Mitigating (Some) Use-after-frees in the Linux Kernel

Jann Horn, Google Project Zero

Agenda

- Preparation: Fancy RCU extension possibilities
- Motivation
- Design of a use-after-free mitigation prototype
- Pitfalls and limitations
- Aspirational ideas for long-term development
- Performance numbers

Fancy RCU extension possibilities

(not actually implemented, just as stepping stone)

(no, I'm not saying that you should actually do this)

Unconditional RCU-ref => counted-ref

- RCU limitation: Can't block inside read-side critical section

Classic options:

- retry loop around rcu_dereference() + refcount_inc_not_zero()
- optimistic GFP_NOWAIT

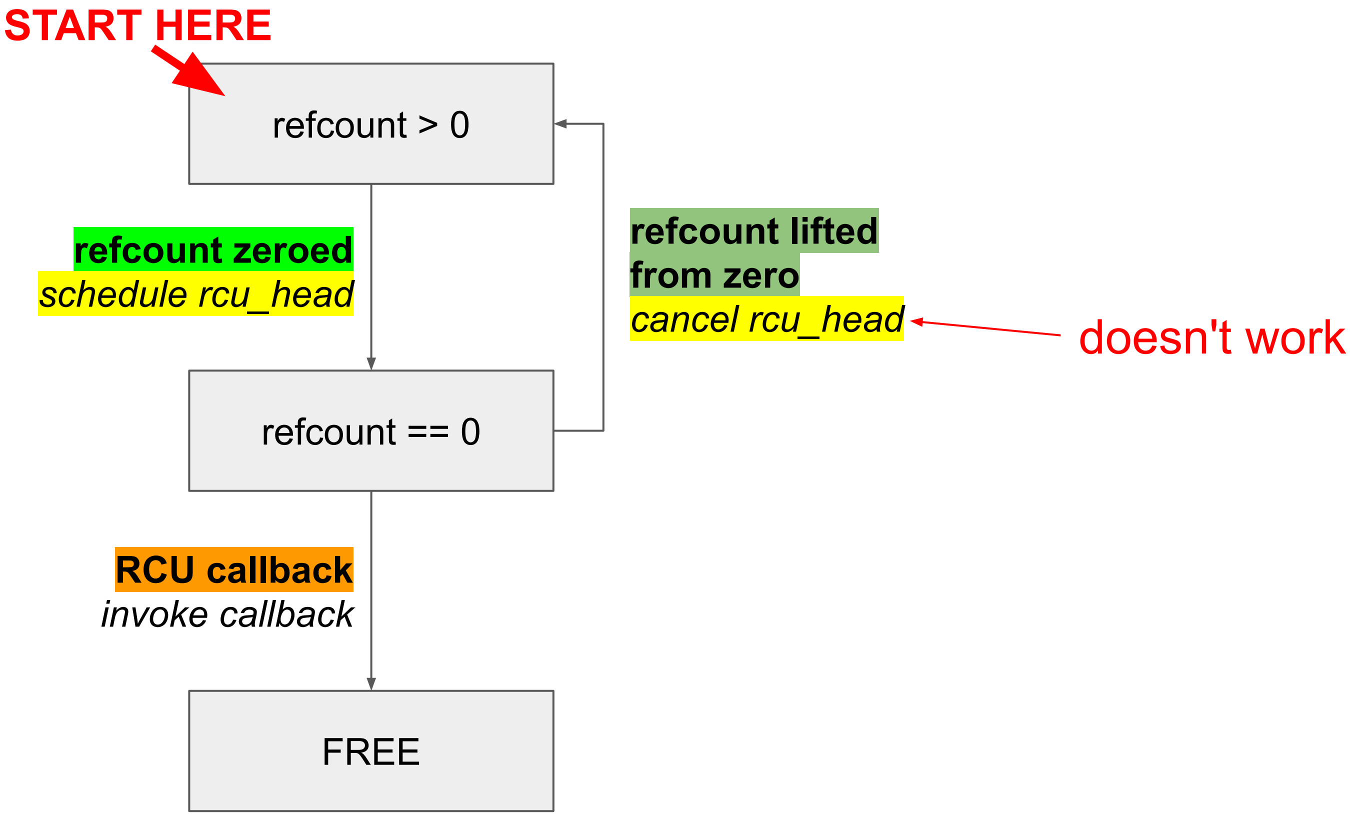
```
rcu_read_lock();
foo = rcu_dereference(ptr->foo);
...
if (...) {
  ... = kmalloc(...,GFP_KERNEL);
  ...
}
...
rcu_read_unlock();
```

