Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers

Thurston Dang[1], Petros Maniatis[2], David Wagner[1]

[1]University of California, Berkeley

[2]Google Brain

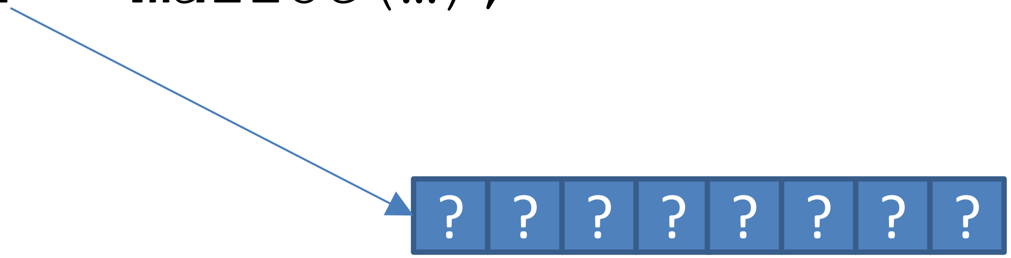
Overview

Provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme and no source code required

1. Temporal memory safety
2. Design Goals
3. Defenses
4. Our scheme and compatibility improvements
5. Empirical and theoretical evaluation

Temporal memory safety

```
aFuncPtr = malloc(…);
```



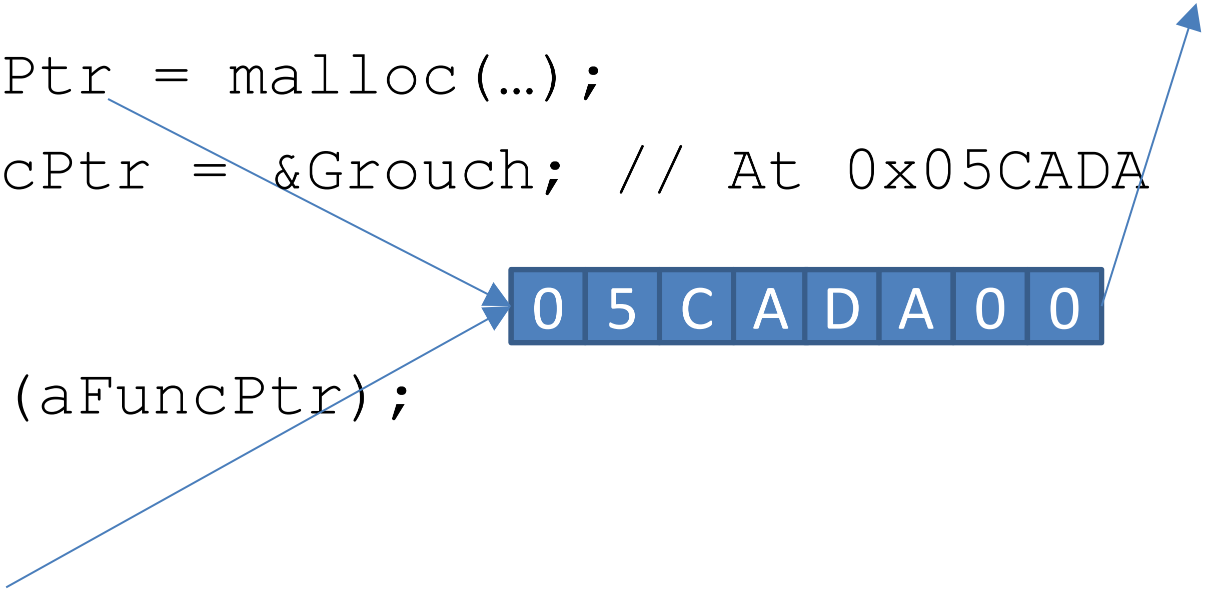
?	?	?	?	?	?	?	?

Temporal memory safety

```
aFuncPtr = malloc(…);
*aFuncPtr = &Grouch; // At 0x05CADA

free (aFuncPtr);

userName = malloc(…);
fgets(…); // Type in &Elmo (0xE11770)
```



```
Grouch() {
  …
}
```

Temporal memory safety

```
                                        ch() {
aFu
*al

                                        () {
fre

use
fge
```

```
(*someFuncPtr)(); // Use-after-free!
```

Overview

Provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme and no source code required

1. Temporal memory safety
2. Design Goals
3. Defenses
4. Our scheme and compatibility improvements
5. Empirical and theoretical evaluation

Design Goals

- No requirement for source code
- Deterministic protection
- Compatibility with typecasts, pointer arithmetic, and other common programming idioms
- Low runtime and memory overhead

Overview

Provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme and no source code required

1. Temporal memory safety
2. Design Goals
3. Defenses
4. Our scheme and compatibility improvements
5. Empirical and theoretical evaluation

DangNull

Dhurjati & Adve

Potato ?

What do these schemes do?

PageHeap

FreeSentry

How can we put them in a common framework?

CETS

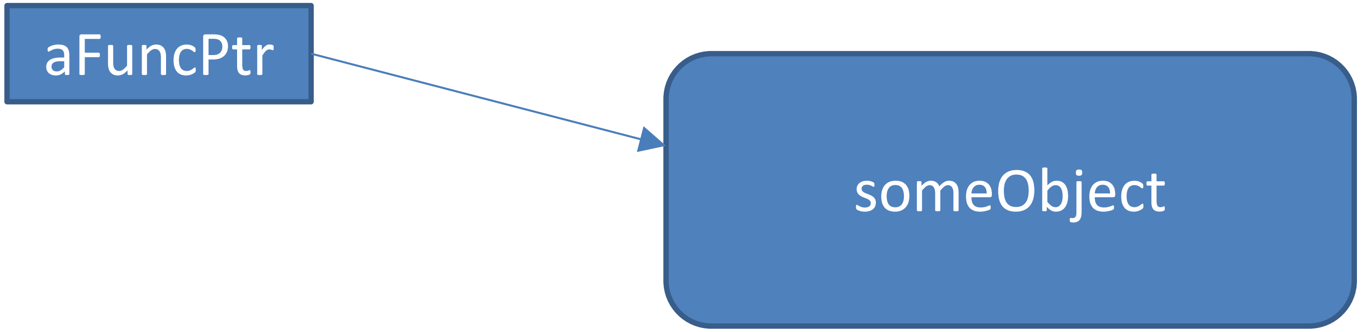
Electric Fence

DangSan

Berkeley
UNIVERSITY OF CALIFORNIA

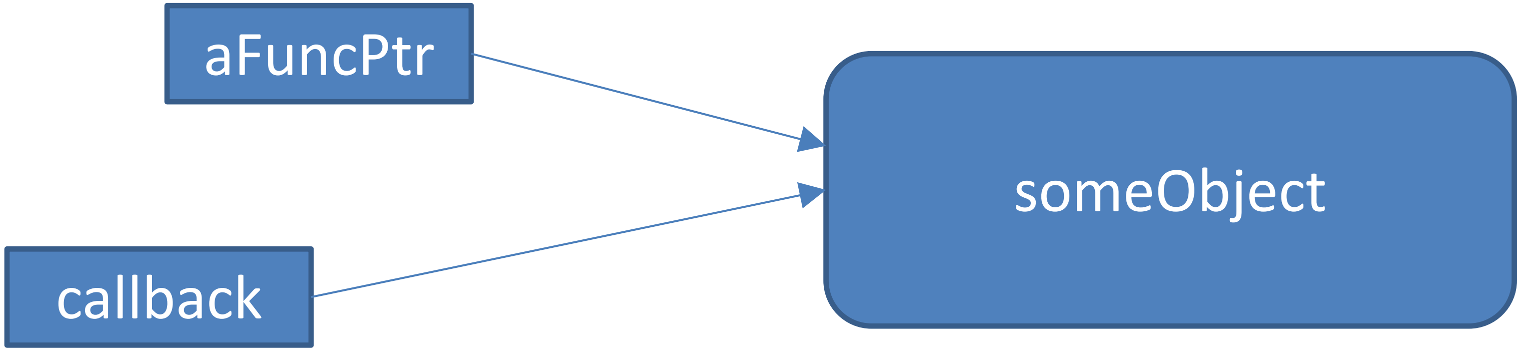
Toy example

```
aFuncPtr = malloc(…);
```



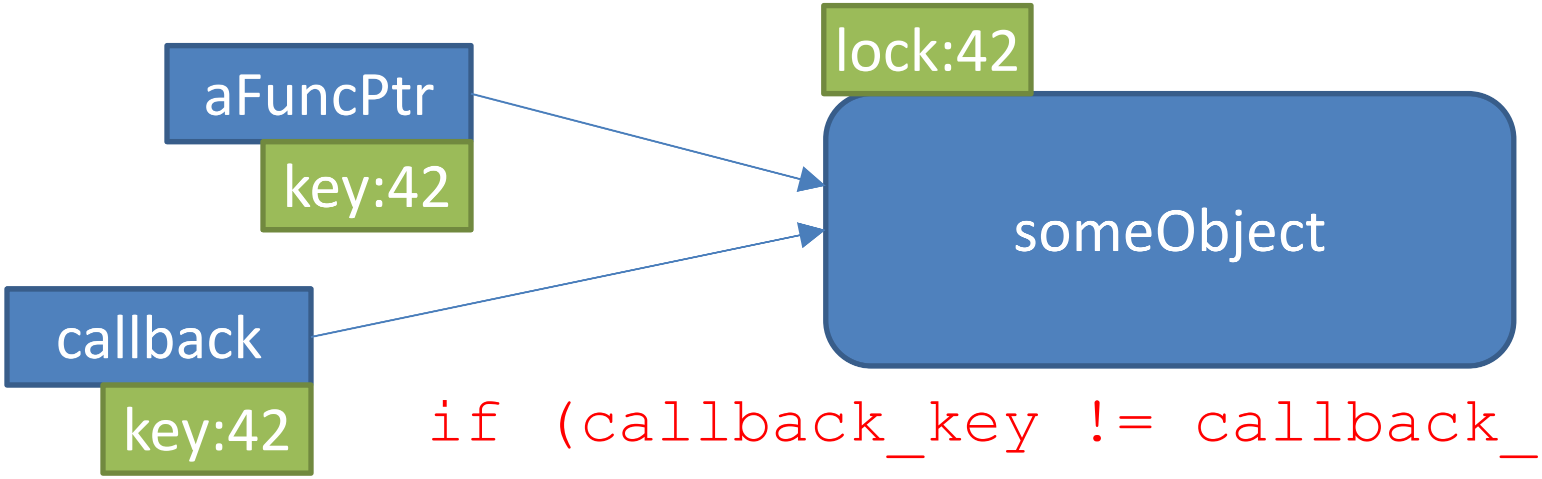
Toy example

```
aFuncPtr = malloc(…);
callback = someFuncPtr;
```



