CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++

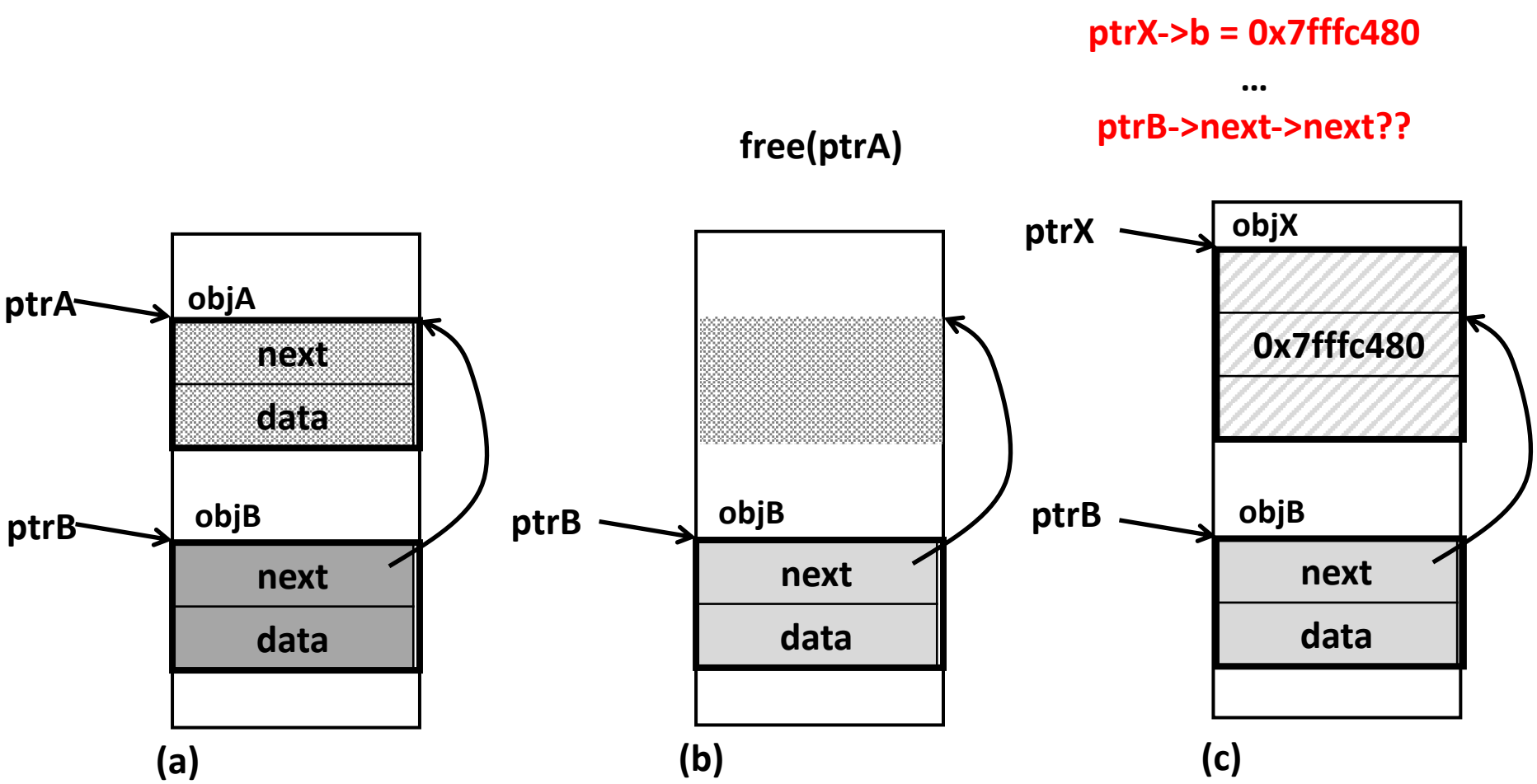
Seoul National University

Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yunheung Paek

Soongsil University

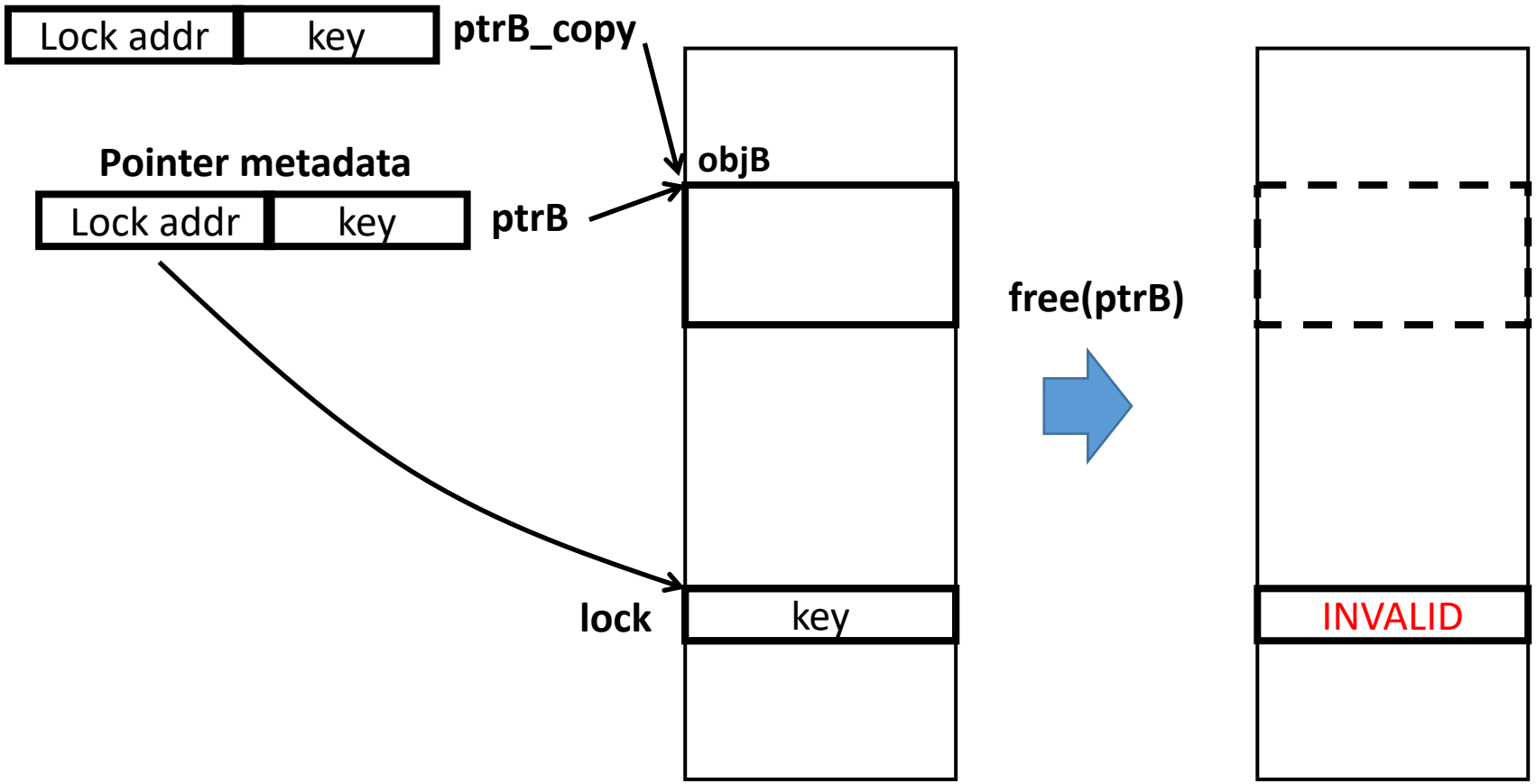
Yeongpil Cho

Use-After-Free (UAF)