Unconditional RCU-ref => counted-ref

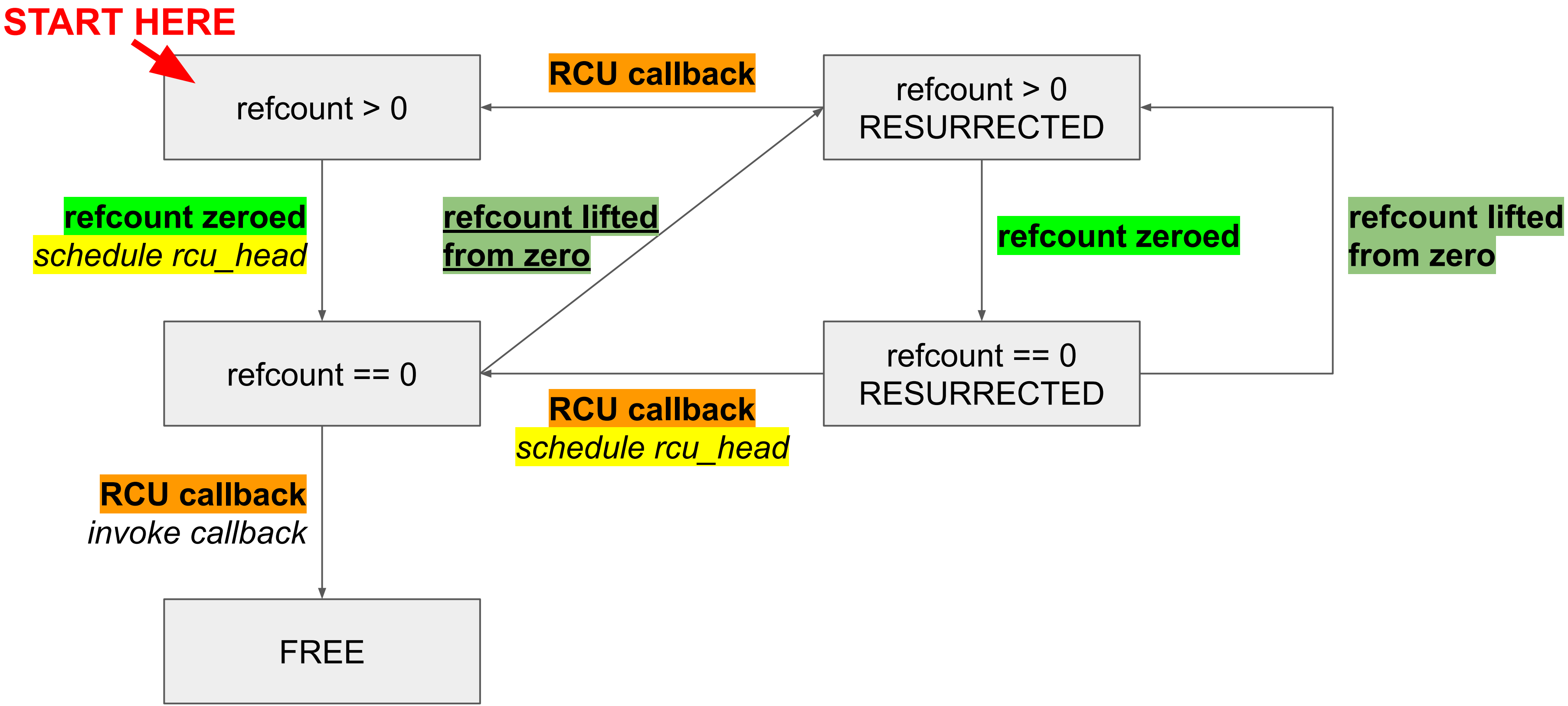
- Idea: Permit refcount increment through RCU reference
- **foo must only be freed after `foo->refs` has been zero for an entire RCU grace period**
- can be built on top of rcu_head API

```
rcu_read_lock();
foo = rcu_dereference(ptr->foo);
...
if (...) {
  ref_inc(&foo->refs);
  rcu_read_unlock();
  ... = kmalloc(...);
  rcu_read_lock();
  ref_dec(&foo->refs);
  ...
}
...
rcu_read_unlock();
```

Resurrectable refcount wrapper around rcu_head



Resurrectable refcount wrapper around rcu_head



sched-out mode switch

cache line contention

rarely actually blocks

```
rcu_read_lock();
foo = rcu_dereference(ptr->foo);
...
if (...) {
  ref_inc(&foo->refs);
  rcu_read_unlock();
  ... = kmalloc(...);
  rcu_read_lock();
  ref_dec(&foo->refs);
  ...
}
...
rcu_read_unlock();
```

- elide the refcounting unless we actually block?
    - without extra path for `GFP_NOWAIT` fail?

sched-out mode switch

- idea: preempt notifier
- `rcu_pin()` registers `rcu_ref` on task/pcpu
- on first sched-out:
  - set BLOCKED flag on pin
  - `ref_inc()`
  - `rcu_read_unlock()`
  - unregister from task
- on `rcu_unpin()` with BLOCKED:
  - `rcu_read_lock()`
  - `ref_dec()`

- Requires RCU core modifications
- Requires extra check in context switch

```
rcu_read_lock();
foo = rcu_dereference(ptr->foo);
...
if (...) {
  struct rcu_pin pin;
  rcu_pin(&pin, &foo->refs);
  rcu_permit_preempt();
  ... = kmalloc(...);
  rcu_deny_preempt();
  rcu_unpin(&pin, &foo->refs);
  ...
}
...
rcu_read_unlock();
```

Motivation and Mitigation Design

Scope of security bugs

Local impact ("logic bugs"):

- broken bind/rename handling in VFS path traversal code
- broken `PTRACE_TRACEME` security check

=> immediate impact mostly related to subsystem functionality

Global impact (e.g. memory corruption):

- shared futex slowpath pinned inode with `iget()`
- missing locking between coredumping and userfaultfd

=> impact independent of subsystem functionality

Performance issues vs. security issues

Performance issues:

- issues are noticeable
- profiling can (mostly) pinpoint issues
- small fixes can have large positive impact

Security issues:

- issues are (mostly) invisible
- issues can be almost anywhere

=> Turning security issues into **fixable** performance issues might be helpful

Pattern for a simple kernel UAF-write exploit

Scenario: can write arbitrary value into A->member after A was freed

- trigger allocation of A
- trigger freeing of A
- trigger allocation and initialization of B at A's old address
    - choose B such that A->member overlaps with B->function_pointer
- choose pointer P to a gadget in kernel code
- write P through A->member (corrupting B->function_pointer)
- trigger call to B->function_pointer