Scheme 1: lock-and-key schemes (change lock) e.g., CETS

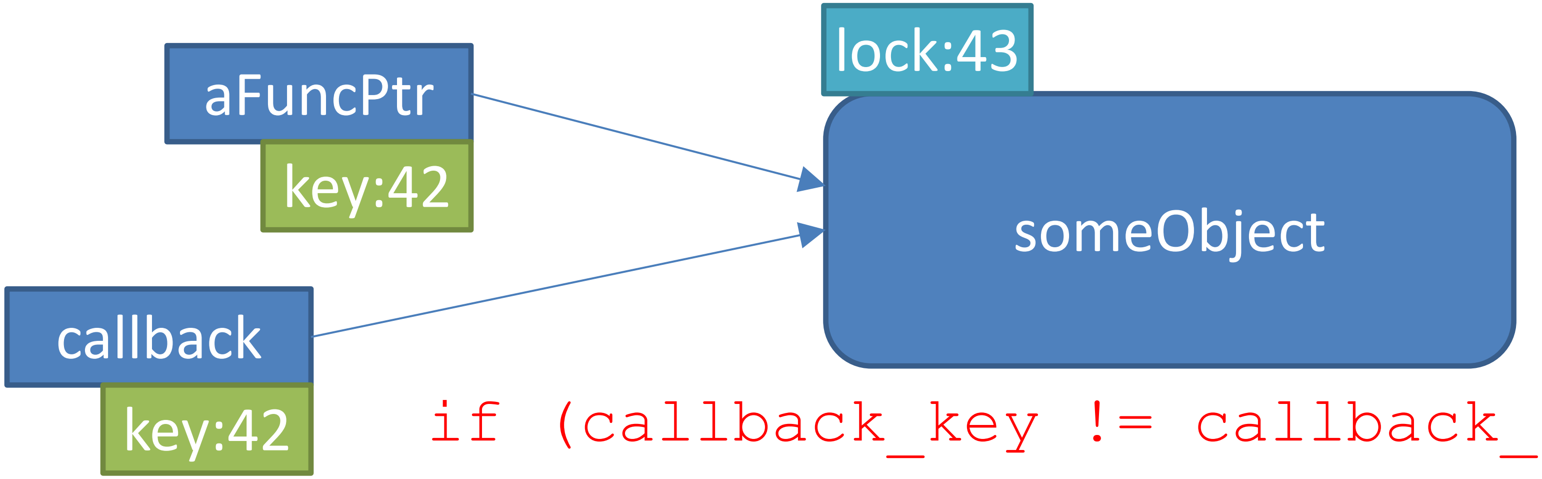
```
aFuncPtr = malloc(…); // Change lock
callback = someFuncPtr;
```



```
if (callback_key != callback_lock)
    abort()
            free (aFuncPtr);
```

Scheme 1: lock-and-key schemes (change lock) e.g., CETS

```
aFuncPtr = malloc(…); // Change lock
callback = someFuncPtr;
```

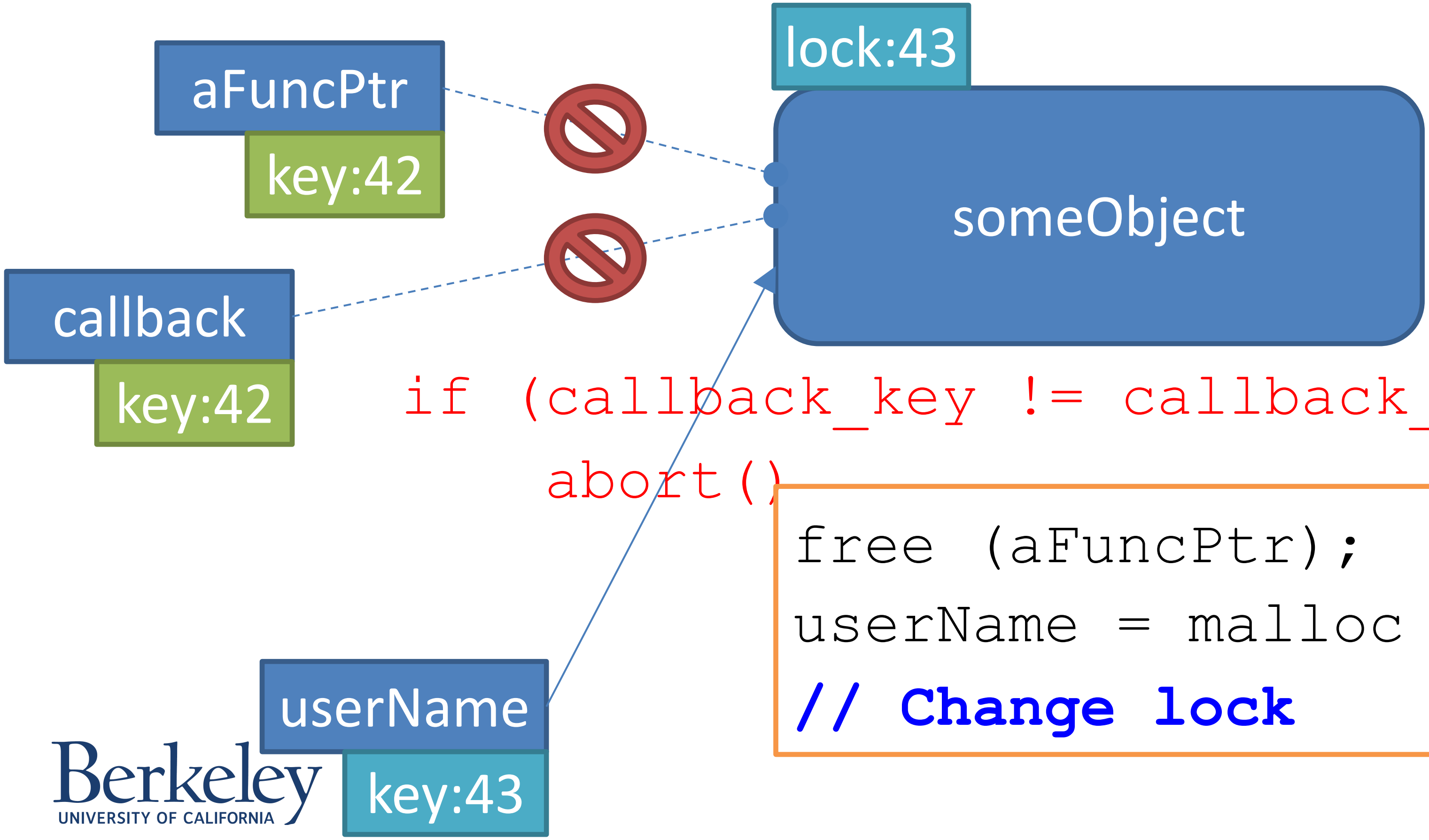


```
if (callback_key != callback_lock)
    abort()
```

```
free (aFuncPtr);
userName = malloc (…);
// Change lock
```

Scheme 1: lock-and-key schemes (change lock) e.g., CETS

```
aFuncPtr = malloc(…); // Change lock
callback = someFuncPtr;
```

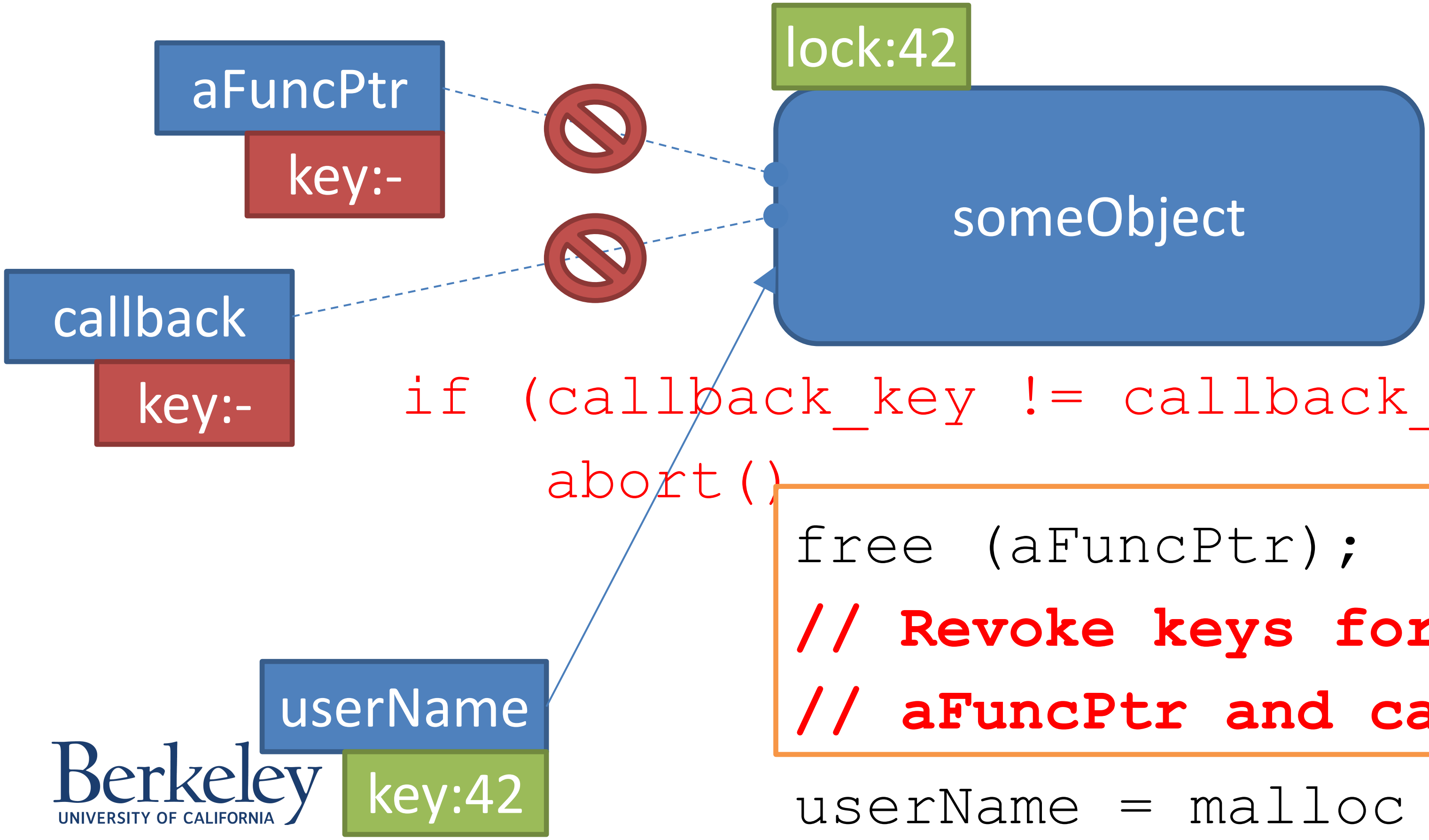


```
if (callback_key != callback_lock)
    abort()
```

```
free (aFuncPtr);
userName = malloc (…);
// Change lock
```