UAF Defenses – Access Validation

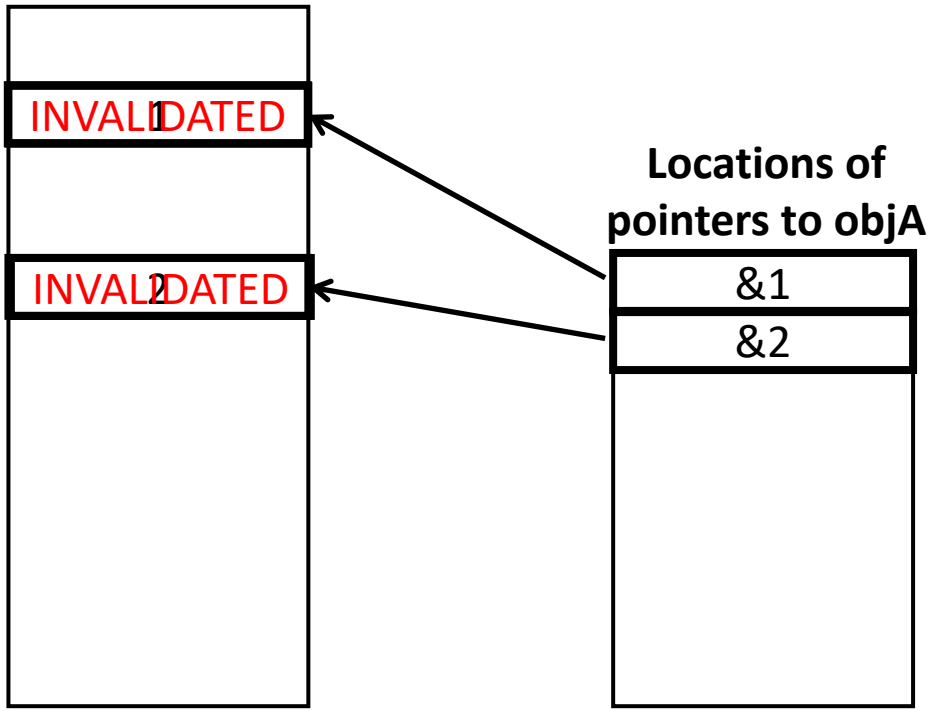
- Check every memory access



CETS: Compiler-Enforced Temporal Safety for C, ACM Sigplan Notices, 2010

UAF Defenses – Pointer invalidation

- Invalidate pointers on free()
  - Track only when a pointer is stored, not on every memory access



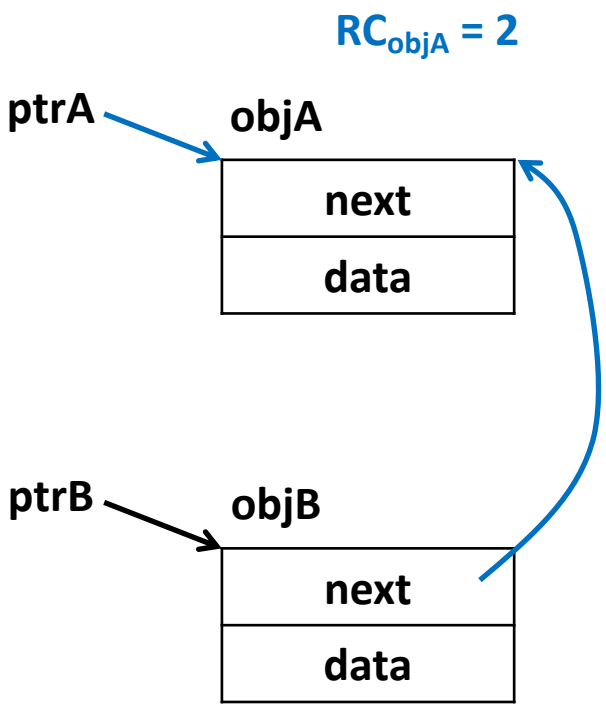
SPEC2006
Runtime + 44%
Memory + 126%

Too high for runtime protection

Dangsan: Scalable useafter-free detection, EuroSys 2017

Revisit reference counting

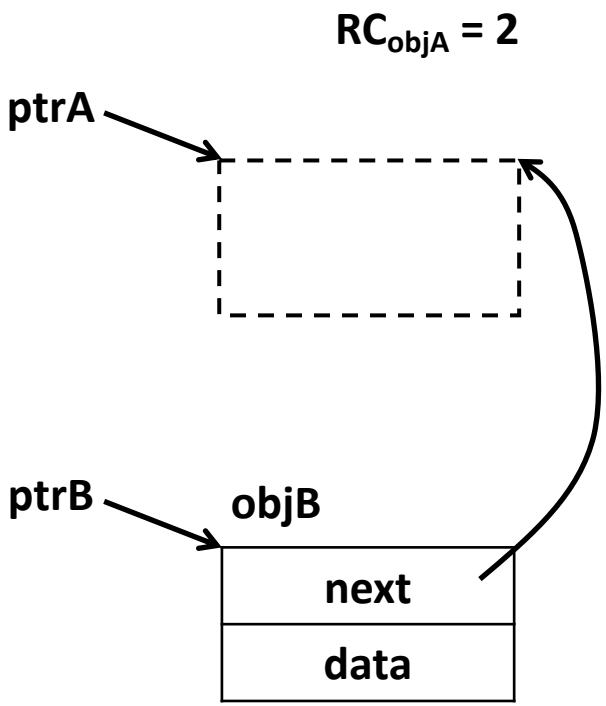
- Goal: free memory only when all the dangling pointers are gone




```
1  struct node { struct node *next; int data; };
2  struct node *ptrA, *ptrB;
3
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

Revisit reference counting

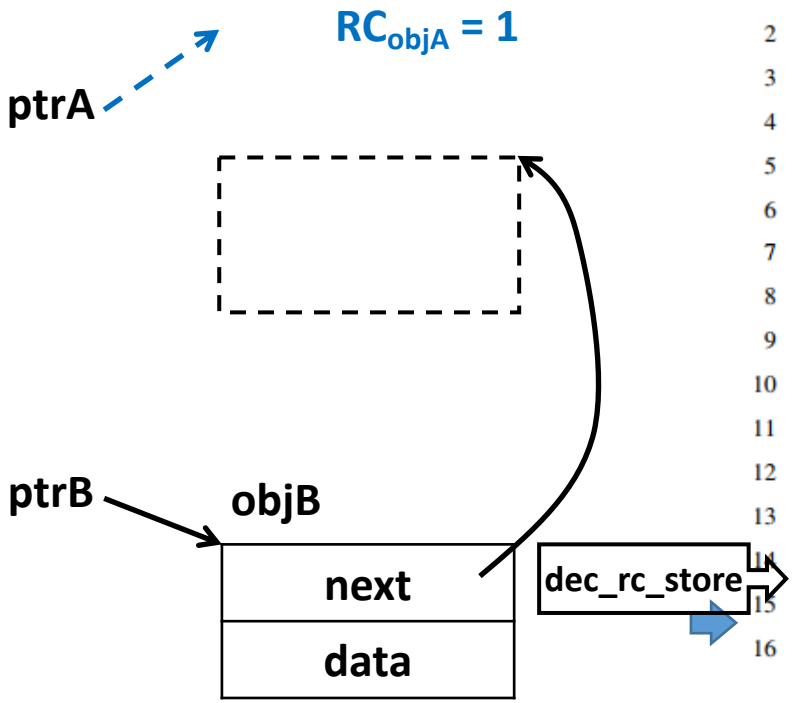
- Goal: free memory only when all the dangling pointers are gone




```
1  struct node { struct node *next; int data; };
2  struct node *ptrA, *ptrB;
3
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