Pattern for a simple kernel UAF-write exploit

- trigger allocation of A
  - *mitigations: Seccomp, SELinux, ... [attack surface reduction]*
- trigger freeing of A
- trigger allocation and initialization of B at A's old address
  - *mitigation: memory tagging [on future ARM64]*
  - choose B such that A->member overlaps with B->function_pointer
    - *mitigation: struct randomization*
- choose pointer P to a gadget in kernel code
  - *mitigation: KASLR*
- write P through A->member
- trigger call to B->function_pointer
  - *mitigation: CFI*

Pattern for a simple kernel UAF-write exploit

- trigger allocation of A
    - *mitigations: Seccomp, SELinux, ... [attack surface reduction]*
- trigger freeing of A
- trigger allocation and initialization of B at A's old address
    - *mitigation: memory tagging [on future ARM64]*
    - choose B such that A->member overlaps with ~~B->function_pointer~~ **B->buffer_pointer**
        - *mitigation: struct randomization*
- choose pointer P to ~~a gadget in kernel code~~ **important data**
    - *mitigation: KASLR*
- write P through A->member
- ~~trigger call to B->function_pointer~~
    - ~~*mitigation: CFI*~~
- **trigger reads/writes through B->buffer_pointer**

Pattern for a simple kernel UAF-write exploit

- trigger allocation of A
  - *mitigations: Seccomp, SELinux, ... [attack surface reduction]*
- trigger freeing of A
- trigger allocation and initialization of B at A's old address
  - *mitigation: memory tagging [on future ARM64]*
  - choose B such that A->member overlaps with B->buffer_pointer
    - *mitigation: struct randomization*
- choose pointer P to important data
  - *mitigation: KASLR*
- write P through A->member
- trigger reads/writes through B->buffer_pointer

everything except attack surface reduction above is probabilistic

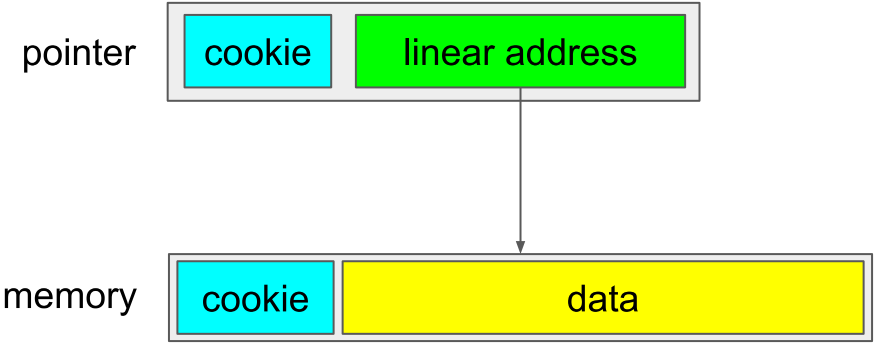
Design goal: As close to the actual bug as possible

- Actual bugs: Reference counting, locking, ...
  - Ideally mitigate here
  - Extremely hard or infeasible to reliably detect (in normal C code)
- Immediate symptom: Memory access through dangling pointer to reused memory
  - **ASAN:** detects free memory access; software; for debugging
  - **HWASAN:** probabilistically detects UAF; software
  - **Memory Tagging (MT):** probabilistically detects UAF; hardware
- Design goal: Deterministic protection in software against use-after-reallocation
- Target environment: Desktop X86-64 system

(ASAN/HWASAN/MT also address OOB bugs, I don't)

Basic design: Fat pointers (HWASAN / MT)

- embedded cookie disambiguates address reuse
- memory access is associated with cookie check
- difference: HWASAN / MT use cookie for probabilistic protection (except for non-UAF goals)



Design Goal: No pointer size change

- For lockless pointer updates
- Avoid metadata inconsistency via data races
- Avoid per-pointer memory usage

(like HWASAN / Memory Tagging)

=> Fat pointer must fit into 64 bits

Design goal: Mergeable object-level checks

```
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr) {
  int res;

  res = ptr->a;
  for (int i=0; i<ptr->b; i++) {
    other_function(ptr);
    res += ptr->c[i];
  }
  return res;
}
```

Design goal: Mergeable object-level checks

```
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr) {
  int res;

  res = CHECKED_LOAD(&ptr->a);
  for (int i=0; i<CHECKED_LOAD(&ptr->b); i++) {
    other_function(ptr);
    res += CHECKED_LOAD(&ptr->c[i]);
  }
  return res;
}
```

Design goal: Mergeable object-level checks

```
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr) {
  int res;
  struct bar *ptr_decoded = START_ACCESS(ptr);

  res = ptr_decoded->a;
  for (int i=0; i<ptr_decoded->b; i++) {
    other_function(ptr);
    res += ptr_decoded->c[i];
  }
  return res;
}
```

Design goal: Mergeable object-level checks

```
struct pin { struct pin *next; void *ptr; };
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr) {
  int res;
  struct pin pin = { .next = current->pins, .ptr = ptr };
  WRITE_ONCE(current->pins, &pin);
  struct bar *ptr_decoded = START_ACCESS(ptr);

  res = ptr_decoded->a;
  for (int i=0; i<ptr_decoded->b; i++) {
    other_function(ptr);
    res += ptr_decoded->c[i];
  }
  WRITE_ONCE(current->pins, pin.next);
  return res;
}
```