Scheme 2: lock-and-key schemes (revoke keys) [works, but slow]

```
aFuncPtr = malloc(…);
callback = someFuncPtr;
```



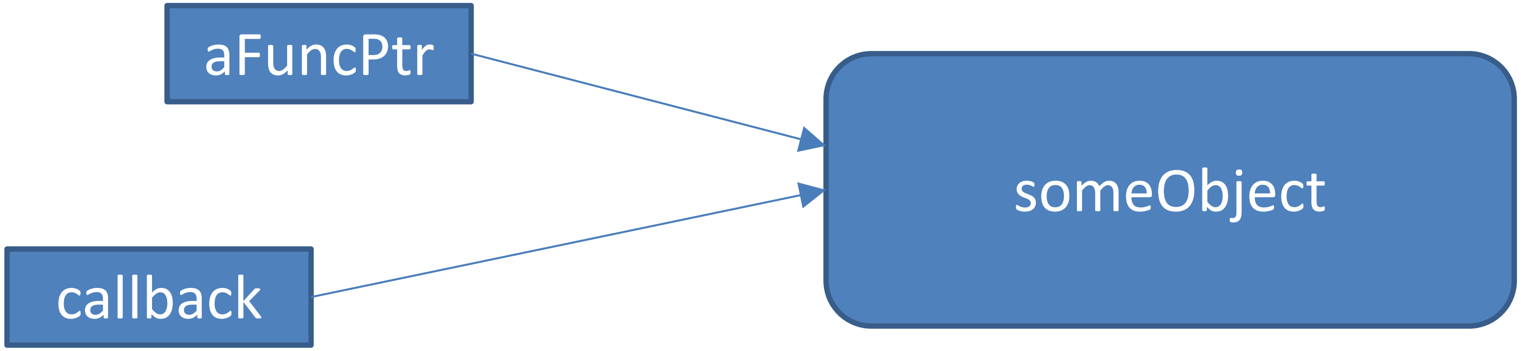
```
if (callback_key != callback_lock)
    abort()
```

```
free (aFuncPtr);
// Revoke keys for
// aFuncPtr and callback
```

```
userName = malloc (…);
```

Scheme 3: dangling pointer nullification e.g., DangNull, FreeSentry, DangSan

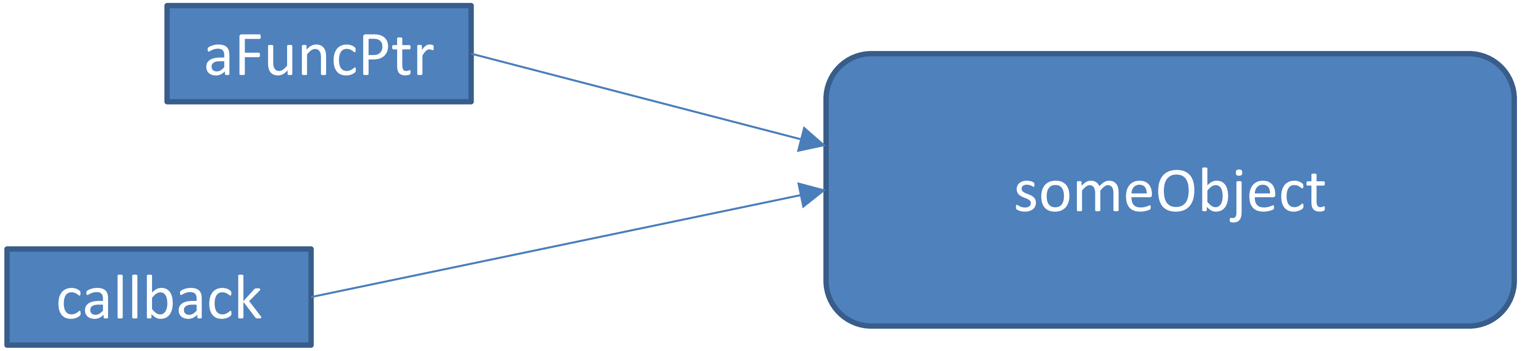
```
aFuncPtr = malloc(…);
callback = someFuncPtr;
```



```
// No pointer dereference checks
// added
                    free (aFuncPtr);
```

Scheme 3: dangling pointer nullification e.g., DangNull, FreeSentry, DangSan

```
aFuncPtr = malloc(…);
callback = someFuncPtr;
```



aFuncPtr

callback

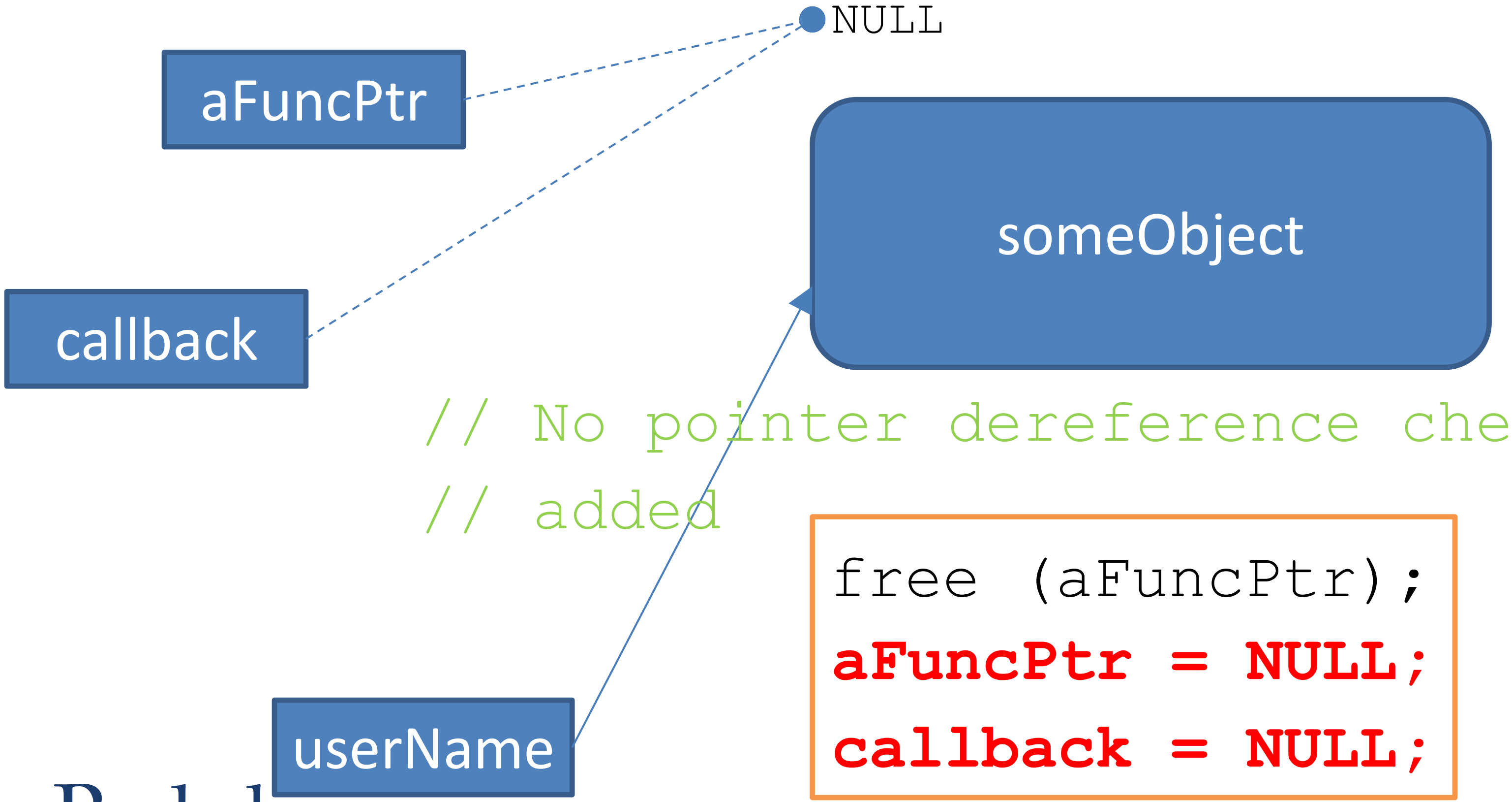
someObject

```
// No pointer dereference checks
// added
```

```
free (aFuncPtr);
aFuncPtr = NULL;
callback = NULL;
```

Scheme 3: dangling pointer nullification e.g., DangNull, FreeSentry, DangSan

```
aFuncPtr = malloc(…);
callback = someFuncPtr;
```



```
// No pointer dereference checks
// added
```

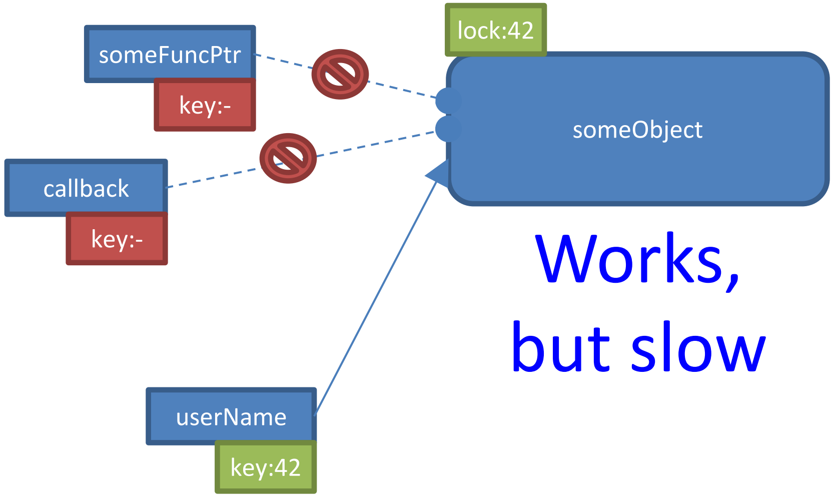
```
free (aFuncPtr);
aFuncPtr = NULL;
callback = NULL;
```

```
userName = malloc (…);
```