Revisit reference counting

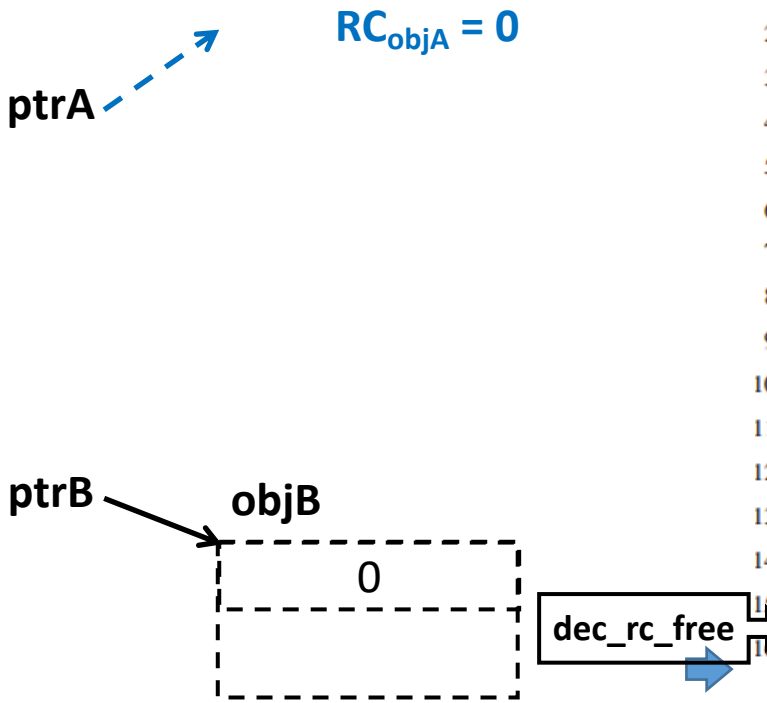
- Goal: free memory only when all the dangling pointers are gone




```
1  struct node { struct node *next; int data; };
2  struct node *ptrA, *ptrB;
3
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

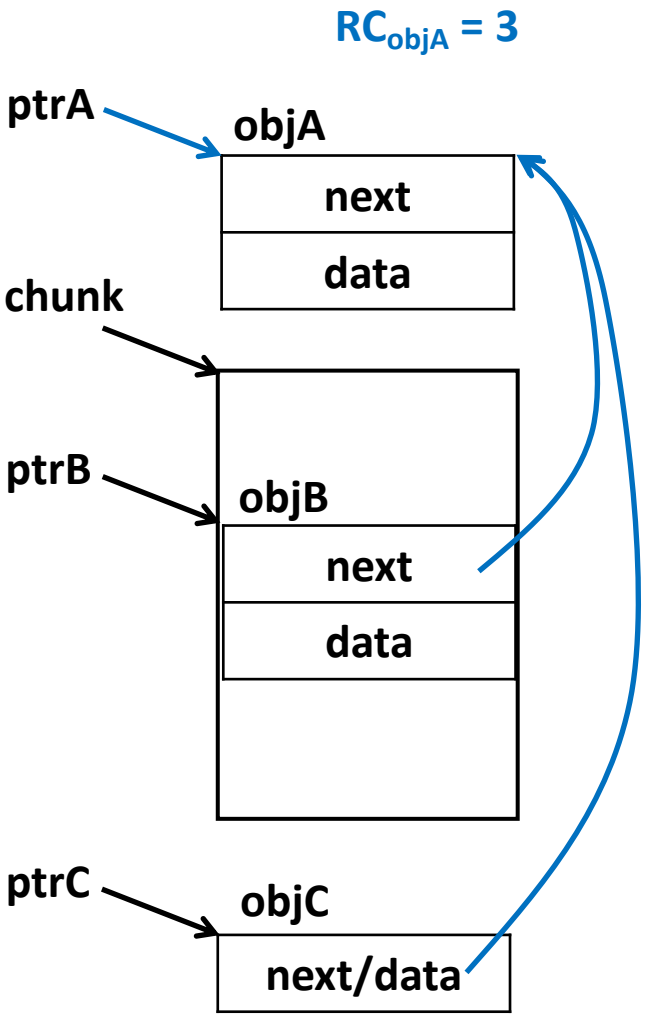
Revisit reference counting

- Goal: free memory only when all the dangling pointers are gone



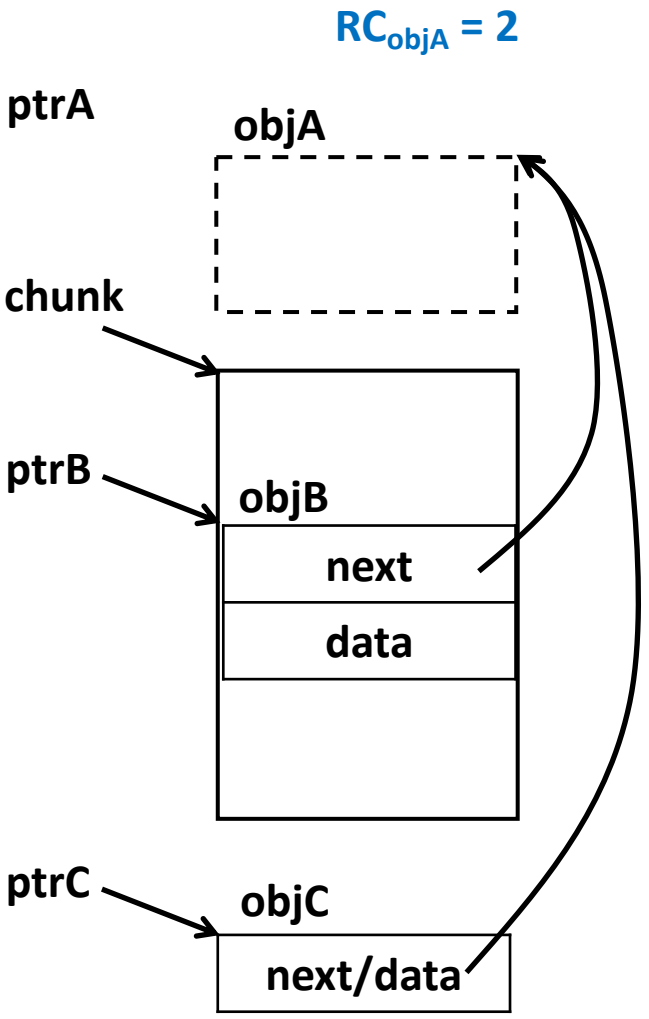

```
1  struct node { struct node *next; int data; };
2  struct node *ptrA, *ptrB;
3
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

C Reference counting challenges




```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);
20 ptrC->data = 1;
```

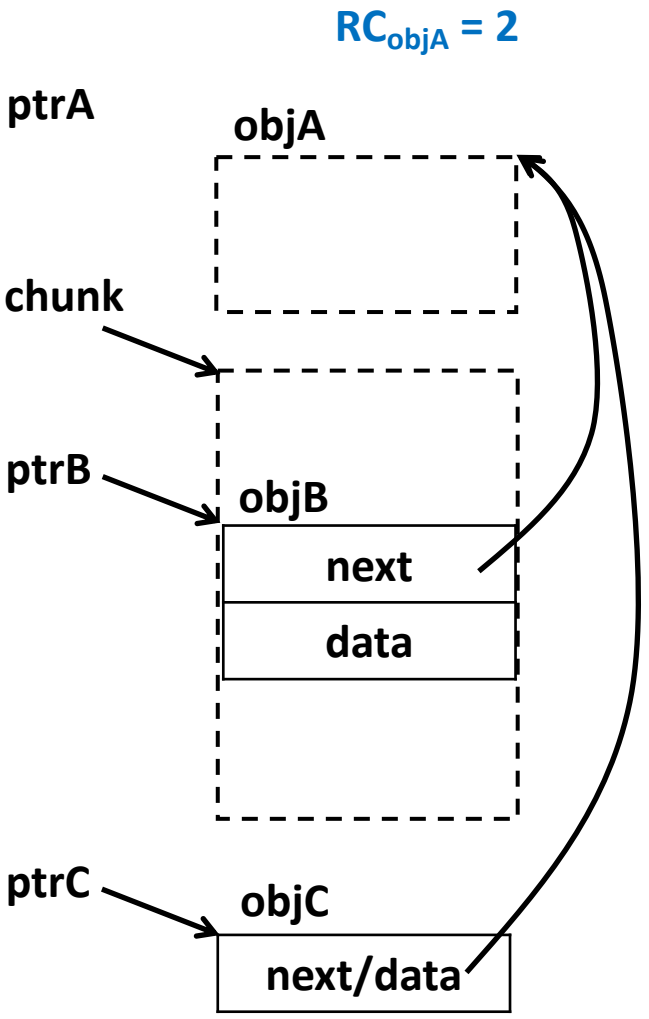
C Reference counting challenges




```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);
20 ptrC->data = 1;
```

