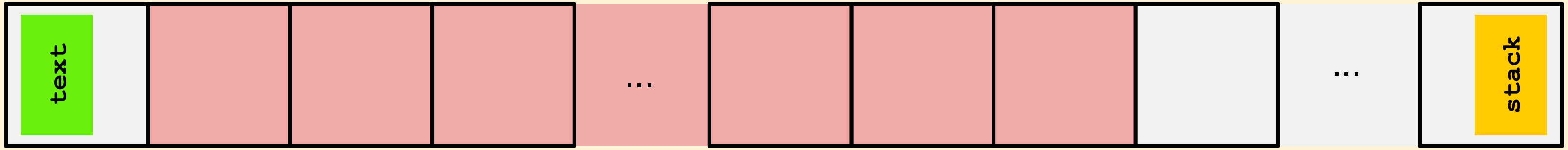
Flexible Low Fat Pointers

A simple encoding does not work well in practice



- Only 48bits are used → high bits must be zero!
- 10bits not big enough ~2^{10}=1024 max object size…

Better approach: *Heap Bounds Protection with Low Fat Pointers* (CC'16)



- Virtual address space subdivided into several large regions (eg. 32GB each)
- Each region is used to allocate objects of a specific size (16B, 32B, 48B, etc.).

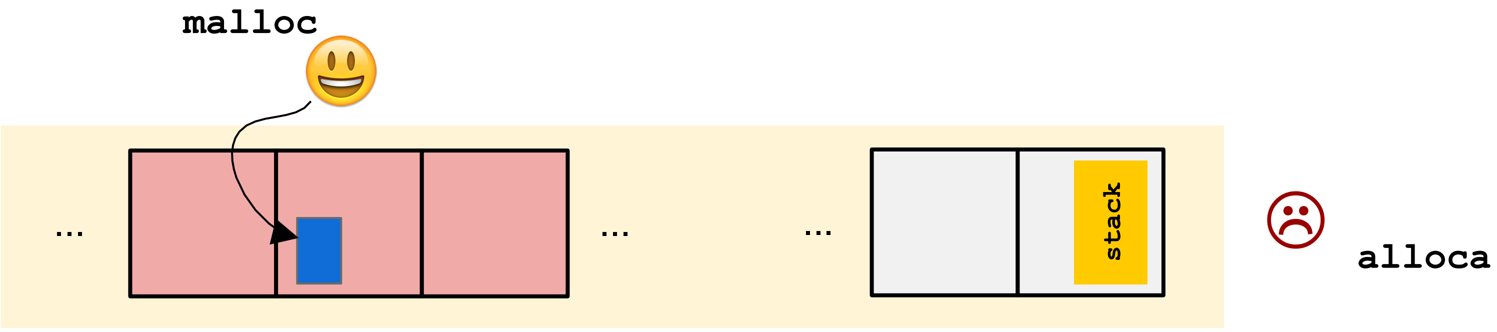
Bounds Checking with Low Fat Pointers

Object size is linked to regions, and used for bounds checking:

```
size(p) = SIZES[p / 32GB]
base(p) = (p / size(p)) * size(p)
```

```
if (p < base(p) || p >= base(p)+size(p))
     error();
*p = v;
```

This works fine for heap allocation, but ***not*** for stack allocation!



Stack Challenges

Problem #1: how to round up object size to allocation size ?

Problem #2: what should the alignment be?

Problem #3: where to place the object?

Problem #4: how to not waste memory?

Solutions:

Lookup tables

Virtual memory tricks

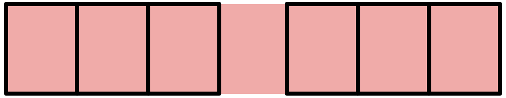
Allocation Size Over Approximation

Given:

```
char object[N];   /* Stack allocation */
```

Problem: which region does **`object`** belong to???

Must decide in a few instructions.