refcounted on sched-out

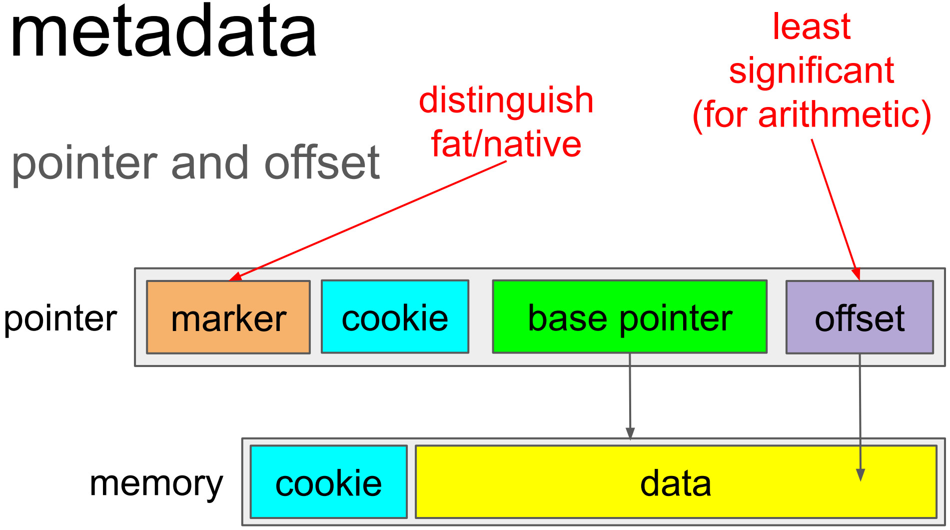
Design goal: Mergeable object-level checks

- Optimization: One list element per function frame, with pin array
- Optimization: percpu variable instead of `current->pins`
  - switched on task switch (like stack protector)
- Alternative (discarded): ORC unwinding instead of linked list
  - Problems anytime unwinding is unreliable
  - More complex
  - ORC unwinding under the runqueue lock 😱

- Want per-object metadata

Fat pointers for per-object metadata

- fat pointer must store separate base pointer and offset

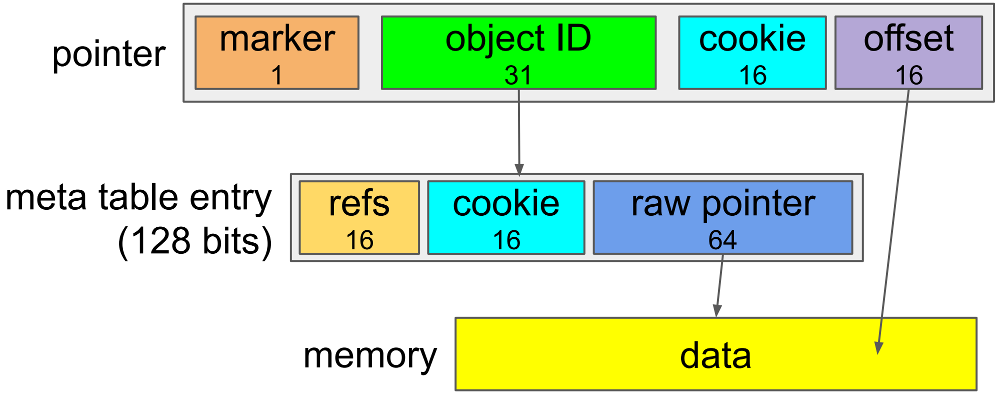


Problems:

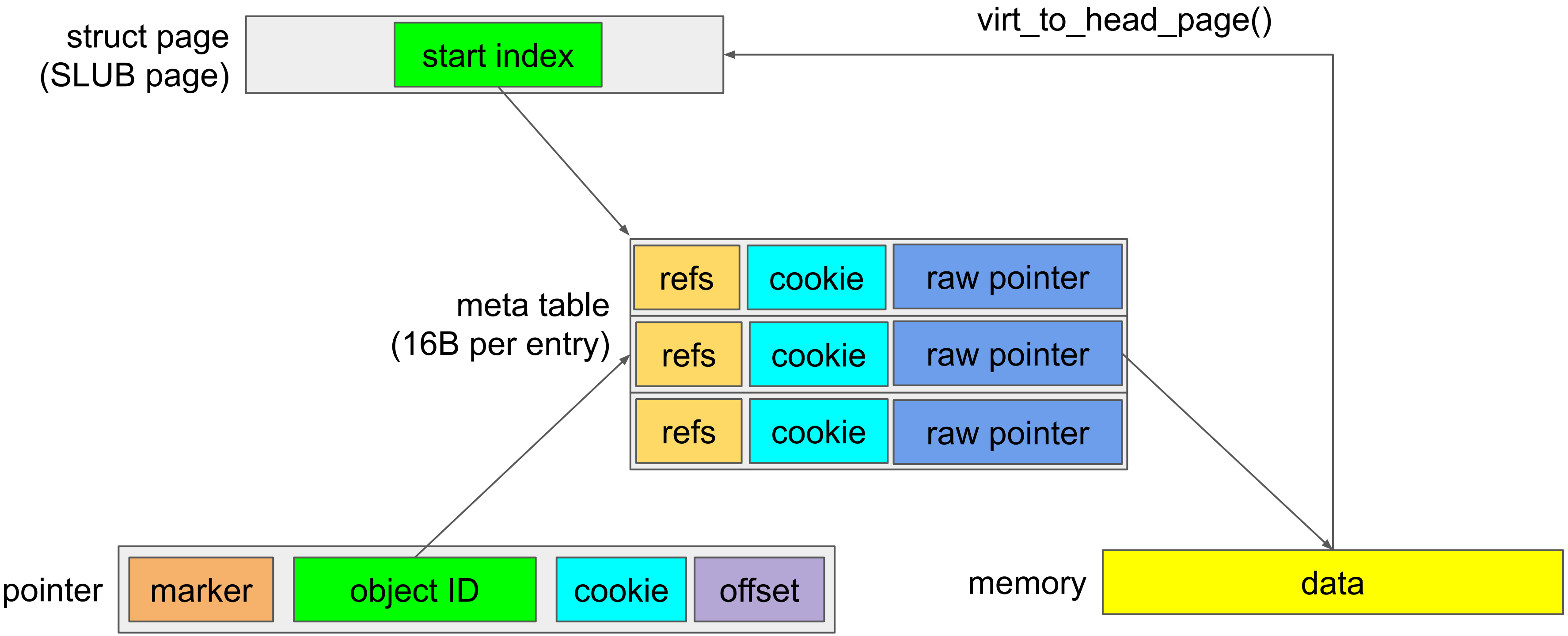
- pointer bits are limited; example:
  - marker: 1 bit
  - cookie: 15 bits
  - offset: 16 bits
  - base pointer (relative to base): $\log_2$(64GiB / 16 bytes) = 32 bits
- virtual memory repartitioning (without shadow mapping)
  - (okay for probabilistic detection)
  - can't use physical mapping + SLUB page freeing
- data alignment
- cookie depletion