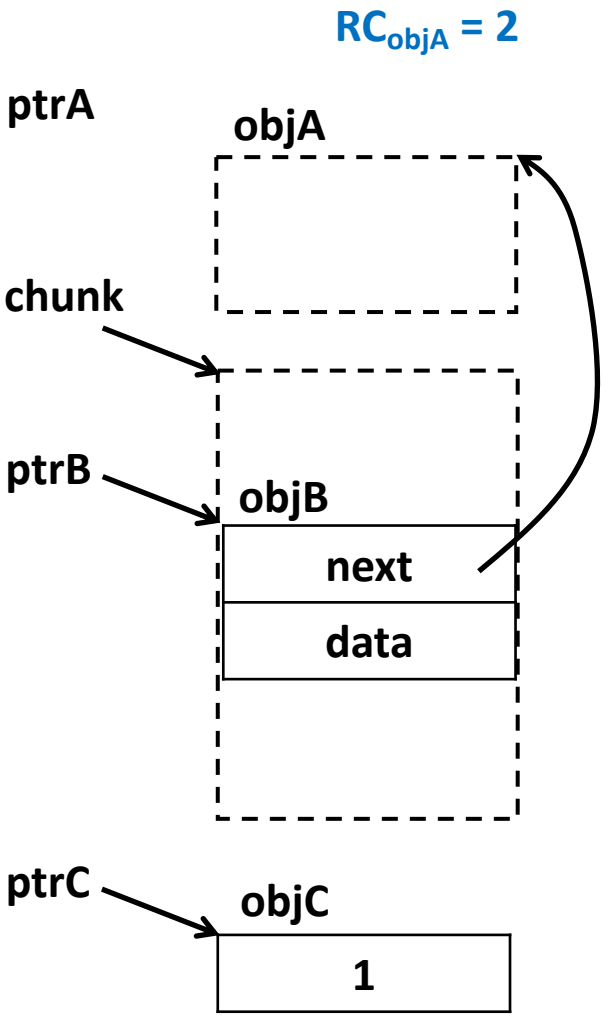
dec_rc_store

C Reference counting challenges




```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);   <- dec_rc_free
20 ptrC->data = 1;
```

C Reference counting challenges

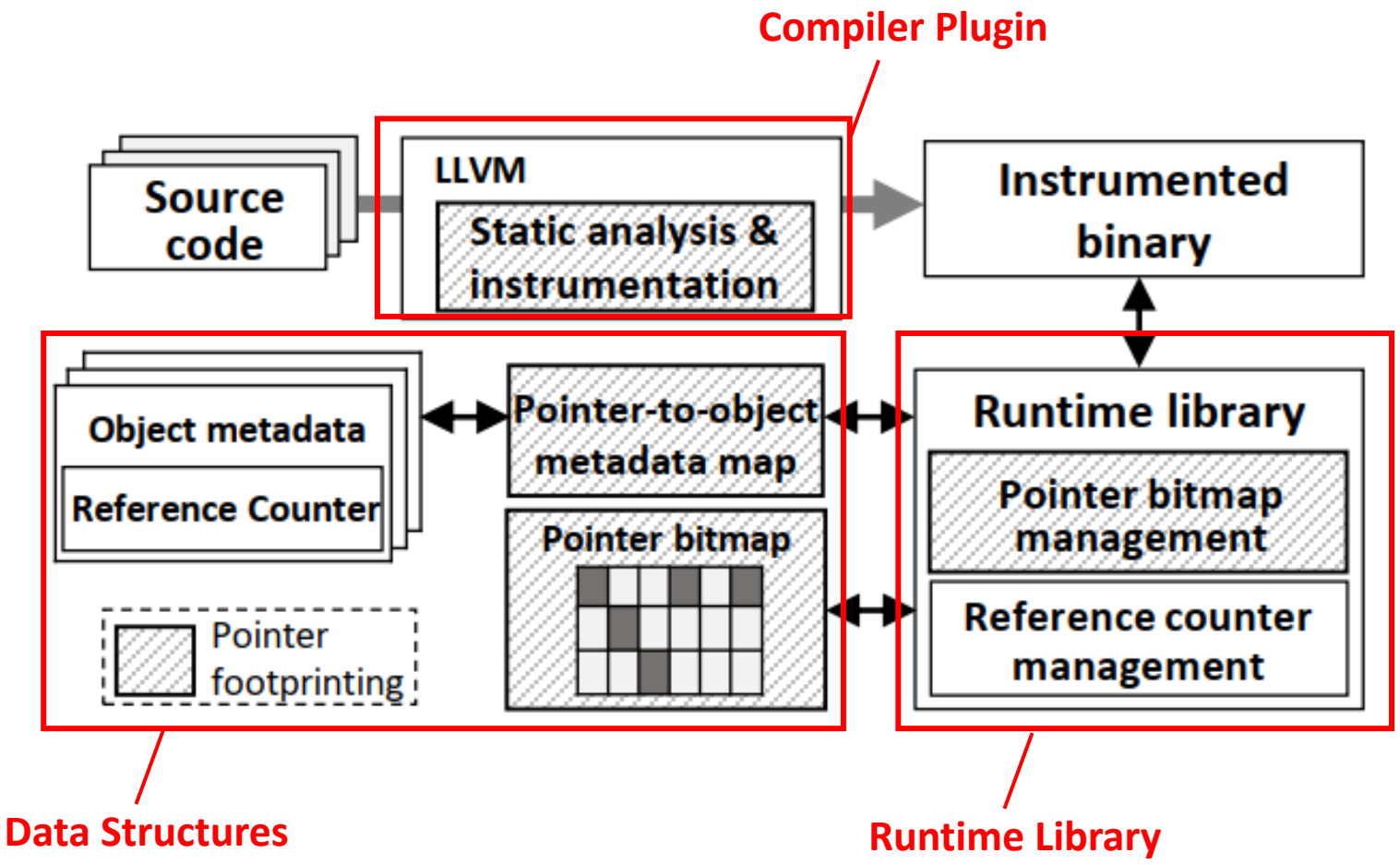



```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);
20 ptrC->data = 1;
```

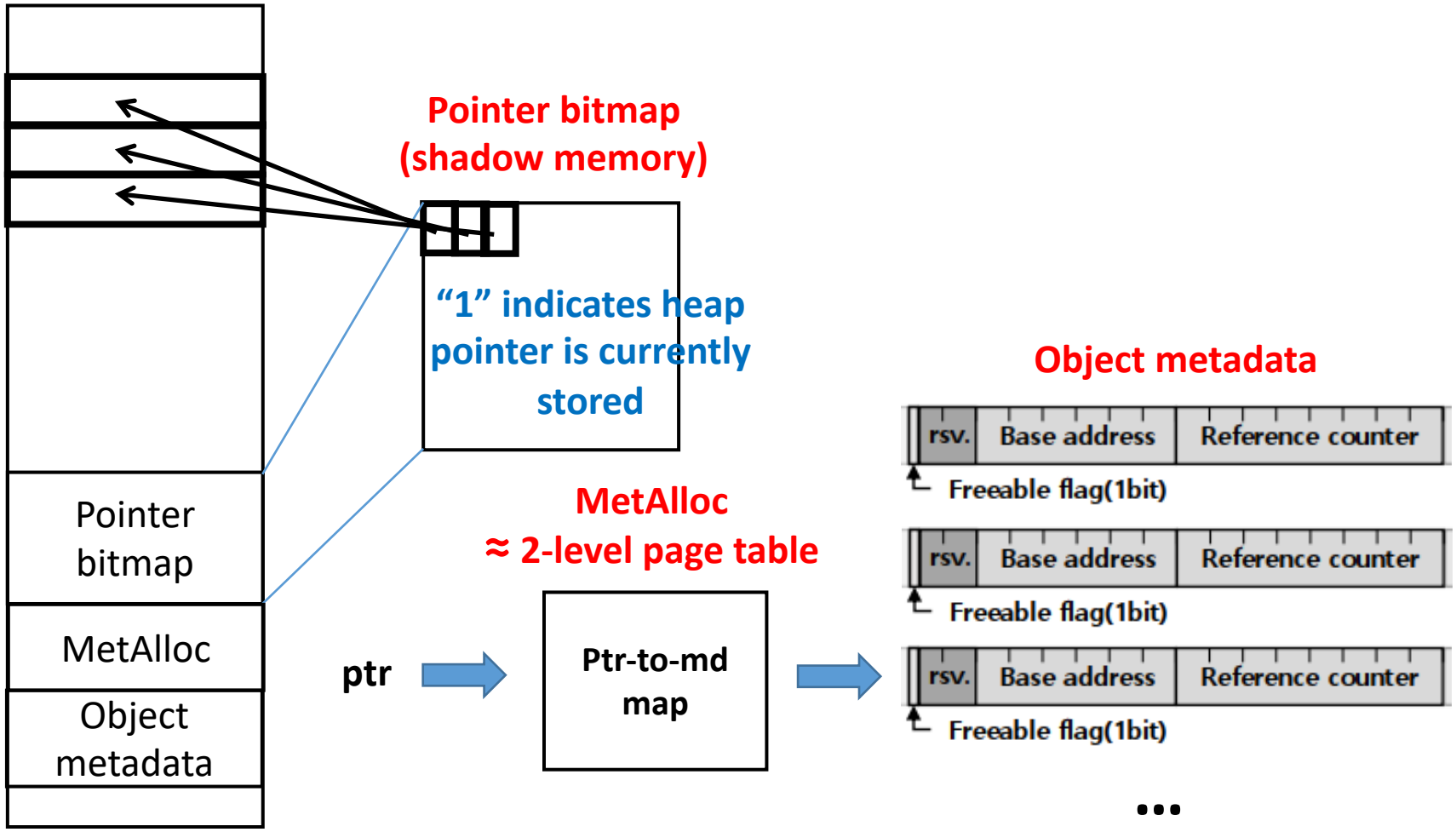
C Reference counting challenges

- How to know where the pointers are stored
- How to find right instrumentation points
  - Instrumenting only store instructions that have to do with the pointers