Solution:

Use a *lookup table* (**`SIZES`**) indexed by lzcnt(N)

```
char object[50];
```

```
lzcnt(50) = 58
```

SIZES

...	55	56	57	58	59	60	61	62	63	64
...	512	256	128	64	32	16	16	16	16	16

```
lzcnt %rax, %rax
sub SIZES(,%rax,8), %rsp
```

Allocation Size Alignment

Problem: We have to align the object.

```
base(p) = (p / size(p)) * size(p)
```

Solution: just use the `attribute(aligned(N))`:

```
char object[64] attribute(aligned(64));
```

```
and $-64, %rsp
```

For ***variable length*** *objects* we also use lookup tables.

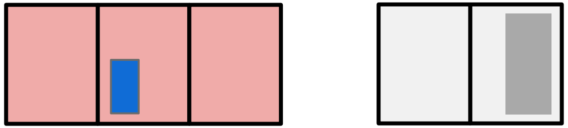
	55	56	57	58	59	60	61	62	63	64
...	-512	-256	-128	-64	-32	-16	-16	-16	-16	-16

MASKS

```
lzcnt %rax, %rax
and MASKS(,%rax,8), %rsp
```

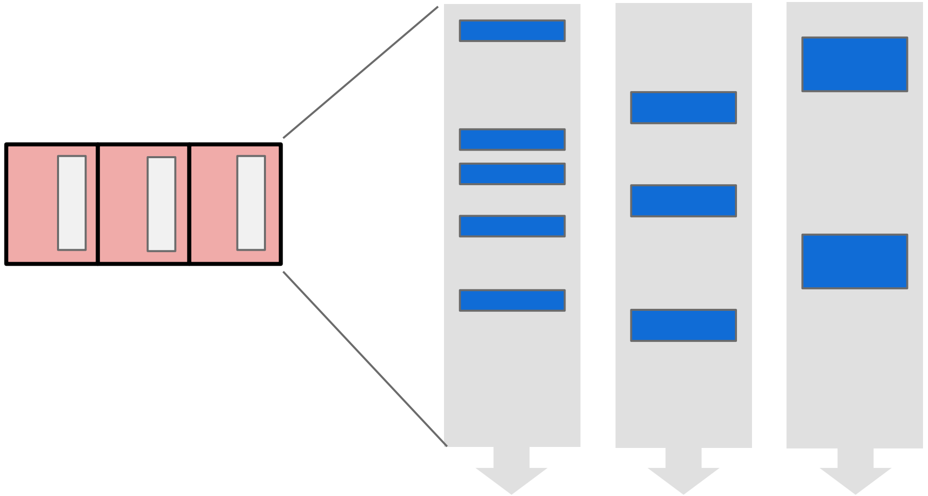
Stack Object Mirroring

Problem: stack objects are allocated from the stack!



☹ **`alloca`**

Solution: Split the stack into N stacks, one for each size region:



😃 **`alloca`**

Related work: *shadow stacks*

Stack Object Mirroring (cont.)

Stack Object Mirroring also implemented using tables:

	55	56	57	58	59	60	61	62	63	64
...	Δ_{55}	Δ_{56}	Δ_{57}	Δ_{58}	Δ_{59}	Δ_{60}	Δ_{61}	Δ_{62}	Δ_{63}	Δ_{64}

OFFSETS

Δ_{58}= **&region #4 - &stack**

```
lzcnt %rax, %rax
add OFFSETS(,%rax,8), %rsp
```

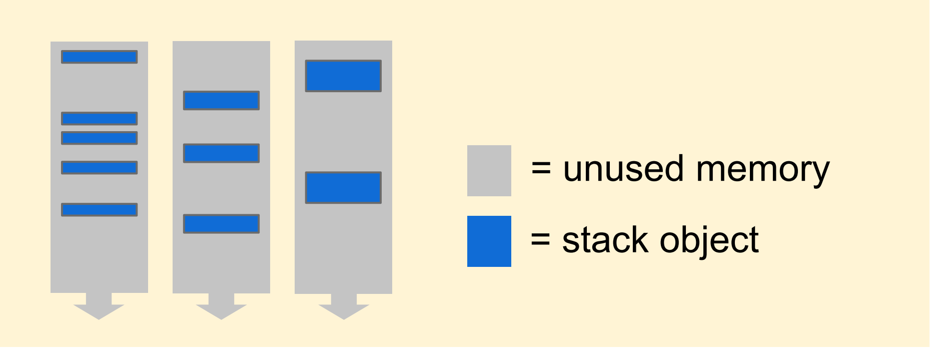
Each object allocated in correction region.