Software *ptr checks

Hardware *ptr checks

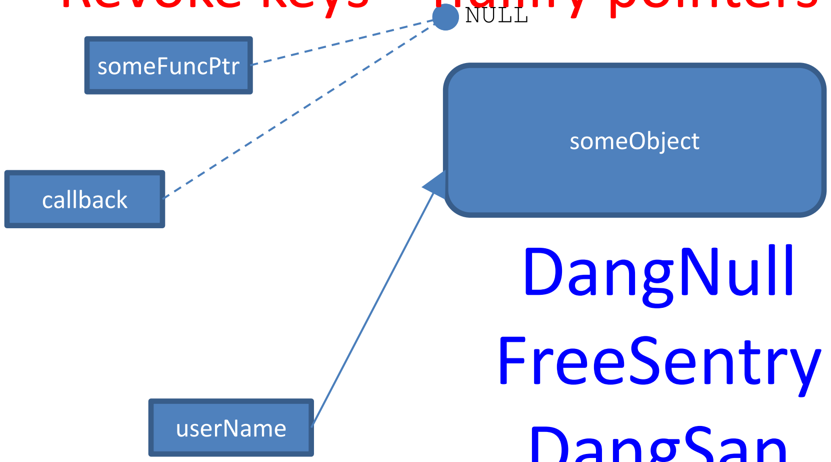
Revoke the keys



Works,
but slow

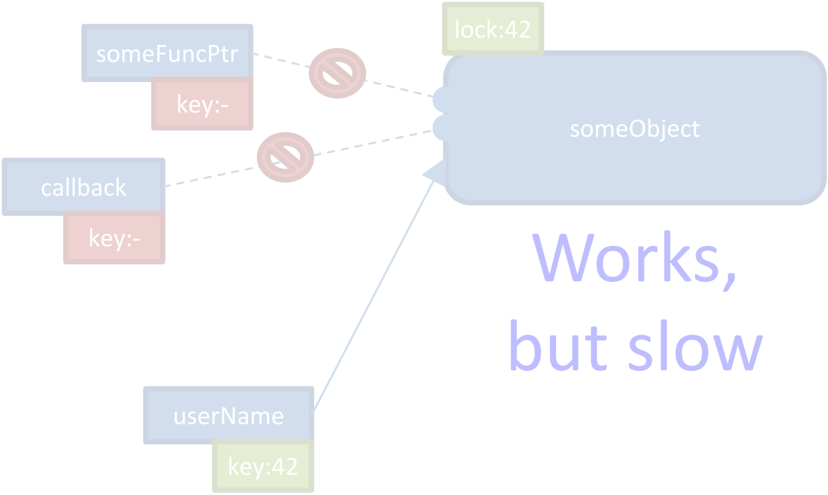
Key = pointer

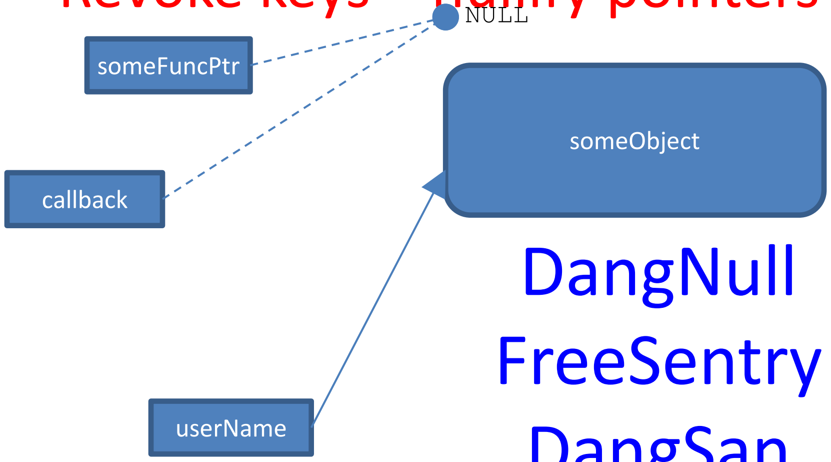
Revoke keys = nullify pointers

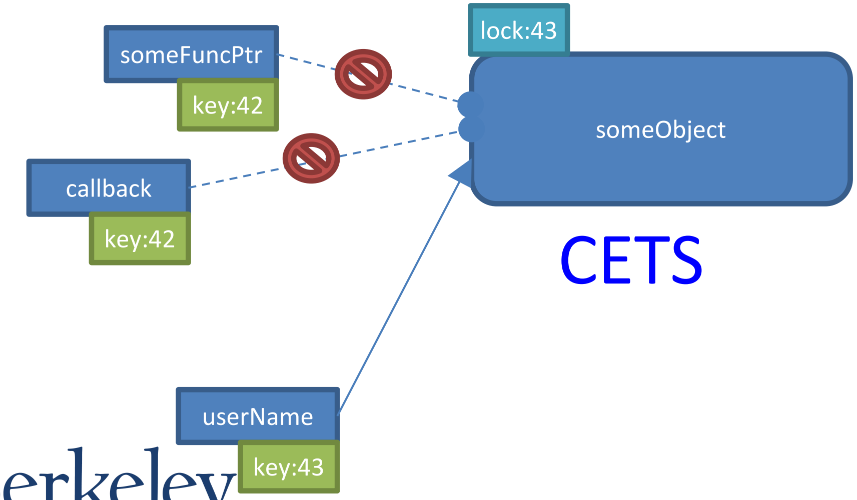


DangNull
FreeSentry
DangSan

	Software *ptr checks	Hardware *ptr checks
Revoke the keys	Works, but slow	Key = pointer Revoke keys = nullify pointers DangNull FreeSentry DangSan
Change the lock	CETS	Object = lock Change the lock = ??? *Page-permissions-based schemes* ???

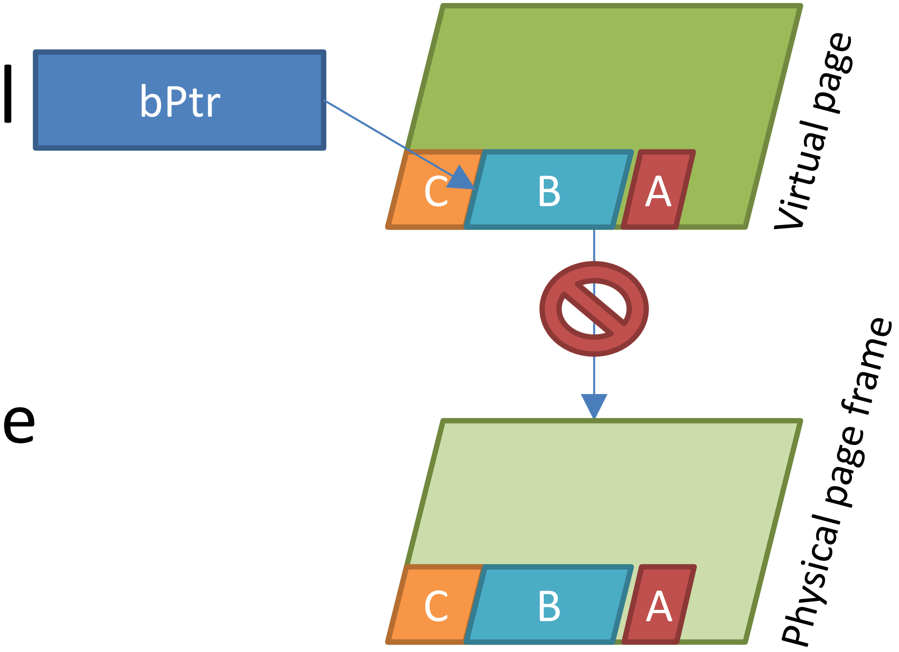






Recall that objects are accessed via *virtual addresses*

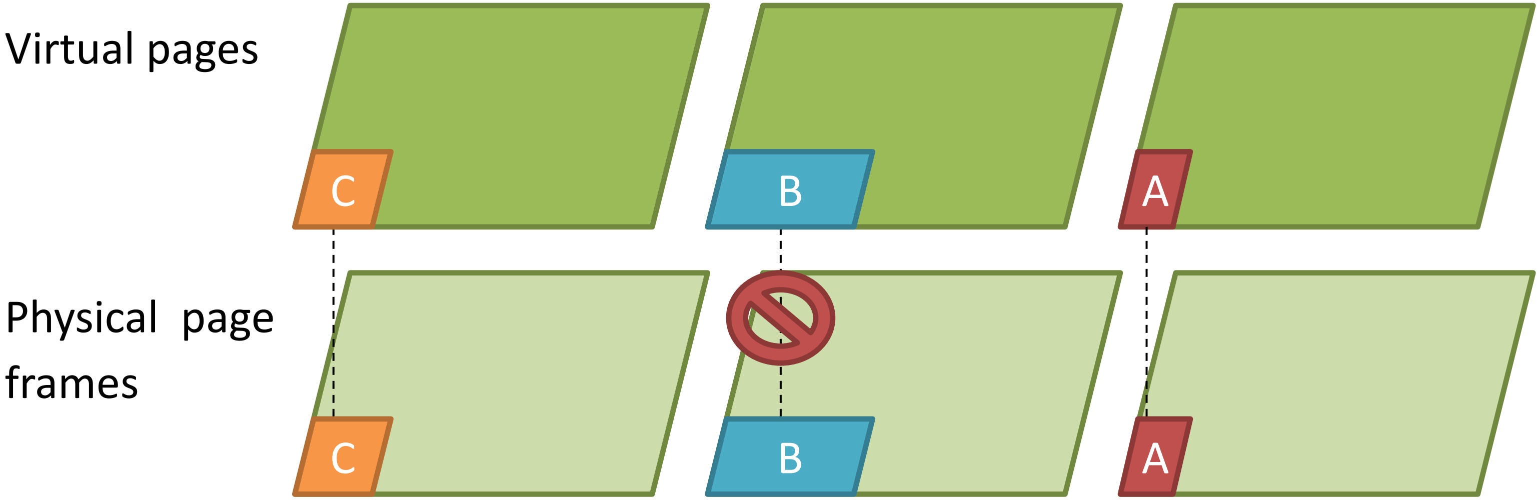
- We can mark a virtual *page* as inaccessible
  - 4KB-page granularity
  - Many objects per page