Overview of our approach



Data Structures



Metalloc: Efficient and comprehensive metadata management for software security Hardening, European Workshop on System Security, 2016

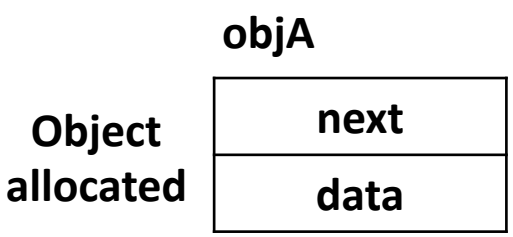
Runtime Library

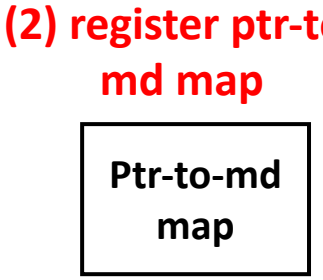
Runtime library function	Invoked at	Description
`crc_alloc`	Heap allocation	Add a mapping for the new heap object to the pointer-to-object metadata map
`crc_store`	Candidate store Instruction	Handle a pointer generation and/or kill due to memory store
`crc_memset`	Memset	Handle pointer kills due to memset'ing a region with identical bytes
`crc_memcpy`	Memcpy	Handle pointer generations and/or kills due to copying of a memory region
`crc_free`	Heap deallocation	Handle pointer kills by heap object deallocation
`crc_return`	Function return	Handle pointer kills by stack frame deallocation

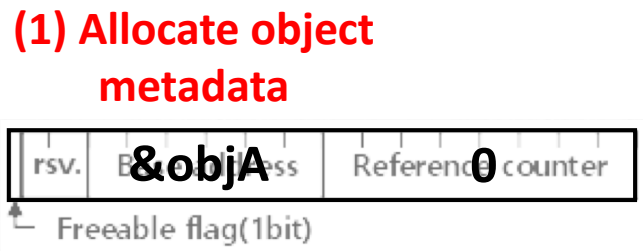
- crc_alloc, crc_free ➔ function hooks
- others ➔ Instrumented by the compiler

crc_alloc

```
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

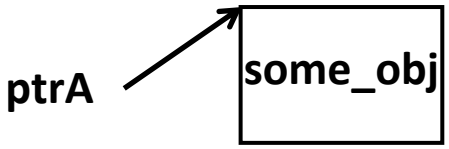




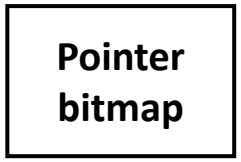


crc_store

```
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

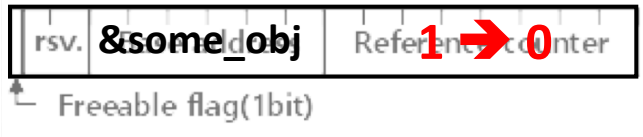


(1) Check if heap ptr was stored there



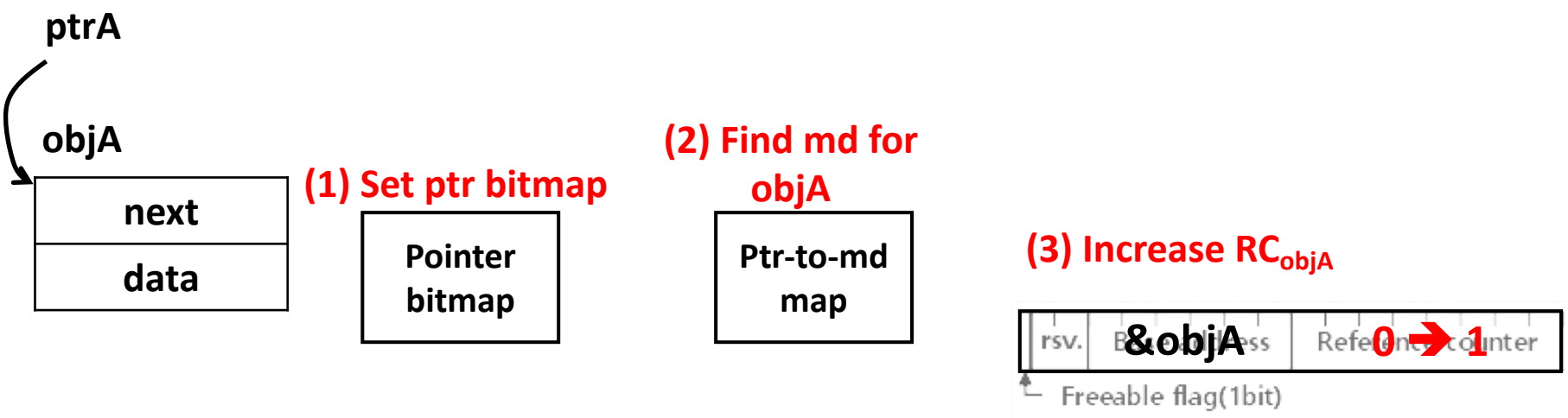
(2) If yes, find md for *some_obj*

(3) Decrease RC_{som_obj} (free *some_obj* if RC = 0)



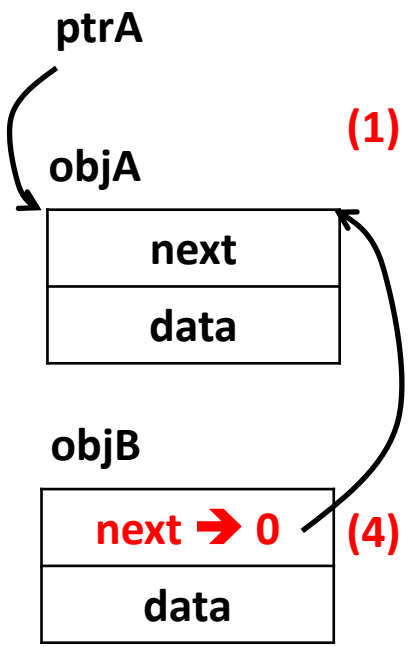
crc_store

```
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```



crc_free

```
4  ptrA = malloc(sizeof(struct node));   // objA
5  ptrB = malloc(sizeof(struct node));   // objB
6
7  ptrB->next = ptrA;
8
9  /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```



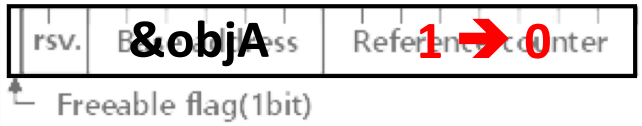
(1) Find ptrs in objB

Pointer bitmap

(2) Find md for those ptrs

Ptr-to-md map

(3) Decrease corresponding RC (free objA)



(4) Nullify the ptrs

(5) Free objB only if RC == 0

C Reference counting challenges

- ~~How to know where the pointers are stored~~

- How to find right instrumentation points
  - Instrumenting only store instructions that have to do with the pointers