Backwards compatible with deallocation, **`longjmp`**, C++ exceptions, asm code, etc.

CON: Uses more memory

1 stack replaced with N stacks.

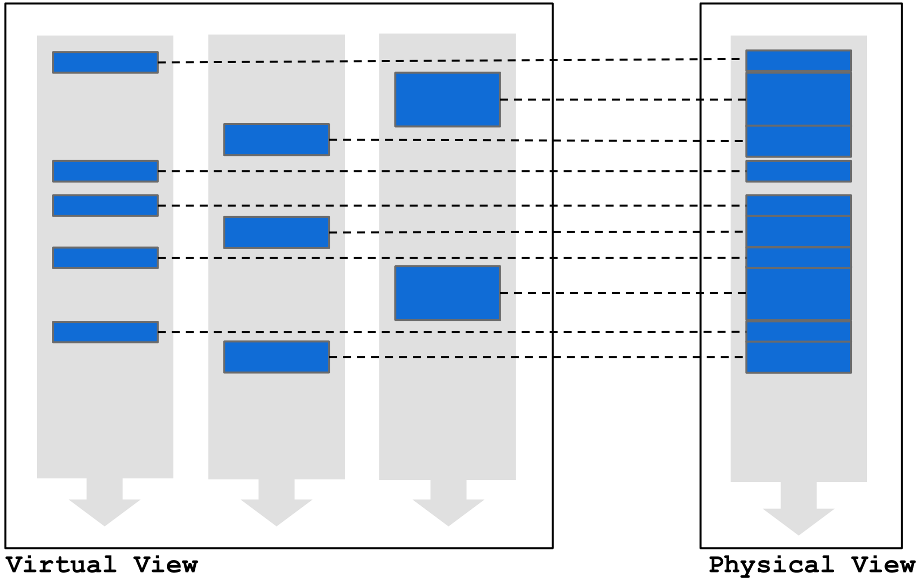
Fragmentation.



Memory Aliasing

Problem: Increasing stack memory is unsatisfactory.

Solution: make all stacks *share the same physical memory*:



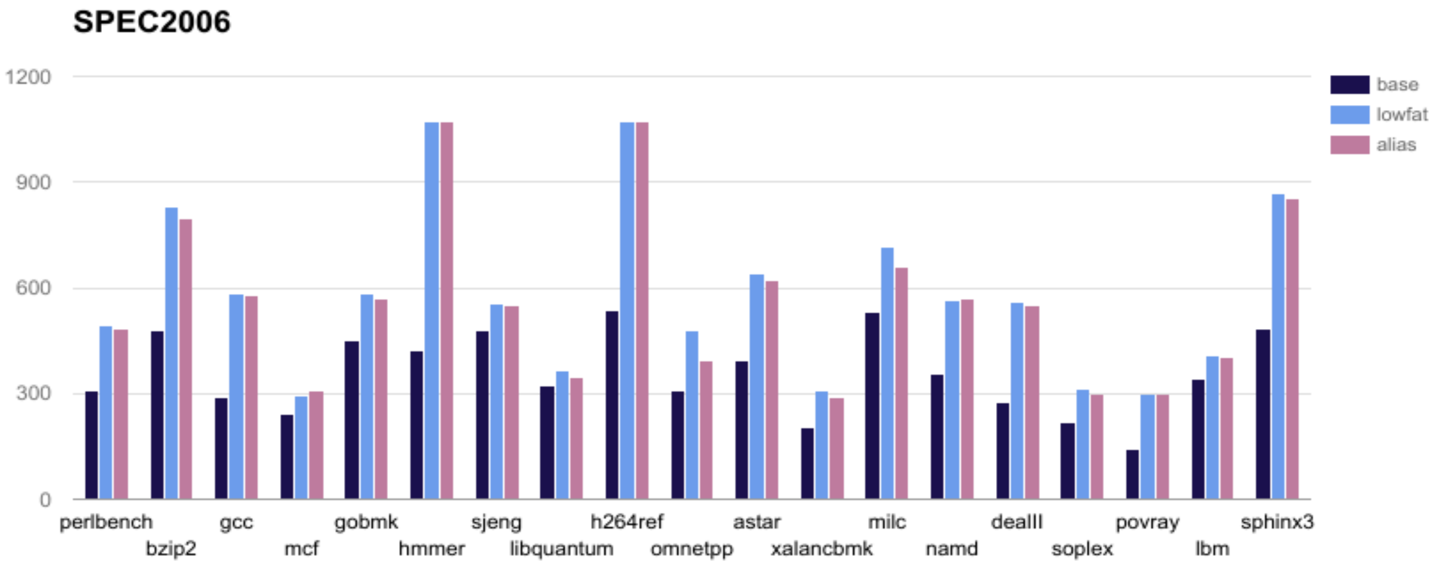
Program uses a single stack

(same as before)

Uses *shared memory objects*

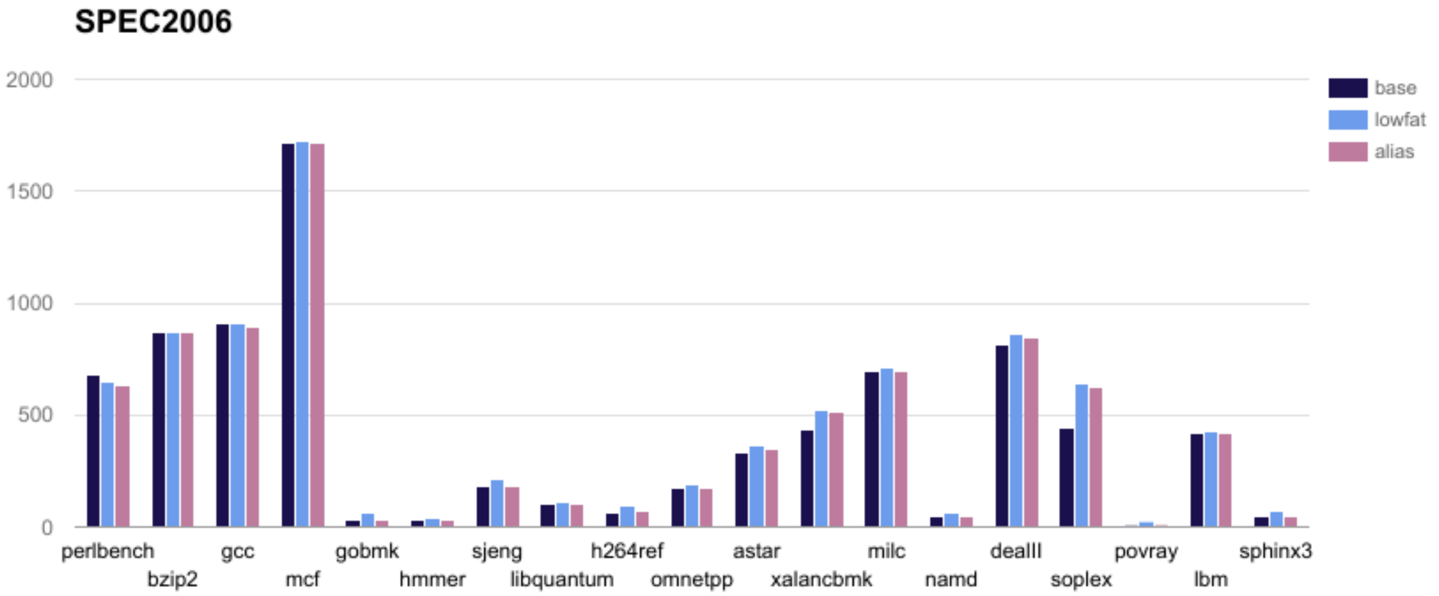
`shm_open`

Evaluation Basic (timings)