Fat pointers for per-object metadata

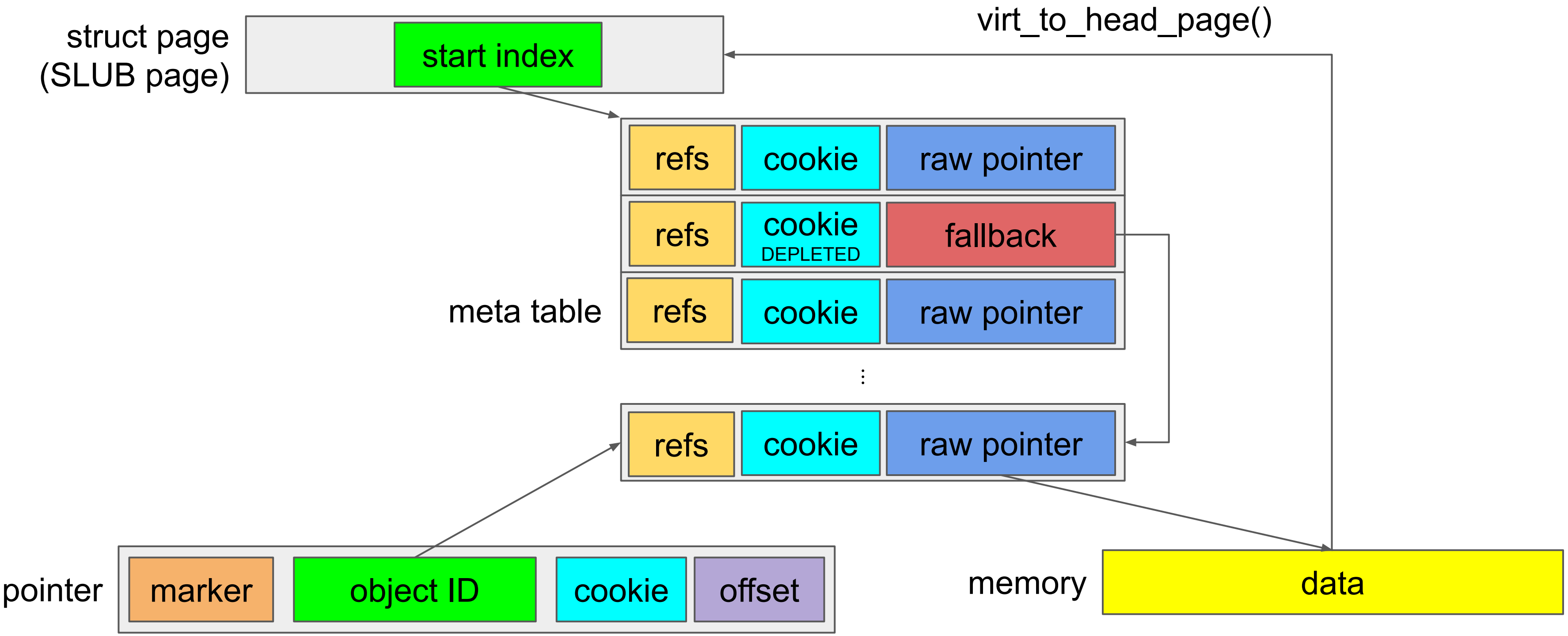
- advantage: much denser identifier space
- advantage: memory repartitioning is much easier
- advantage: when cookies run out, can use a "fallback" entry
- disadvantage: memory indirection



Mapping between SLUB objects and meta structs



Depleted allocations, fallback identifiers



Depleted allocations, fallback identifiers

- Split metadata ID space into 2^{30} normal entries, 2^{30} fallback entries
- Normal entries:
  - Enough for ~8GiB of kmalloc-8 allocations or ~440 GiB of buffer_head allocations
- Fallback entries:
  - 2^{16} alloc+free cycles per fallback entry reservation
  - $2^{16} * 2^{30} = 2^{46}$ alloc calls before exhaustion
    - Pessimistic example, if allocating once every 100 cycles on one 2GHz CPU: 2^{46} / 20Mhz ≈ 40 days
  - Memory leakage: 16B * $2^{-16} = 2^{-12}$B per alloc call
    - Pessimistic example, if allocating once every 100 cycles on one 2GHz CPU for a day: 20Mhz * 1day * 2^{-12}B ≈ 402 MiB
      - [can be optimized, see bonus slides section]

Delayed freeing

- Delay freeing until no more references can exist
- Kinda like NO_HZ_FULL RCU
- Refcounts count references from non-running tasks
- Unreferenced free objects land on percpu queue (state QUEUED)
- When nothing on stack (exit to userspace or switch to idle):
  - process percpu queue (unreferenced elements move onto global queue)
  - kick off sync with running CPUs if global queue is getting too big
  - if sync with all running CPUs is done, process global queue