Berkeley UNIVERSITY OF CALIFORNIA

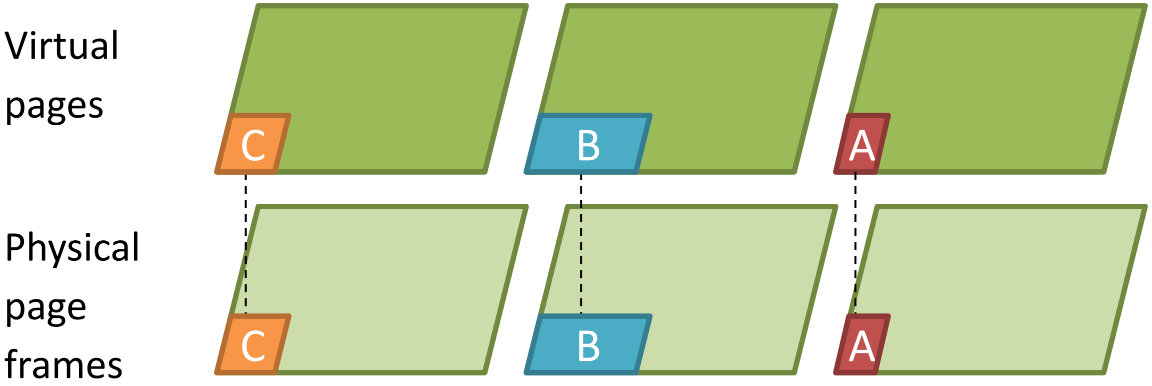
Scheme 4: page table protections e.g., Electric Fence, PageHeap

- One object per page



Scheme 4: page table protections e.g., Electric Fence, PageHeap

- One object per page

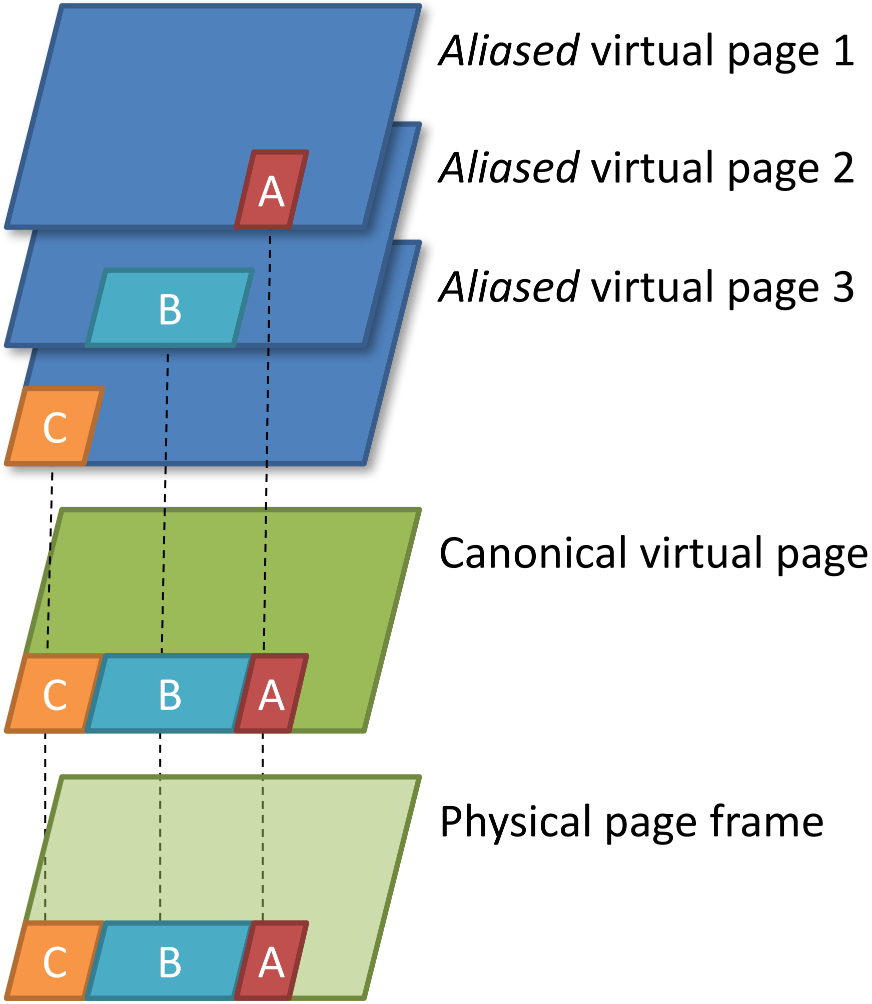


- Drawbacks:
  - Inefficient use of physical memory (+ cache)
  - Many system calls (to update virtual page mappings)
  - TLB pressure

Berkeley
UNIVERSITY OF CALIFORNIA

Scheme 4+: page table protections revisited e.g., Dhurjati & Adve

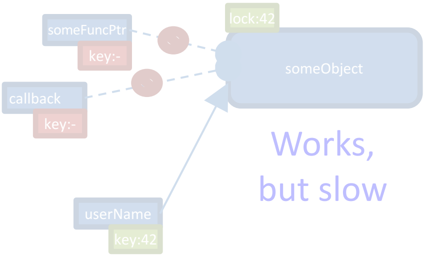
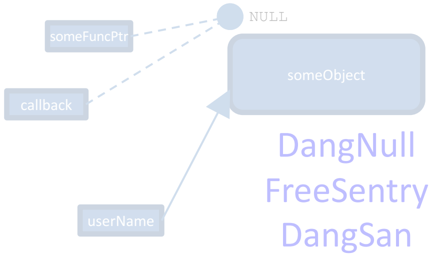
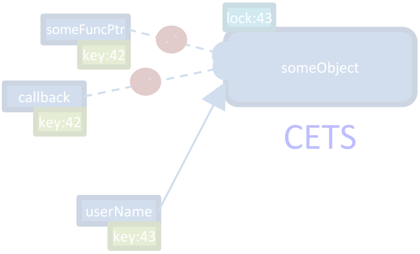
- Drawbacks:
  - ~~Inefficient use of physical memory~~
  - Many system calls (to update aliased page mappings)
  - TLB pressure



Dealing with freed objects

- Vanilla approach keeps memory mappings (`PROT_NONE`) for "freed" objects
  - ✘ memory leak (`vm_area_struct`s)

- Dhurjati & Adve use "automatic pool allocation"
  - ✘ requires source code analysis
  - ✘ can suffer from long-lived pools

	Software *ptr checks	Hardware *ptr checks
Revoke the keys	 Works, but slow	 DangNull FreeSentry DangSan
Change the lock	 CETS	Page-permissions-based schemes e.g., Electric Fence, PageHeap, Dhurjati & Adve Theoretically: best approach Conventional wisdom: impractical **Oscar**

Overview

Provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme and no source code required

1. Temporal memory safety
2. Design Goals
3. Defenses
4. Our scheme and compatibility improvements
5. Empirical and theoretical evaluation

Our Design

- Builds upon core idea of page permissions with aliased virtual pages, with:

a) no requirement for source code

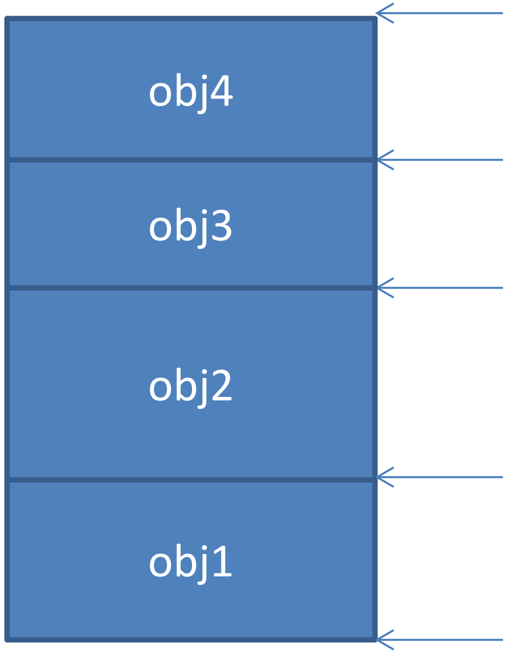
b) less stateholding of kernel metadata for freed objects

c) better compatibility with fork()

d) optimizations to reduce runtime overhead

Handling long-lived applications (b)

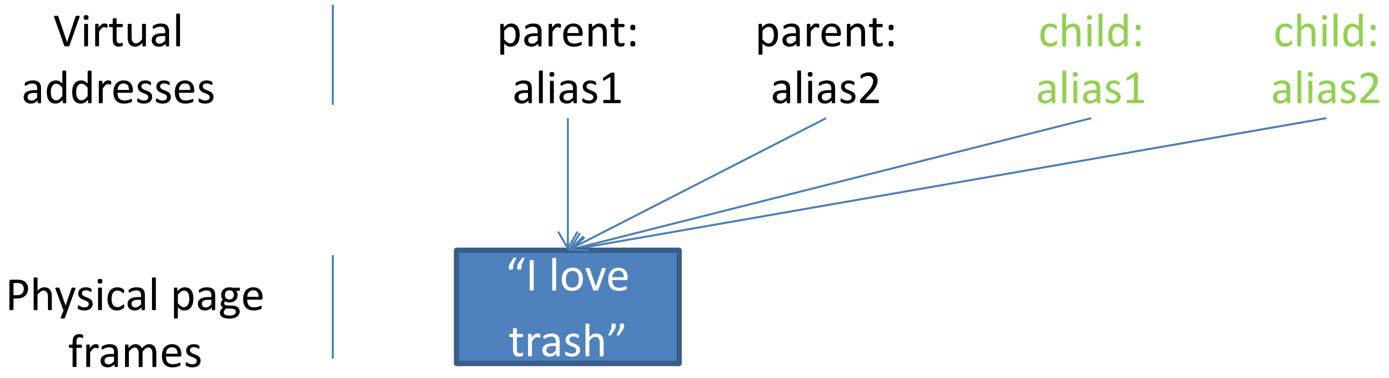
- We keep a high water mark for allocations
  - will never allocate new objects in address space of old objects