Compiler plugin

- Selectively instrument memory stores

Runtime library function
crc_alloc
crc_store
crc_memset
crc_memcpy
crc_free
crc_return

```
1  for storeInst in program:
2    dest = storeInst.dest
3    val = storeInst.val
4
5    if !isPointerType(val) && !isCastFromPtr(val):
6      if !shouldInstrument(storeInst.dest):
7        continue
8
9    if isLoadStoreSame(dest, val):
10     continue
11
12   callInst = createCallInst(crc_store, dest, val)
13   storeInst.insertBefore(callInst)
```

Compiler plugin

```
1  for storeInst in program:
2    dest = storeInst.dest
3    val = storeInst.val
4
5    if !isPointerType(val) && !isCastFromPtr(val):
6      if !shouldInstrument(storeInst.dest):
7        continue
8
9    if isLoadStoreSame(dest, val):
10     continue
11
12   callInst = createCallInst(crc_store, dest, val)
13   storeInst.insertBefore(callInst)
```

LLVM IR: `store %struct.sv* some_ptr , %struct.sv** %25, align 8`

val type **val** **dest type** **dest**

Compiler plugin

```
1  for storeInst in program:
2    dest = storeInst.dest
3    val = storeInst.val
4
5    if !isPointerType(val) && !isCastFromPtr(val):
6      if !shouldInstrument(storeInst.dest):
7        continue
8
9    if isLoadStoreSame(dest, val):
10     continue
11
12   callInst = createCallInst(crc_store, dest, val)
13   storeInst.insertBefore(callInst)
```

C code: `AvARRAY(av)[--key] = some_ptr ;`

LLVM IR: `store %struct.sv* some_ptr , %struct.sv** %25, align 8`

val type **val** **dest type** **dest**

Compiler plugin

```
for storeInst in program:
  dest = storeInst.dest
  val = storeInst.val

  if !isPointerType(val) && !isCastFromPtr(val):
    if !shouldInstrument(storeInst.dest):
      continue

  if isLoadStoreSame(dest, val):
    continue

  callInst = createCallInst(crc_store, dest, val)
  storeInst.insertBefore(callInst)
```

Not ptr type?

C code:

```
Perl_repeatcpy(pTHX_ register char *to, register const char *from,
    *to++ = *from++;
```

Not ptr type?

LLVM IR:

```
store i8 %244, i8* %.212.i
```

val

Compiler plugin

```
for storeInst in program:
  dest = storeInst.dest
  val = storeInst.val

  if !isPointerType(val) && !isCastFromPtr(val):
    if !shouldInstrument(storeInst.dest):
      continue

  if isLoadStoreSame(dest, val):
    continue
```

Ptr type!

C code:

```
struct sv ** MARK
repeatcpy((char*)(MARK + items), (char*)MARK,
Perl_repeatcpy(pTHX_ register char *to, register const char *from,
*to++ = *from++;
```

Ptr type!

LLVM IR:

```
%228 = bitcast %struct.sv** %225 to i8*
%.1211.i = phi i8* [ %243, %.lr.ph13.i ], [ %228, %.lr.ph13.i.preheader ],
%244 = load i8, i8* %.1211.i,
store i8 %244, i8* %.212.i
```

val

C Reference counting challenges

- ~~How to know where the pointers are stored~~
- ~~How to find right instrumentation points~~
  - ~~Instrumenting only store instructions that have to do with the pointers~~

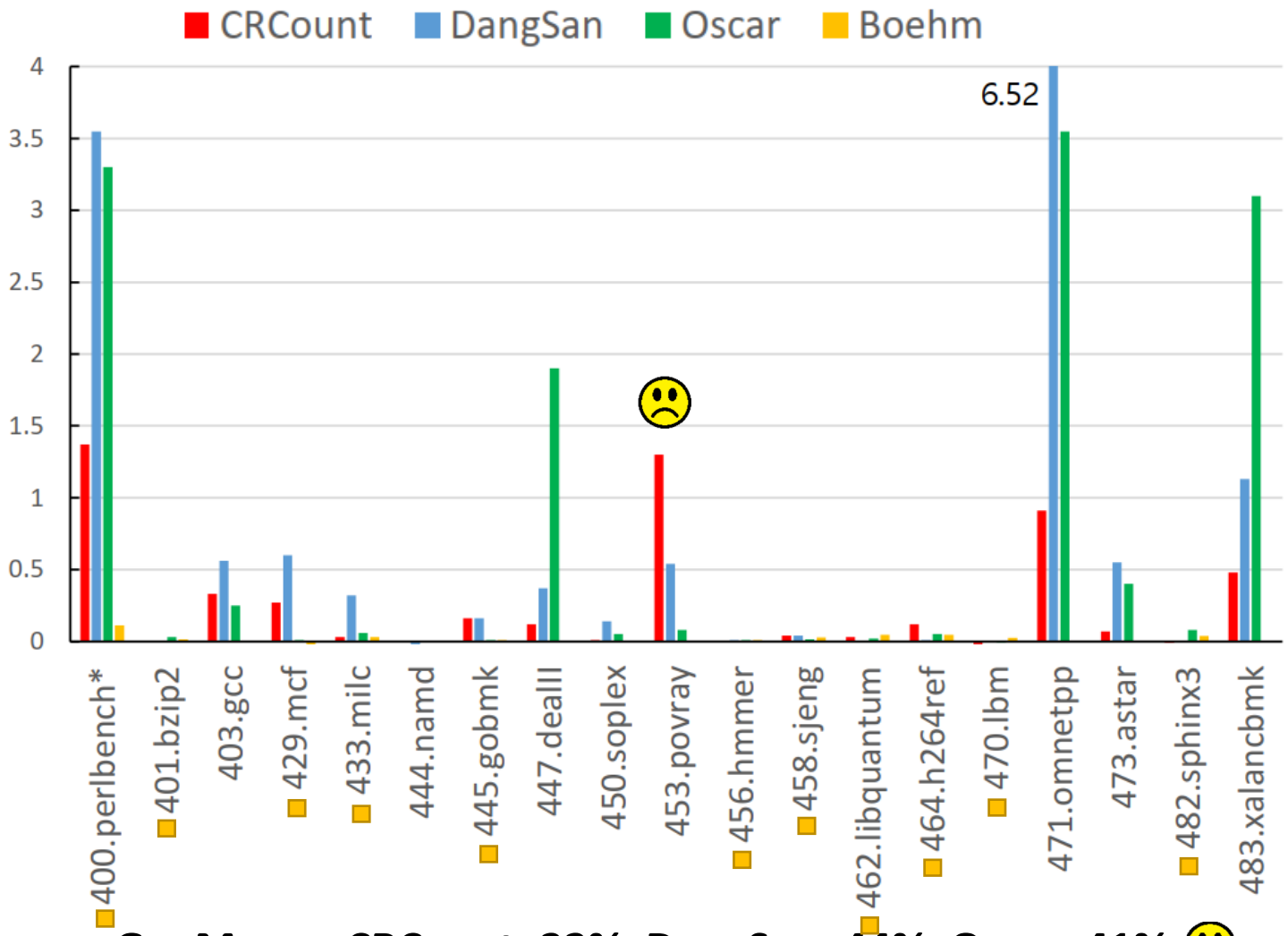
Performance Evaluation

- Intel Xeon, 10 cores @ 2.2GHz, 64GB Memory
- Compared
  - CRCount (This work)
  - DangSan (EuroSys`17)
  - Oscar (Security `17)
  - Boehm-Demers-Weiser Garbage Collector (latest version)
    - Only for some benchmarks
- Benchmarks
  - SPEC2006 (single-threaded)
  - PARSEC (multi-threaded)
  - Web servers

BDW Garbage Collector

- Actively maintained C garbage collector
- GC_malloc() + a few other APIs
- Automatically frees GC_malloc'ed object when no references to the object
- --enable-redirect-malloc
  - Redirect malloc() to GC_malloc()
- -DIGNORE_FREE
  - Ignore free()
- Worked for most of the benchmarks
  - Some C bench did not work, C++ bench need manual work