Optimization: Local freeing

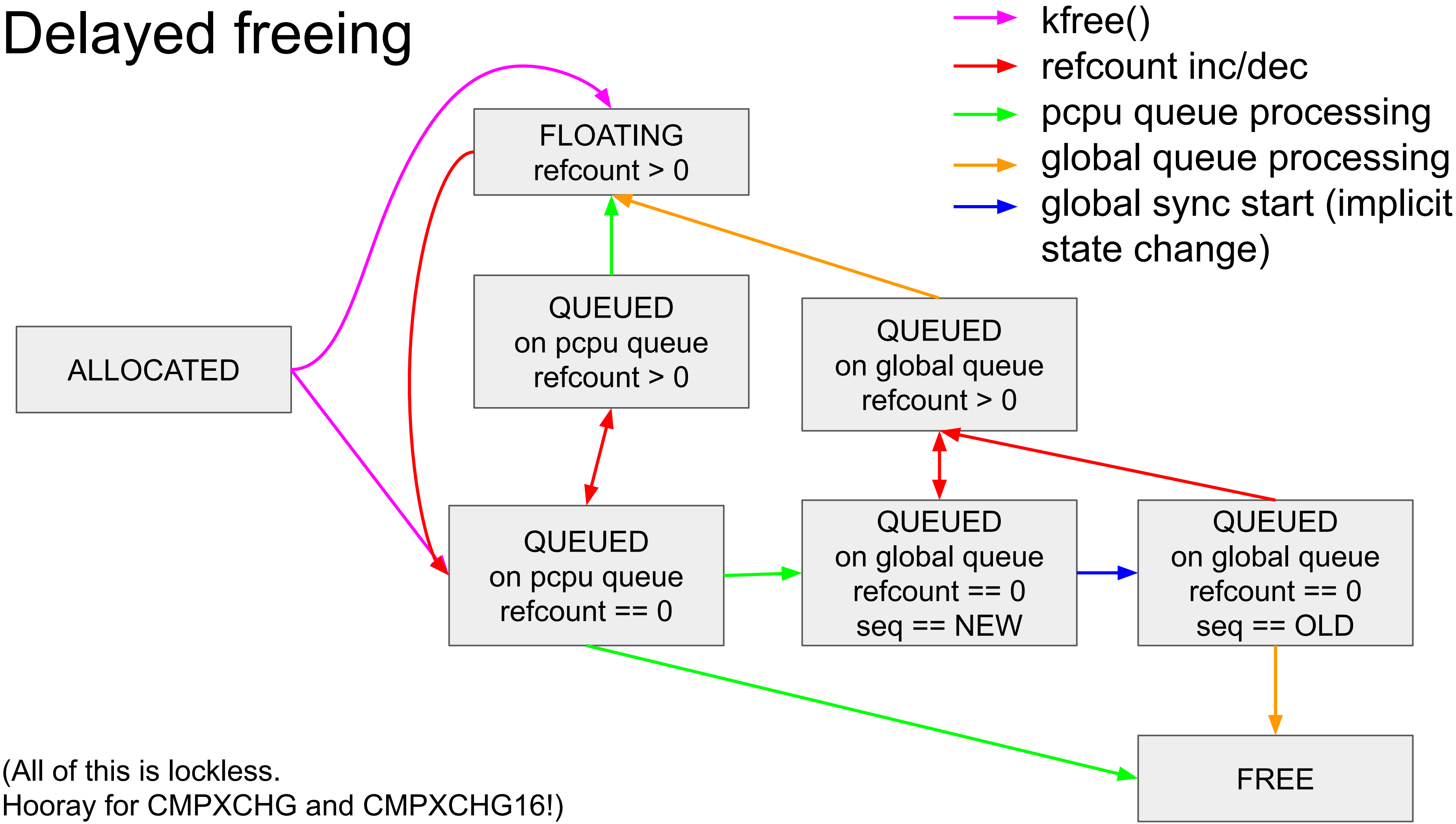
- On alloc: Store CPU number in metadata
- On access: Wipe CPU number on mismatch with current
- On free: Skip global queue on match

On-access pseudocode:

```
u8 me = get_current_cpu_num();
u8 stored = READ_ONCE(meta->cpu_num);
if (stored != GLOBAL && stored != me)
  WRITE_ONCE(meta->cpu_num, GLOBAL);
```

can be optimized, see bonus slides at the end

Delayed freeing



(All of this is lockless.
Hooray for CMPXCHG and CMPXCHG16!)

Design goal: Speculatable checks

```
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr, int count) {
  int res = 0;
        check here?
  for (int i=0; i<count; i++) {
    other_function(ptr);
    res += ptr->c[i];        check here?
  }
  return res;
}

foo(bogus_pointer, 0)
```

(pins-related code omitted for simplicity)

Design goal: Speculatable checks

```
struct bar { int a; int b; int c[100]; }
int foo(struct bar *ptr, int count) {
  int res = 0;
  struct bar *ptr_decoded = START_ACCESS(ptr);

  for (int i=0; i<count; i++) {
    other_function(ptr);
    res += ptr_decoded->c[i];
  }
  return res;
}
```

returns non-canonical pointer

#GP on access

- approach copied from ARMv8.3 Pointer Authentication
- breaks only if pointer can become valid after load - we have no pointer reuse

Current coverage limitations

- Currently not watching in idle task (including its interrupts)
- Disabled for task_struct
- Disabled for all constructor/RCU slabs
  - Should add a slower implementation of these (also for ASAN / Memory Tagging / ...)
- Nothing except SLUB. None of:
  - on-stack allocations
  - struct page (and associated pages in linear mapping)
  - vmalloc
  - ...