- Lower memory overhead: no `vm_area_struct`s for freed objects

Correct semantics for fork() (c)

- We need *“MAP_SHARED”* to create aliases
- Unwanted “side-effect”: parent and child will share physical memory



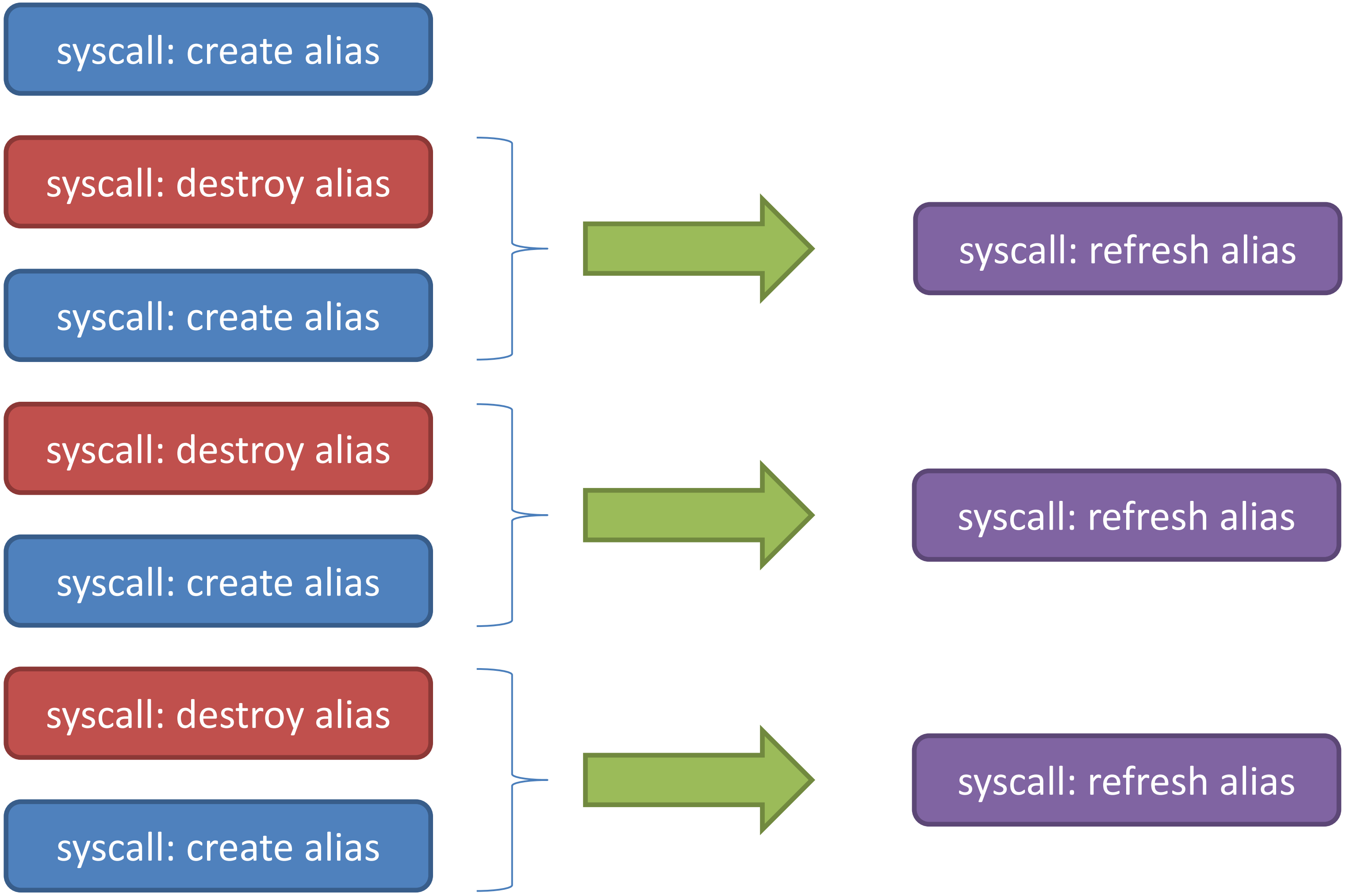
  – To our knowledge, not discussed in prior work

- We solve this with a fork() wrapper that weans the child from the parent’s physical frames

Making large allocations faster (d)

- We need MAP_SHARED to create aliases
- Another unwanted side-effect of MAP_SHARED: slows down some programs
  - e.g., on ‘mcf’, this has ~30% overhead
- Very large objects are placed by malloc() on their own physical page frames
  - i.e., no aliases required
- We use MAP_PRIVATE for those large objects
  - on ‘mcf’: < 1% overhead

Reducing syscalls by *refreshing* aliases (d)



syscall: create alias

syscall: destroy alias

syscall: create alias

syscall: refresh alias

syscall: destroy alias

syscall: create alias

syscall: refresh alias

syscall: destroy alias

syscall: create alias

syscall: refresh alias

…

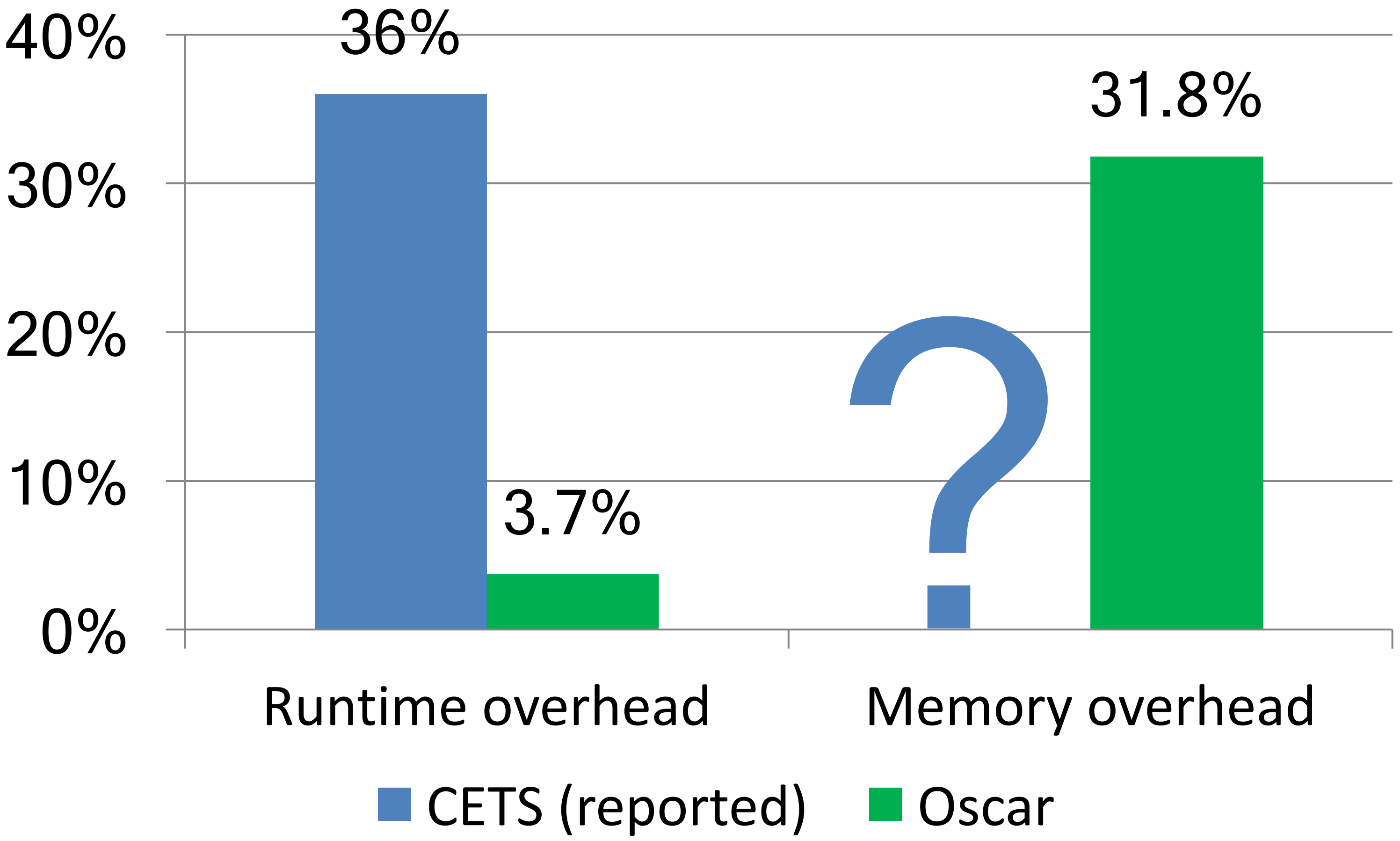
Overview

Provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme and no source code required

1. Temporal memory safety
2. Design Goals
3. Defenses
4. Our scheme and compatibility improvements
5. Empirical and theoretical evaluation

Overhead vs. CETS* (2010)