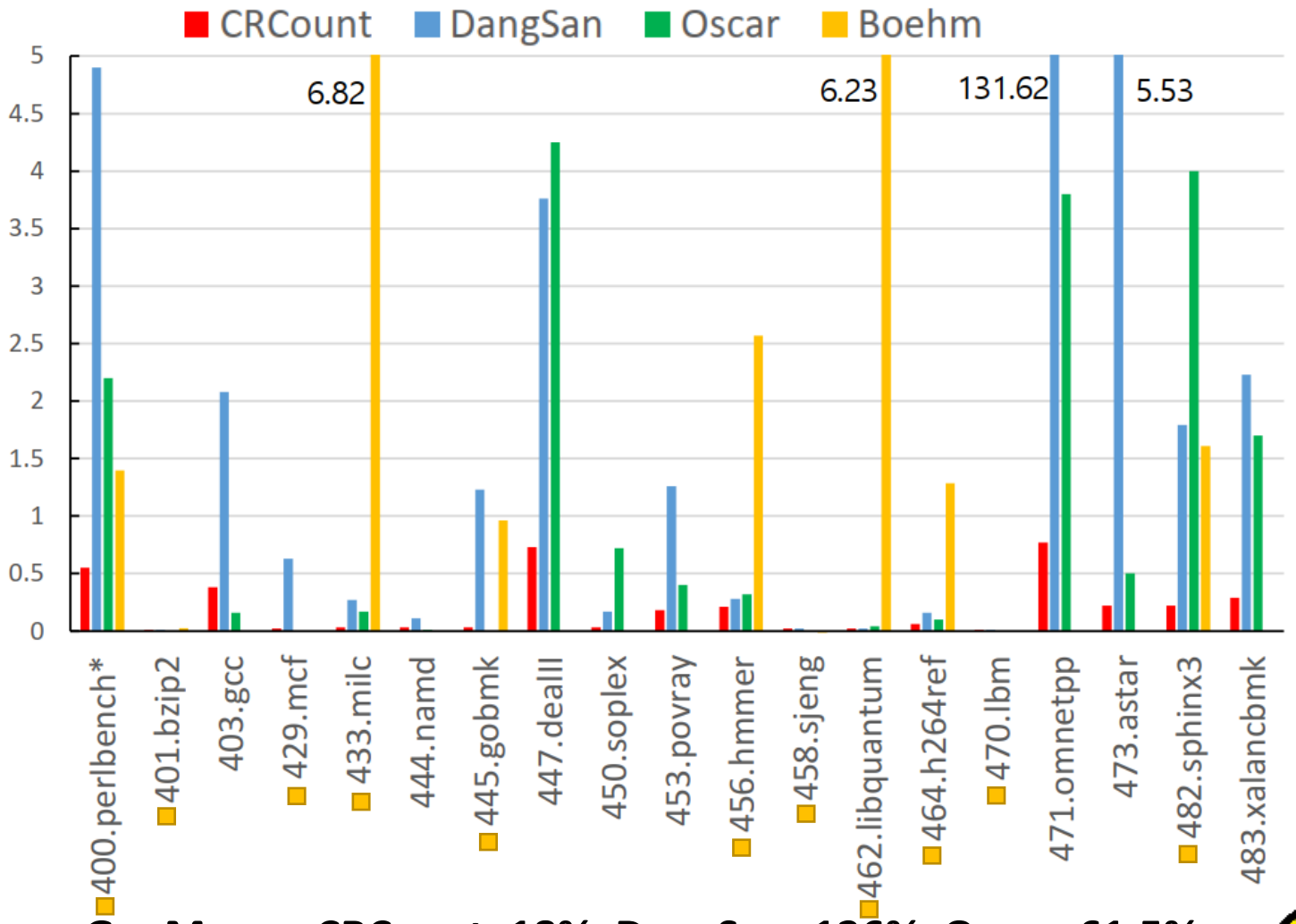
Runtime overhead (SPEC2006)



GeoMean - CRCount: 22%, DangSan: 44%, Oscar: 41% 🙂

GeoMean - CRCount: 13.9%, Boehm: 0.7% ☹

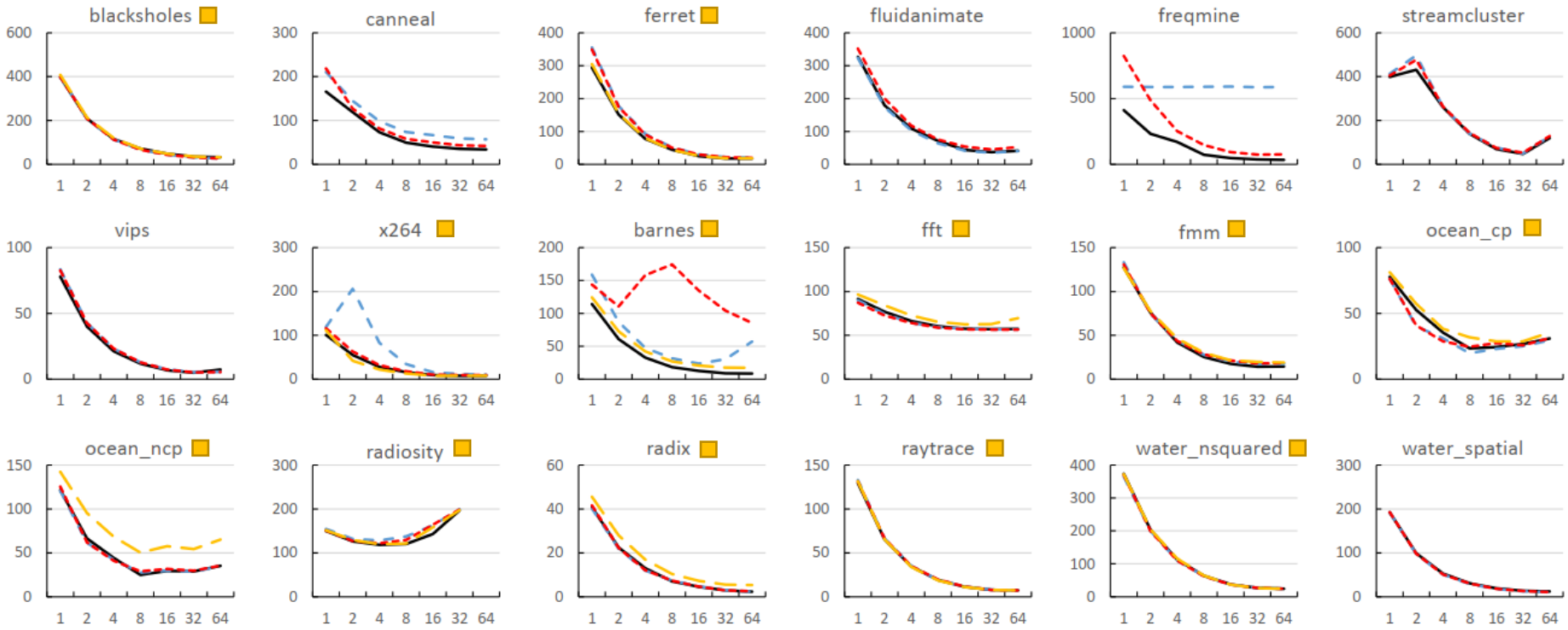
Memory overhead (SPEC2006)