Other limitations

- no infrastructure for references from hardware
  - e.g. references from IOMMU
- use-after-destruction of covered object can still be exploitable as UAF of indirectly reachable non-covered object

Handwavy future plans: Elision

- Allow programmer to prove locking correctness => elide protection
- Make specific locks statically provable (balancing, member protection)
- Rarely-written pointers:
  - require lock annotation
  - mark via attribute
  - split into decoded and raw pointer
  - refcounted raw pointer usable directly, without decoding

Performance numbers

Memory overhead example

- 8GB RAM machine
- Memory mostly filled with filesystem cache
- Overhead relative to SLUB objects: ~4.4%
- Overhead relative to MemTotal: ~0.23%
  - (this number is kinda cheating)

```
orig meta memory: 17264 kB (not counting page tables)
fallback meta memory: 4 kB

total objects: 1285543  (0.120% of 2^30)
total SLAB memory use: 398323784 B  (~380 MiB)
top slabs by object count:
  anon_vma_chain            24000 objects =      1.46 MiB
  inode_cache               30828 objects =     16.70 MiB
  vm_area_struct            33900 objects =      6.47 MiB
  proc_inode_cache          57425 objects =     35.05 MiB
  kernfs_node_cache         67840 objects =      8.28 MiB
  radix_tree_node           67900 objects =     37.82 MiB
  ext4_extent_status       143310 objects =      5.47 MiB
  ext4_inode_cache         148924 objects =    148.84 MiB
  buffer_head              260247 objects =     25.81 MiB
  dentry                   266952 objects =     48.88 MiB
```

CPU overhead (with a truly awful benchmark)

- benchmark: building the kernel
  - tinyconfig; `make -j4 -s`; with hot VFS caches
  - (This is a terrible benchmark! Almost all time is spent in userspace, which is unaffected by the instrumentation.)
- baseline:
  - 58.50s; 58.40s; 58.09s
- instrumented, but not enabled for any slabs:
  - 61.63s; 61.62s; 61.93s
  - ~6% overhead relative to baseline
- with mitigation:
  - 62.92s; 63.03s; 63.05s
  - ~8% overhead relative to baseline

CPU overhead (low-IPC, parallel, not many allocations)

- benchmark: `git status` (with hot VFS caches)
- baseline:
  - 172ms, 173ms, 176ms
- compiler instrumentation only, no infrastructure, helpers stubbed out:
  - 186ms, 183ms, 187ms
  - ~8% overhead relative to baseline
- instrumented+infrastructure, but not enabled for any slabs:
  - 242ms, 237ms, 220ms
  - ~37% overhead relative to baseline
- with mitigation:
  - 276ms, 284ms, 277ms
  - **~60% overhead relative to baseline**

CPU overhead (producer-consumer pattern)

- benchmark: unix domain socket, 1M single-byte messages, one task sends, one task receives, pinned to fixed (different) CPUs
  - exercise global freeing path
  - terrible cache locality
- baseline:
  - 509ms, 495ms, 501ms
- with mitigation:
  - 1293ms, 1297ms, 1314ms
  - **~159% overhead**

Conclusions

- Memory overhead is not a huge problem
- CPU overhead for kernel-heavy tasks is pretty bad (roughly 60% - 160% in my tests)
- Lowering CPU overhead to something reasonable likely requires more lifetime annotations

Code

- Kernel: https://github.com/thejh/linux branch khp
- Compiler: https://github.com/thejh/llvm-project branch khp
- Slides: https://sched.co/ckpO

Bonus slides

(in case we have too much time left at the end)
(which we definitely won't)
(aaah I have to move so many slides into the bonus section)

Handwavy future plans: OOB access

- no classic "OOB access detection":
    - only detects inter-object overflow
    - not a good fit for object-level checks
- instead, focus on type checks:
    - intrinsically object-level
    - detects type confusion, too
    - for arrays, treat length as part of type information
    - most accesses are probably to single objects
    - hopefully easier to elide
        - variable/member annotation for "this is a live type-checkable pointer"?
        - may require generics-style annotations for lists
- 16 bits are still free in live object metadata
    - should be enough for most types - rest has to use out-of-line storage

Micro-Optimization: Equal-Hamming-Weight IDs

- Assign 8-bit IDs with hamming weight 4 to CPUs (80 IDs possible)
- Store inverted IDs in object metadata
- For two valid IDs, `ID_A & ~ID_B` is zero iff the IDs are the same
- `ID_A & 0` is always zero