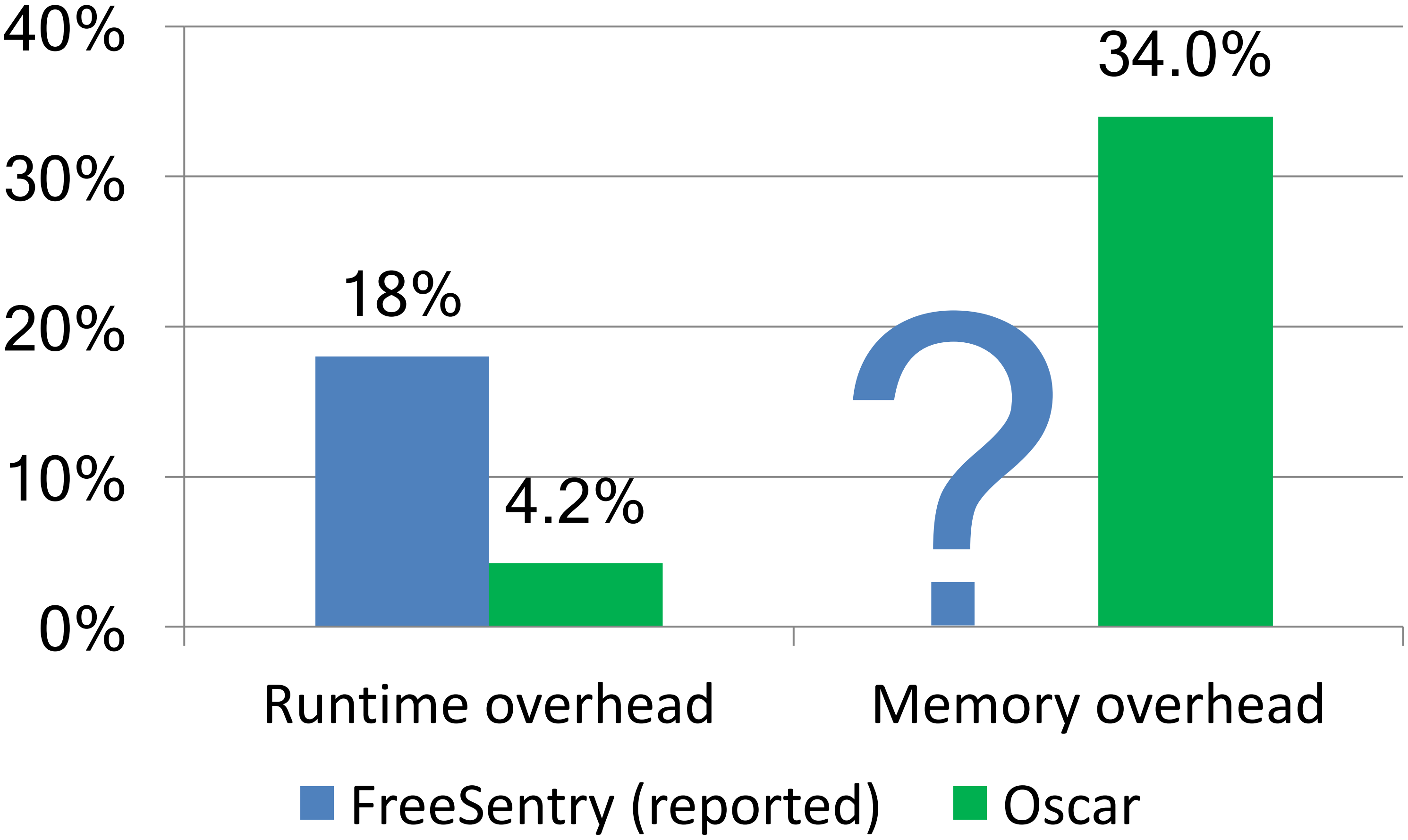
*actually, SoftBoundCETS temporal-only, which is faster than CETS



N.B. CETS includes stack use-after-free protection

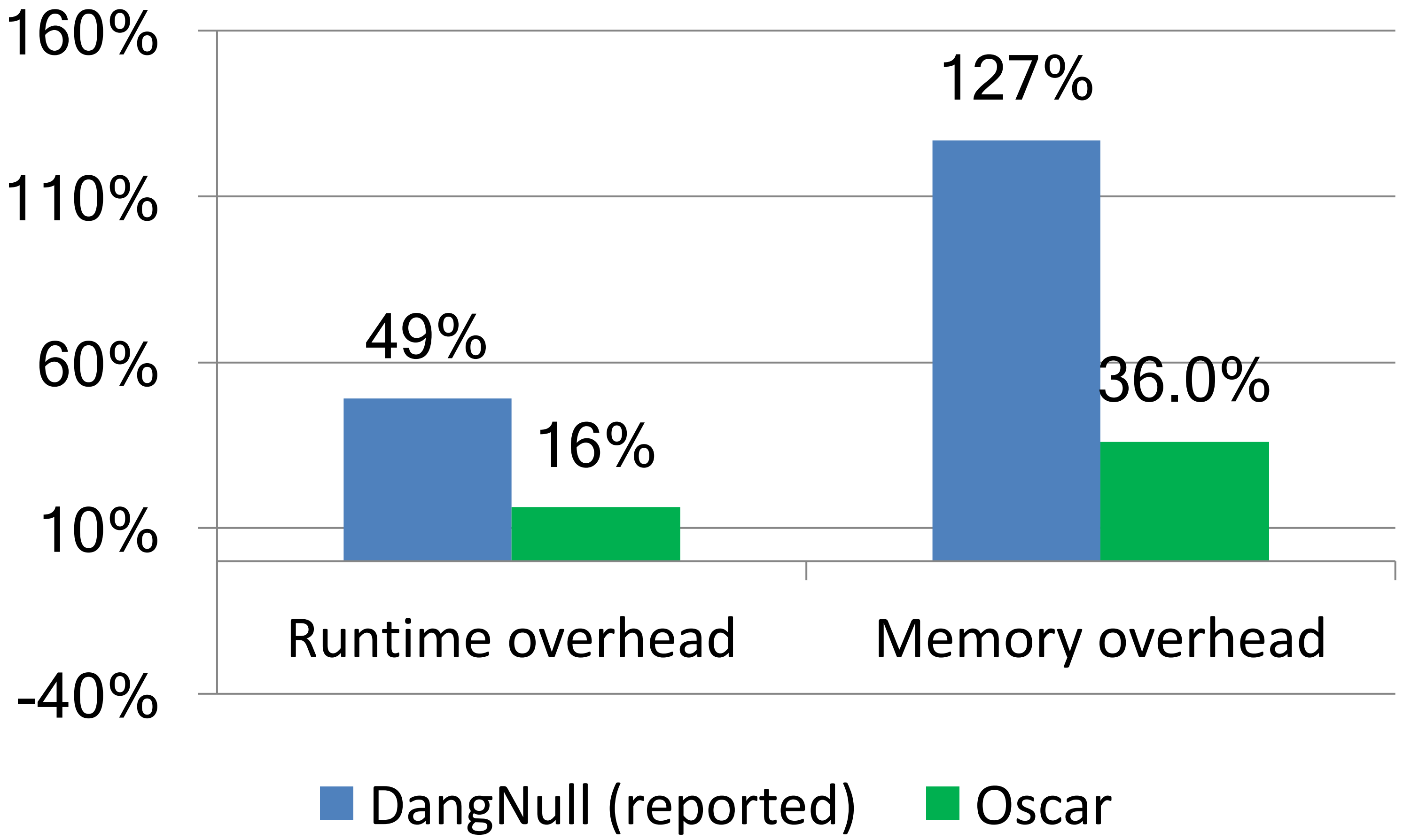
Memory overhead not reported by CETS

Overhead vs. FreeSentry (2015)



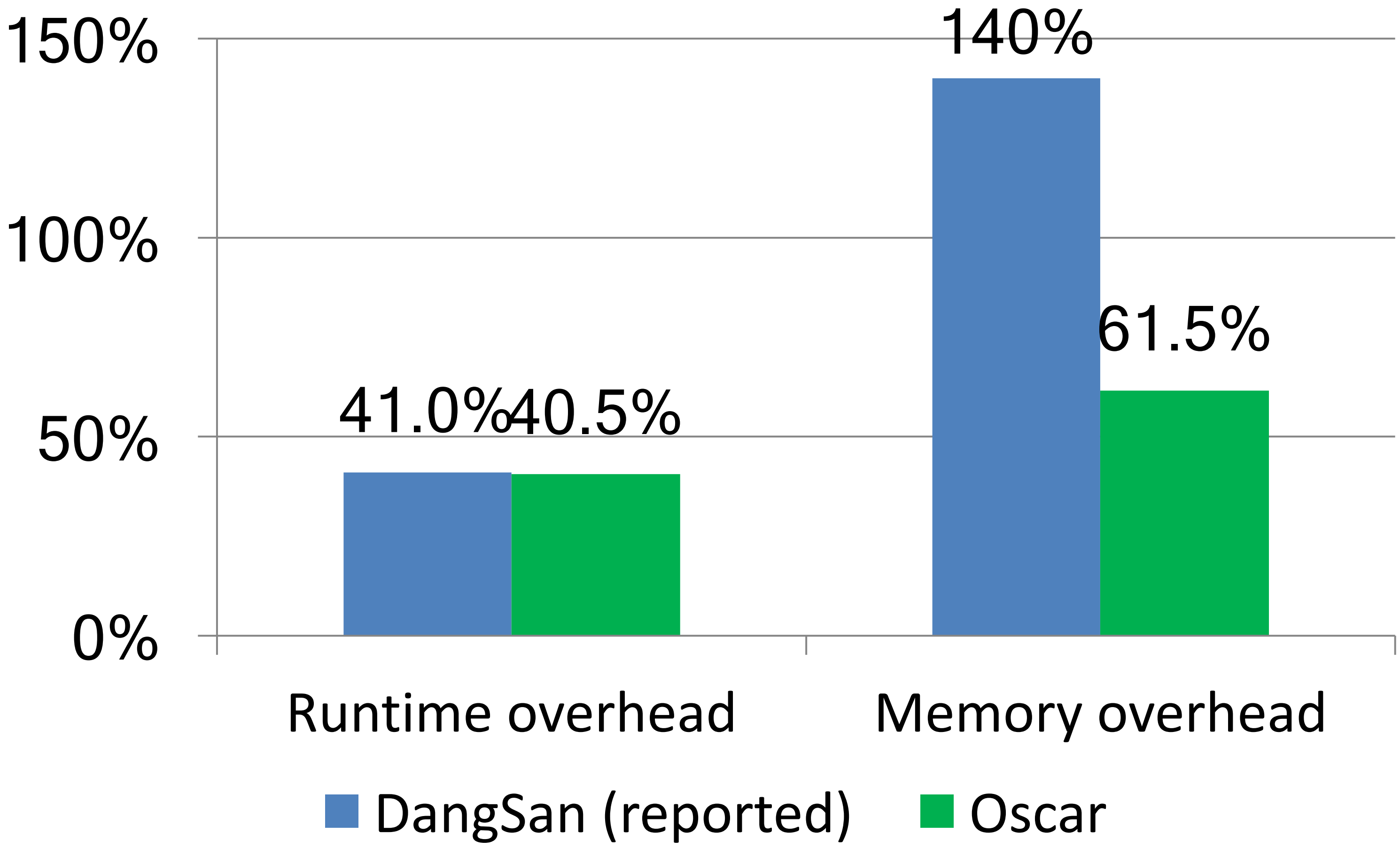
Memory overhead not reported by FreeSentry

Overhead vs. DangNull (2015)