GeoMean - CRCount: 18%, DangSan: 126%, Oscar: 61.5%

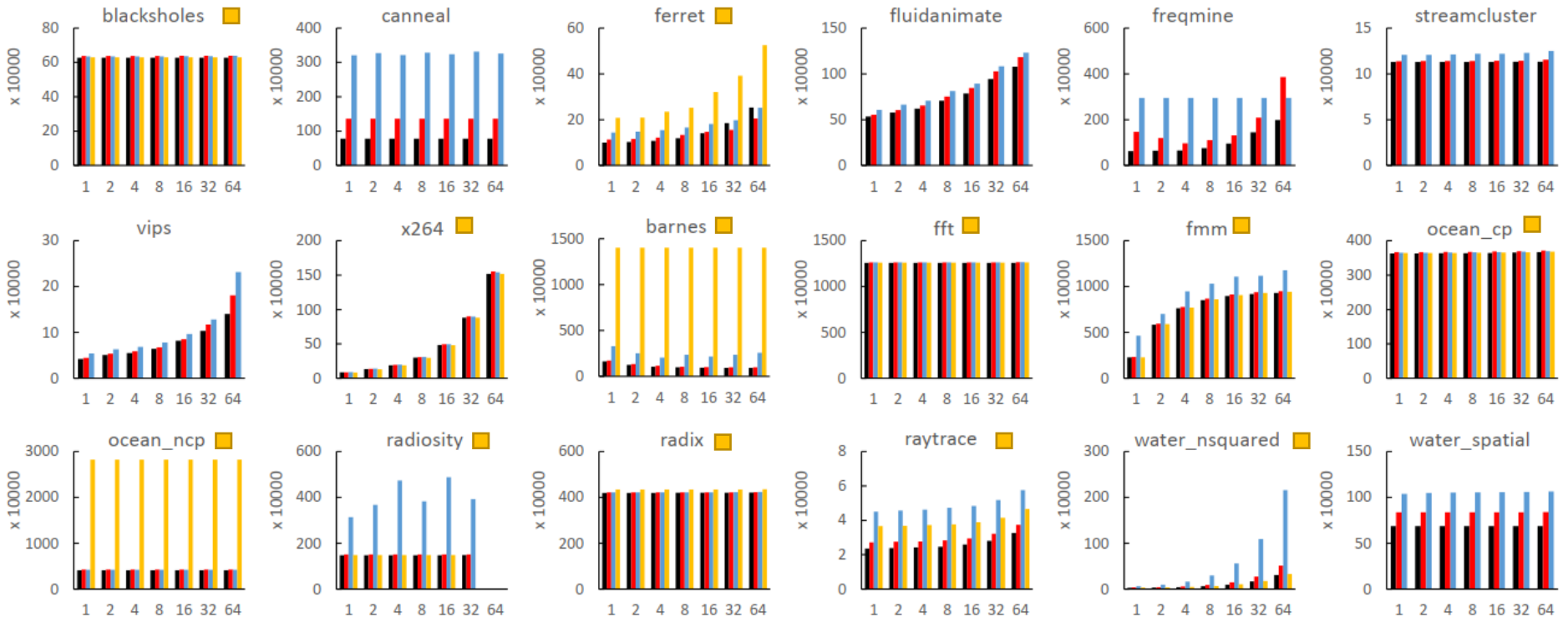
GeoMean - CRCount: 9.7%, Boehm: 126%

Runtime overhead (PARSEC)



GeoMean - CRCount: 6.1 ~ 22.4%, DangSan: 6.3 ~ 17.0%
GeoMean - CRCount: 4.9 ~ 28.6%, Boehm: 5.3 ~ 38.9%

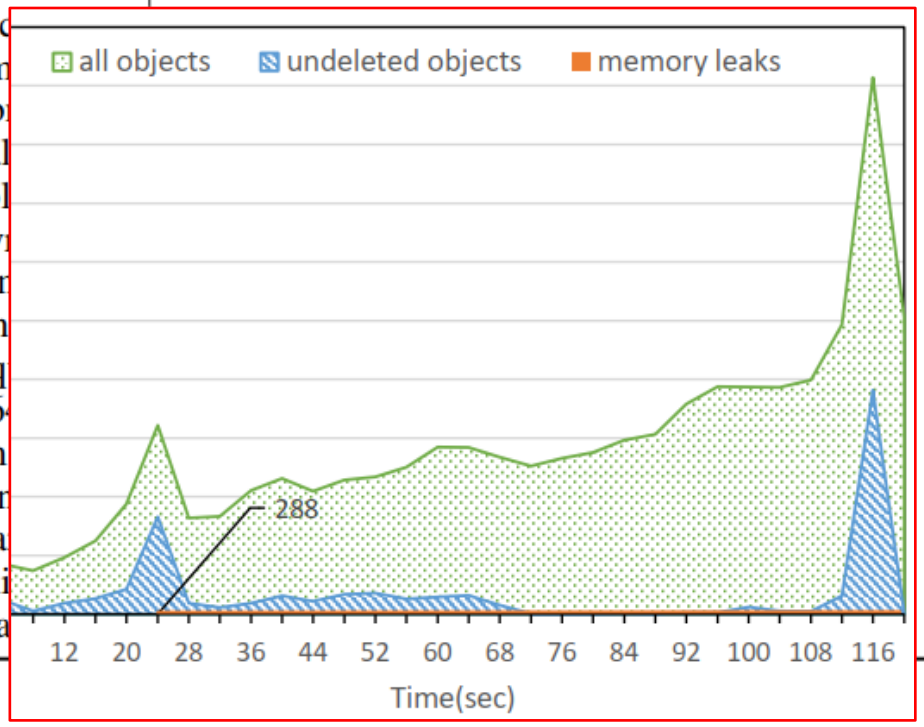
Memory overhead (PARSEC)



GeoMean - CRCount: 9.2 ~ 11.6%, DangSan: 45.0 ~ 52.7%
GeoMean - CRCount: 5.4 ~ 6.0%, Boehm: 56.6 ~ 70.9%

Quarantine zone size

benchmark	CRCount						
	# tot alloc.	# ptr stores by inst.	# ptr stores by memcpy	max mem.	max undeleted	max undel. / max mem.	leaks
400.perlbench	350m	44507m	242m	1103 MB	5838 KB	0.005	1680 B
401.bzip2	264	2200k	0	3362 MB	0	0	0
403.gcc	28m	9328m	13m	4075 MB	7491 MB	1.838	288 KB
429.mcf	21	10086m	574k	1676 MB	0	0	0
433.milc				679 MB	21 MB	0.032	0
444.nam				46 MB	0	0	0
445.gob				117 MB	34 KB	0	0
447.deal				791 MB	2048 KB	0.003	0
450.sopl				877 MB	27 MB	0.032	0
453.povr				2 MB	18 KB	0.007	0
456.hmr				41 MB	12 MB	0.291	0
458.sjen				172 MB	0	0	0
462.libq				96 MB	32 B	0	0
464.h26				111 MB	1609 KB	0.014	0
470.lbm				409 MB	0	0	0
471.omr				154 MB	1301 KB	0.008	481 KB
473.asta				471 MB	91 MB	0.195	0
482.sphi				44 MB	11 KB	0	0
483.xala				385 MB	1018 KB	0.003	576 B



Usually small

Memory leak

benchmark	max mem.	leaks
400.perlbench	1103 MB	1680 B
401.bzip2	3362 MB	0
403.gcc	4075 MB	288 KB
429.mcf	1676 MB	0
433.milc	679 MB	0
444.namd	46 MB	0
445.gobmk	117 MB	0
447.dealII	791 MB	0
450.soplex	877 MB	0
453.povray	2 MB	0
456.hmmer	41 MB	0
458.sjeng	172 MB	0
462.libquantum	96 MB	0
464.h264ref	111 MB	0
470.lbm	409 MB	0
471.omnetpp	154 MB	481 KB
473.astar	471 MB	0
482.sphinx3	44 MB	0
483.xalancbmk	385 MB	576 B

- Due to failure to track when pointers are killed
- Can be critical for long-running software
- ➔ Run light-weight GC when new objects in the quarantine reaches certain threshold
  - Use pointer-bitmap for pointer locations
- For 256MB, only 0.4% slowdown (for gcc)

Security eval

- CRCount only delays memory reuse
  - Attacks through the dangling pointer silently prevented
- Implemented CRCount-det
  - To detect dangling pointer dereference
- Memory reuse successfully delayed in all cases

Application	CVE	Vulnerability	Original	CRCount	CRCount-det
openlitespeed-1.3.7	2015-3890	UAF	No effect	No effect	Detected UAF
wireshark-2.0.1	2016-4077	UAF	No effect	No effect	Detected UAF
PHP-5.5.9	2016-3141	UAF	Crash (double free)	Detected double free	Detected UAF
PHP-5.5.9	2016-6290	UAF	No effect	Detected double free	Detected UAF
PHP-5.5.9	2016-5772	Double free	Crash (double free)	Detected double free	Detected double free
ed-1.14.1	2017-5357	Invalid free	Crash (invalid free)	Detected invalid free	Detected invalid free

Limitations

- Custom allocator
  - Should manually insert crc_free before custom free()
- Pointer alignment
  - cannot track pointers not aligned to 8B boundary ➔ rare
- Limitations in analysis

```
foo((int)ptr);

void foo(int a) {
   … = a;
}
```

```
struct bar {
  int x;
} bar1;

bar1.x = (int)ptr;
…
bar2.x = bar1.x;
```

Conclusion

- Hard to mitigate UAF efficiently
- Our approach - CRCount
  - Maintain light-weight data structures (Ref. cnts)
  - Minimize instrumentation points
- Efficient compared to existing work
  - 22% runtime, 18% memory overhead on SPEC2006

Thank you for listening!