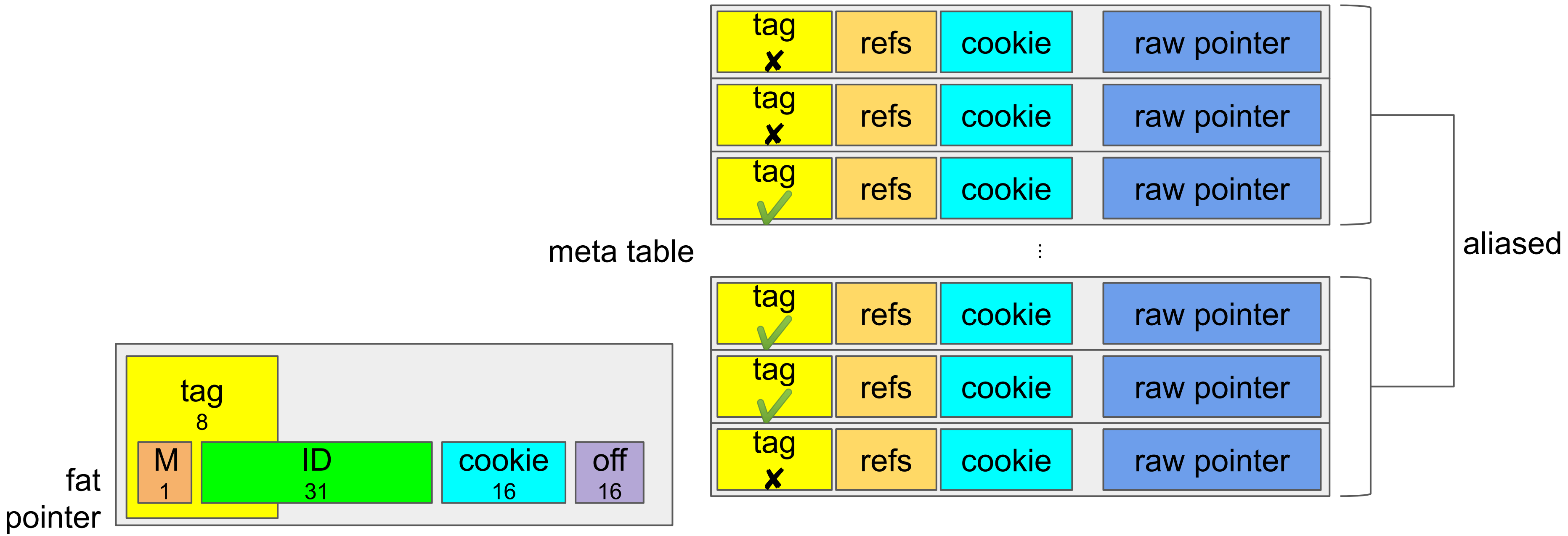
Pseudocode:

```
u8 me = get_current_cpu_num();
u8 stored = READ_ONCE(meta->inverted_cpu_num);
if (stored & me)
  WRITE_ONCE(meta->cpu_num, 0);
```

CPU	ID (in binary)
0	00001111
1	00010111
2	00011011
...	...

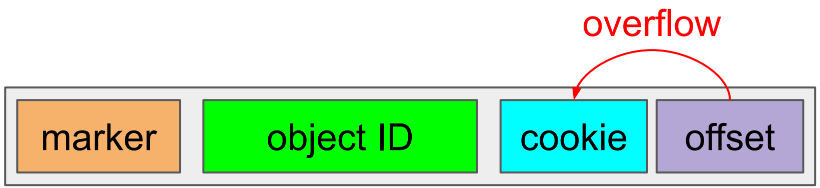
Fallback physical memory reuse [impl incomplete]

- Rough idea: In pointer encoding, steal ID bits to enlarge cookie
- Adjustable ID:cookie split per meta page
- 8-bit tag (top bits of fat pointer) to select which aliased object IDs are valid



Objects >=0x10000 bytes [not yet implemented when the slides were due]

- Important for kmalloc_large coverage (not slab-based)
- Legitimate pointer arithmetic can overflow the offset
- Basic idea: Steal cookie bits for the offset
- Solution:
  - Accept ceil((size+1)/2^{16}) different cookies in cookie check slowpath
  - Bump cookie accordingly on freeing
  - Theoretically permits <4GiB objects, smaller limit in practice for fat-pointer-ASLR
- Cost:
  - Fat pointers become slightly more guessable
  - Faster cookie depletion



Optimization: Fast single-read access [unimplemented?]

For single 8-byte loads with no merging:

- Perform data read **before** cookie check
- Omit pinning logic
- Omit CPU number tracking

Incompatible: constructor/RCU slabs

- constructor slab
  - object initialization on slab page alloc
  - self-referential pointers may exist => address can't change
  - will also be an issue for memory tagging / HWASAN
  - potential solution: re-invoke `->ctor()` for each allocation?
- RCU slab: use-after-free access permitted after reallocation
  - relies on constructor slabs
  - also an issue for KASAN
  - potential solution: enforce RCU-delayed object freeing?
    - turn `kmem_cache_free(x)` into `call_rcu(x + cache->rcu_head_offset, __kmem_cache_free_rcu)` ?
    - might further worsen cache locality a bit

Intentional OOB pointer calculation breaks stuff

```
static inline u32 __pure
crc32_body(u32 crc, unsigned char const buf, size_t len, const u32 (*tab)[256])
{
[...]
        const u32 *b;
[...]
        b = (const u32 *)buf;
[...]
        --b;
        for (i = 0; i < len; i++) {
[...]
                q = crc ^ *++b; /* use pre increment for speed */
[...]
        }
[...]
}
```

already UB according to [C89, "3.3.6 Additive Operators"](#)!

Resurrectable wrapper around rcu_head

```
static void rcu_cb(struct rcu_head *h) {
  struct rcu ref *ref = container of(h, struct rcu_ref, rcu_head);
  if (atomic read(&ref->refs) & RESURRECTED) {
    if (atomic sub and test(&ref->refs, RESURRECTED))
      call_rcu(&ref->rcu_head, rcu_cb);
  } else {
    h->cb(h);
  }
}
void ref_dec(struct rcu_ref *ref) {
  if (atomic dec and test(&ref->refs))
    call_rcu(&ref->rcu_head, rcu_cb);
}
void ref_inc(struct rcu_ref *ref) {
retry:
  if (atomic read(&ref->refs) == 0) {
    if (atomic_cmpxchg(&ref->refs, 0, RESURRECTED + 1) != 0) goto retry;
  } else {
    if (!atomic_inc_not_zero(&ref->refs)) goto retry;
  }
}
```

```
#define RESURRECTED 1UL<<31
struct rcu_ref {
  struct rcu_head rcu_head;
  atomic_t refs;
  void (*cb)(struct rcu_ref *);
};
void ref_init(struct rcu_ref *ref,
      void (*cb)(struct rcu_ref *)) {
  atomic_set(&ref->refs, 1);
  ref->cb = cb;
}
```