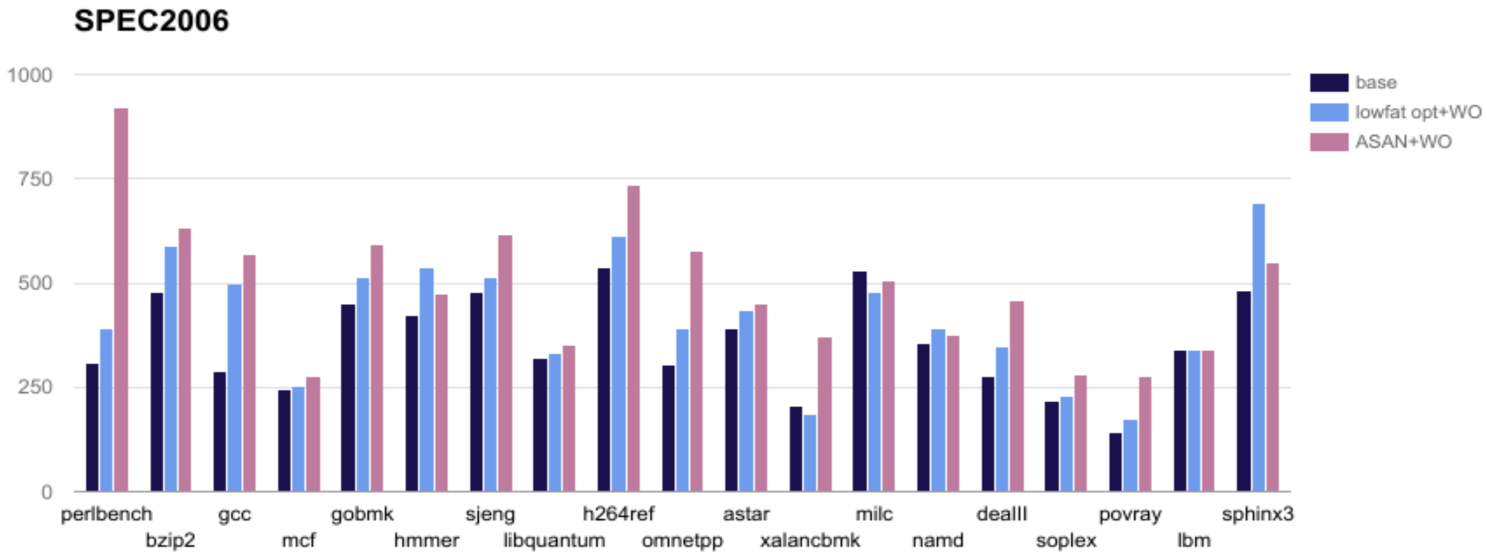
- Baseline: -O2
- Lowfat: 63% overhead (base unoptimized)
- Lowfat alias: 58% overhead with memory aliasing
- Address Sanitizer: 92% overhead

Evaluation (memory)



- 7% overhead
- 3% overhead with memory aliasing

Evaluation Timings Optimized (integrity/writes only [WO])



- Lowfat: 17% overhead
- Address Sanitizer (ASAN): 45% overhead

Summary and Conclusion

Low fat stack allocation effectively replaces:

```
sub %rax, %rsp
and $-16, %rsp
mov %rsp, %rbx
```

with

```
lzcnt %rax, %rax
sub SIZES(,%rax,8), %rsp
and MASKS(,%rax,8), %rsp
mov %rsp, %rbx
add OFFSETS(,%rax,8), %rbx
```

Extends protection to **stack objects (& heap)**

- Consequently also protects stack metadata

Desirable properties of low fat pointers preserved:

- Fast (~17% w.o.)
- Low space overheads (~3-15%)
- No metadata - **highly compatible**!