N.B. DangNull provides weaker protection (pointers stored on heap)

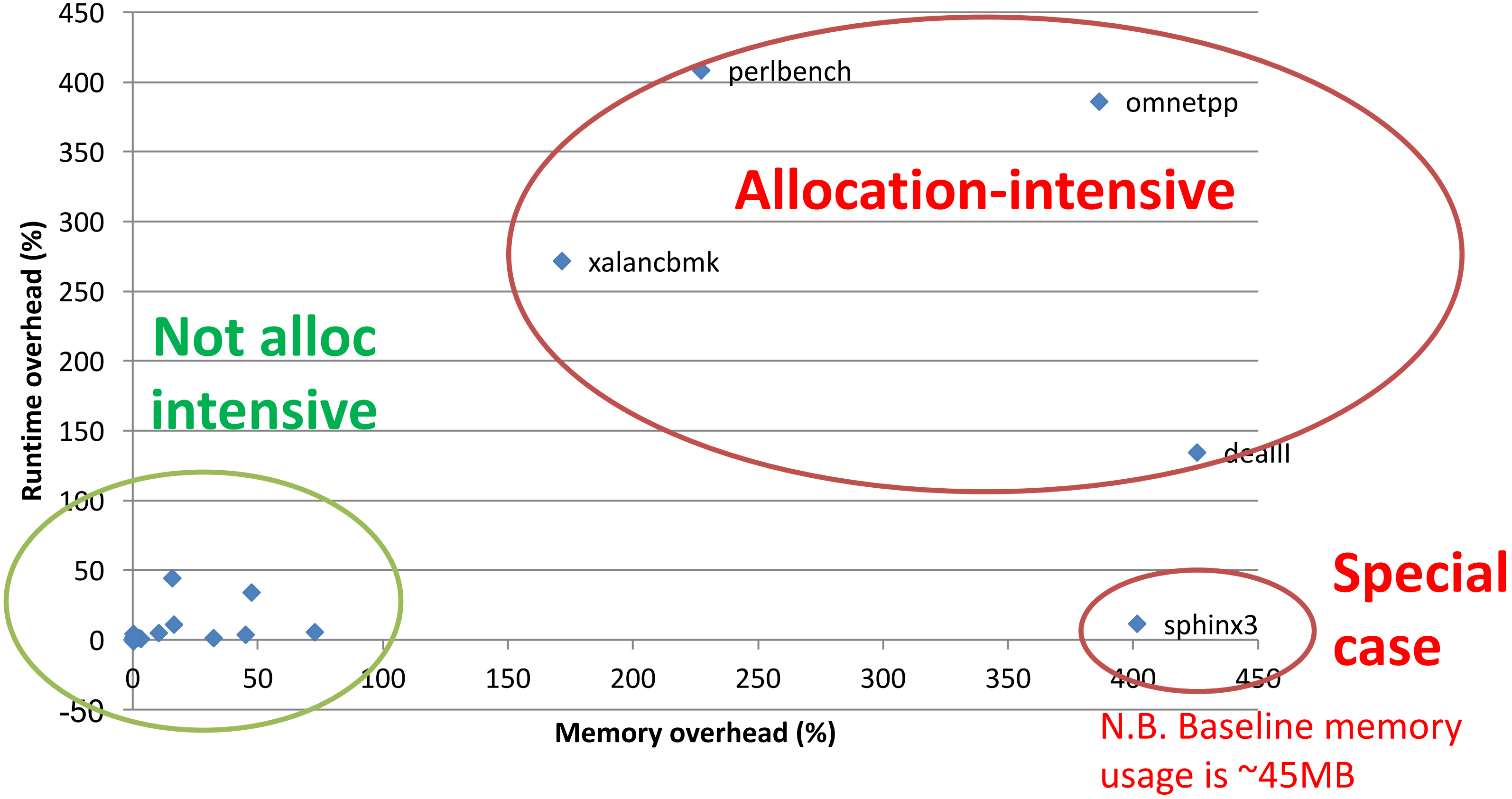
Overhead vs. DangSan (2017)



DangSan and Oscar report all non-Fortran SPEC CPU2006 benchmarks

Berkeley
UNIVERSITY OF CALIFORNIA

Oscar's runtime overhead, memory overhead (and VMA exhaustion: not shown) are correlated



N.B. Baseline memory usage is ~45MB

- Apps with few malloc calls generally have low runtime overhead, low memory overhead, low rate of exhausting virtual address space

Intuition

- Oscar’s overhead is proportional to the number of *objects*
  - other classes of schemes have memory and runtime overhead proportional to the number of *pointers* or pointer operations
- Oscar doesn’t need nor instrument the source code, so it avoids edge cases (e.g., typecasts)
  - other classes of schemes need source and have some compatibility issues

Additional details in paper

- Sources of overhead
- Server support: fork(), custom memory allocators
- Theoretical comparison of schemes

Conclusion

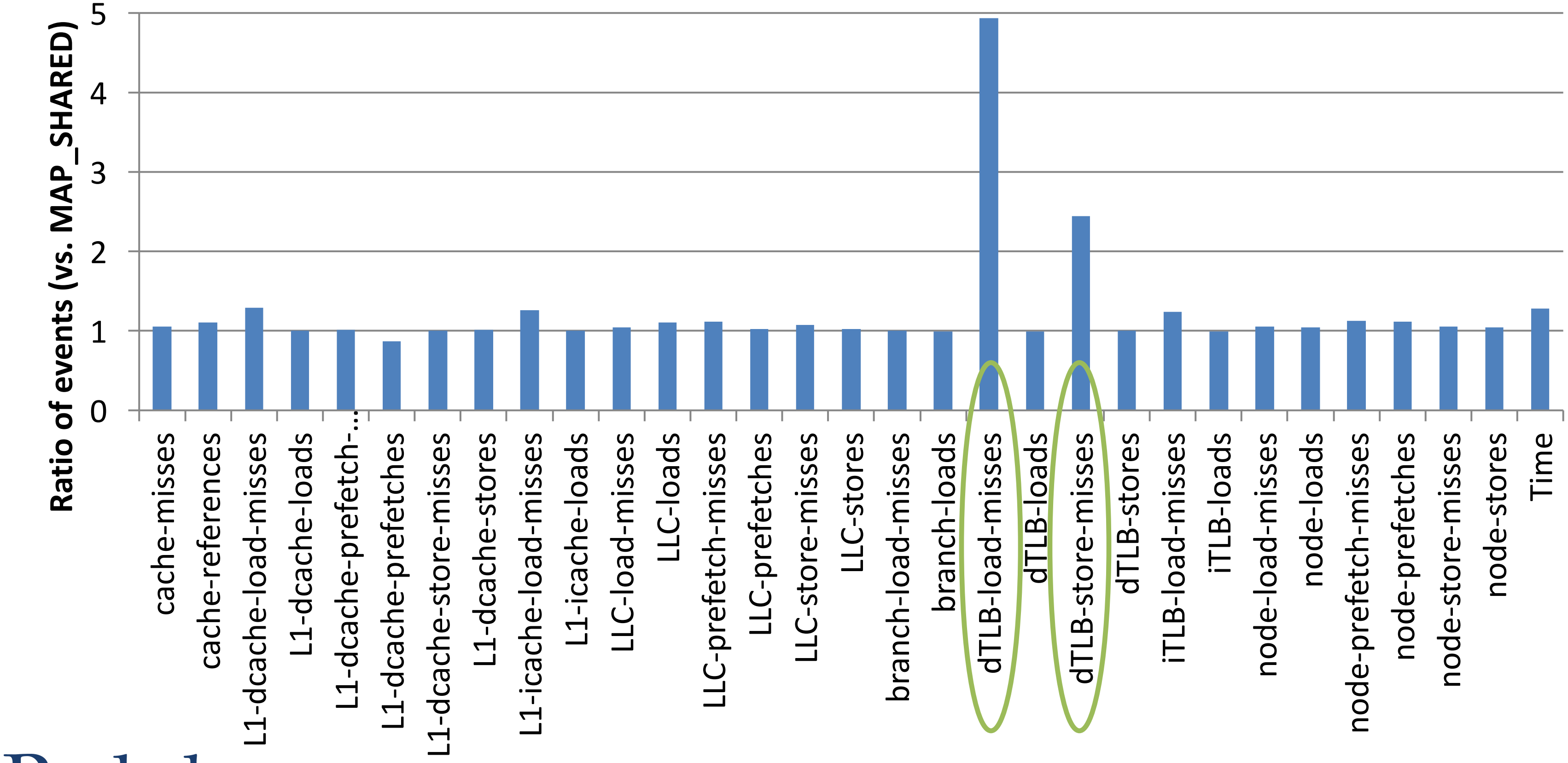
- Oscar provides heap temporal memory safety for C/C++ with lowest overhead of any published scheme
  - no source code required
- Bring about page-permissions based schemes to be worthy of consideration once more

References

- [CETS] NAGARAKATTE, S. G. Practical low-overhead enforcement of memory safety for C programs. University of Pennsylvania, 2012. Doctoral dissertation.
- [DangNull] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing Use-after-free with Dangling Pointers Nullification. In NDSS (2015).
- [DangSan] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: Scalable Use-after-free Detection. In EuroSys (2017), pp. 405–419.
- [Dhurjati & Adve] DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In Dependable Systems and Networks (2006), IEEE, pp. 269–280.
- [Electric Fence] Electric Fence. http://elinux.org/index.php?title=Electric_Fence&oldid=369914, January 2015.
- [FreeSentry] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In NDSS (2015).
- [PageHeap] How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003. https://support.microsoft.com/en-us/kb/286470.

mcf: MAP_SHARED

- MAP_PRIVATE: “Create a private copy-on-write mapping”
- ‘perf’ hardware performance counters:


Ratio of events (vs. MAP_SHARED)
cache-misses
cache-references
L1-dcache-load-misses
L1-dcache-loads
L1-dcache-prefetch-...
L1-dcache-prefetches
L1-dcache-store-misses
L1-dcache-stores
L1-icache-load-misses
L1-icache-loads
LLC-load-misses
LLC-loads
LLC-prefetch-misses
LLC-prefetches
LLC-store-misses
LLC-stores
branch-load-misses
branch-loads
dTLB-load-misses
dTLB-loads
dTLB-store-misses
dTLB-stores
iTLB-load-misses
iTLB-loads
node-load-misses
node-loads
node-prefetch-misses
node-prefetches
node-store-misses
node-stores
Time