MTE As Tested

Mark Brand

POC 2023

About Me

- Security / vulnerability research for ~12 years

- Researcher at Project Zero for ~8 years

About Me

- Security / vulnerability research for ~12 years
- Researcher at Project Zero for ~8 years

About MTE

- On Pixel 8 you can configure sync MTE as the default for most* apps today!

Enabling MTE

```
markbrand@markbrand$ adb shell
shiba:/ $ setprop arm64.memtag.bootctl memtag
shiba:/ $ setprop persist.arm64.memtag.default sync
shiba:/ $ setprop persist.arm64.memtag.app_default sync
shiba:/ $ reboot
```

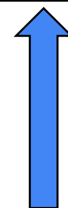
Background

About ~~MTE~~ 64-bit virtual addressing

0x4141414141414141

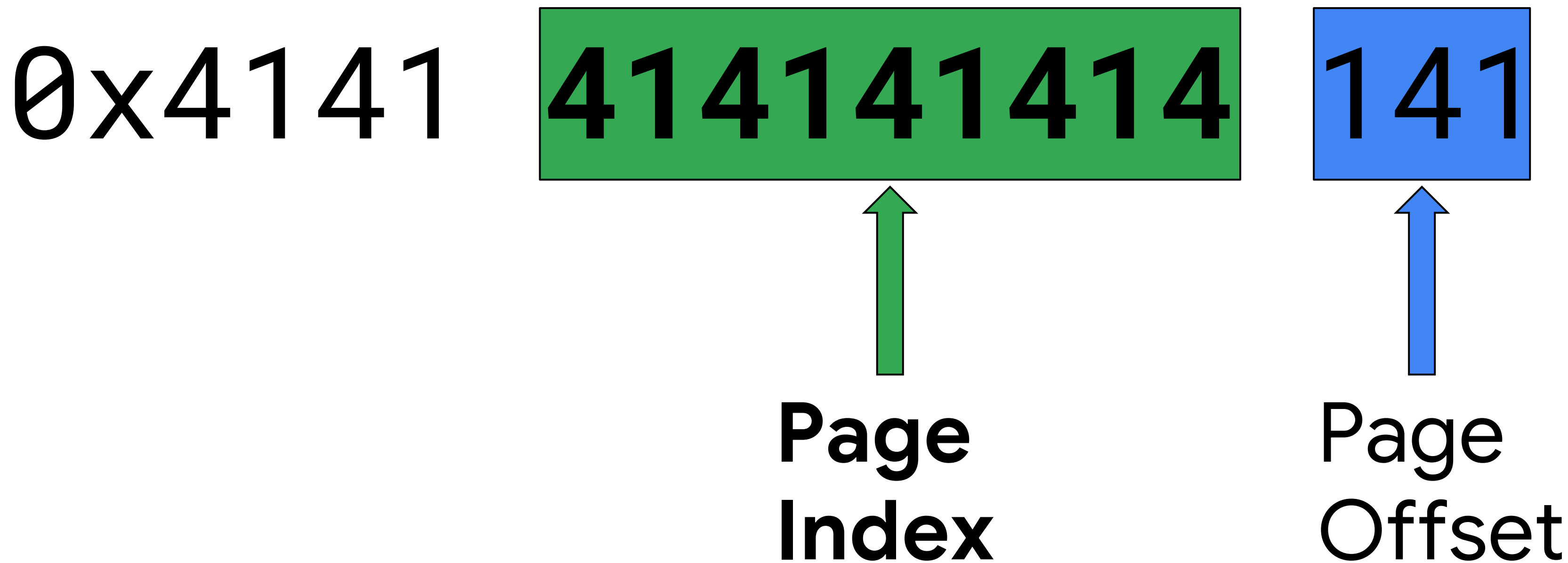
About ~~MTE~~ 64-bit virtual addressing

0x41414141414141414 **141**

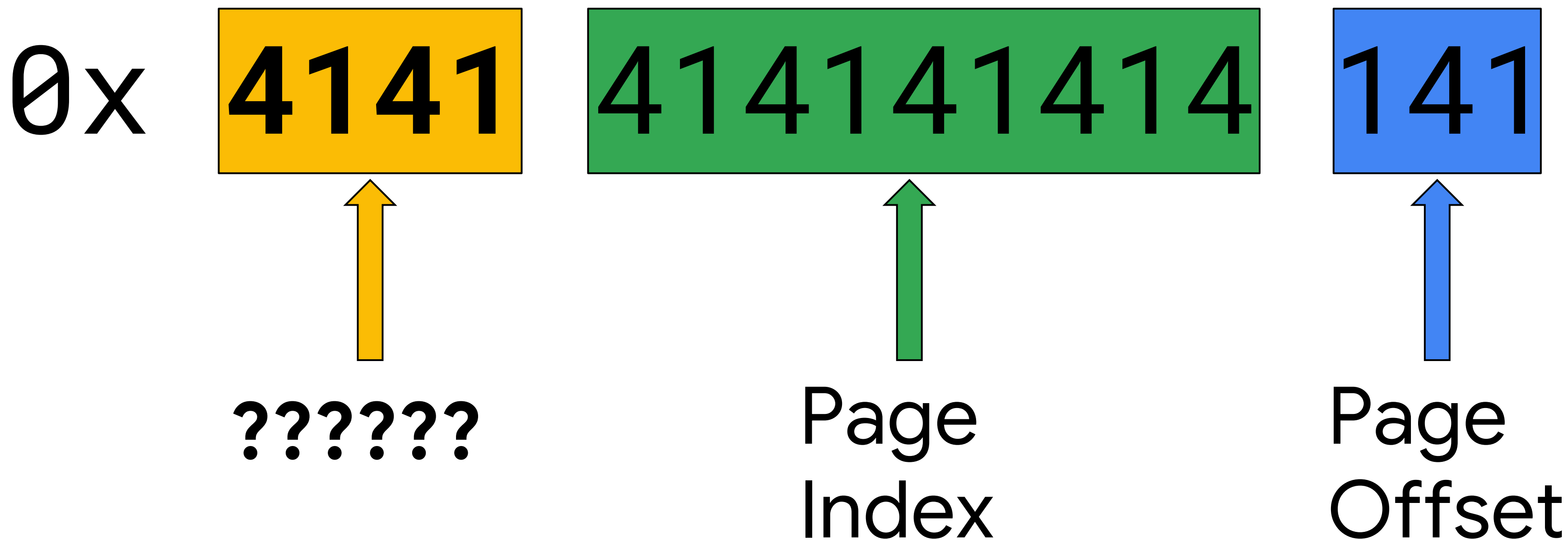


Page Offset

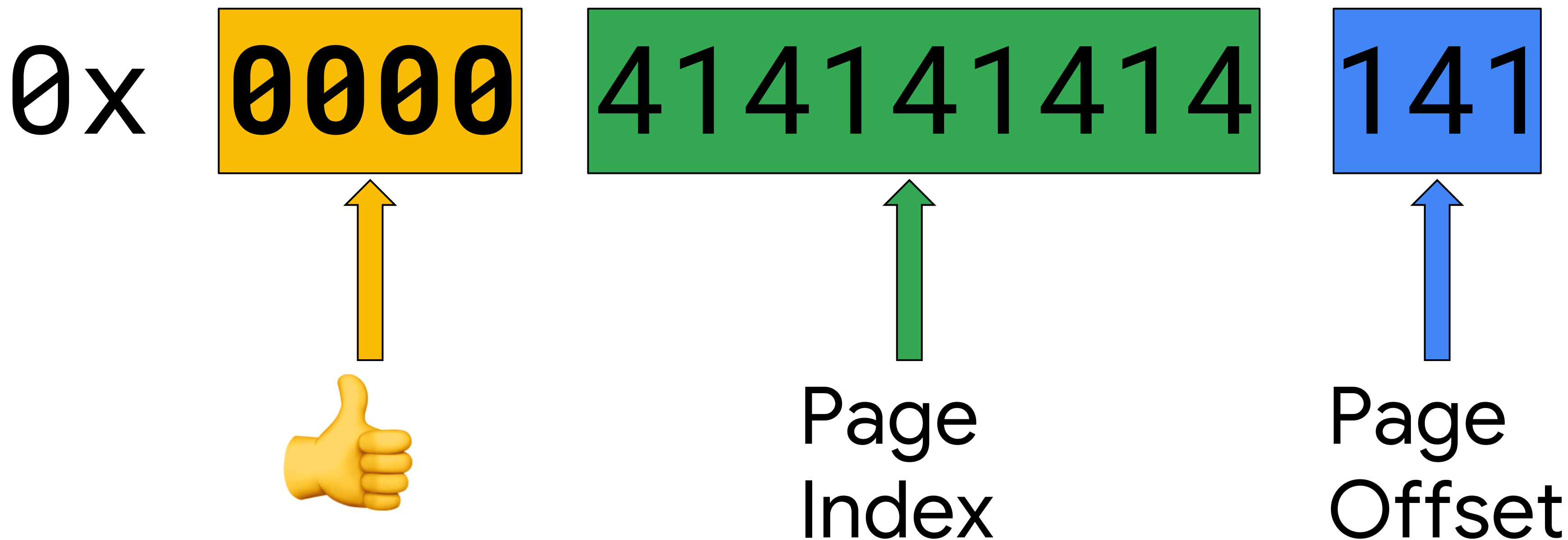
About ~~MTE~~ 64-bit virtual addressing



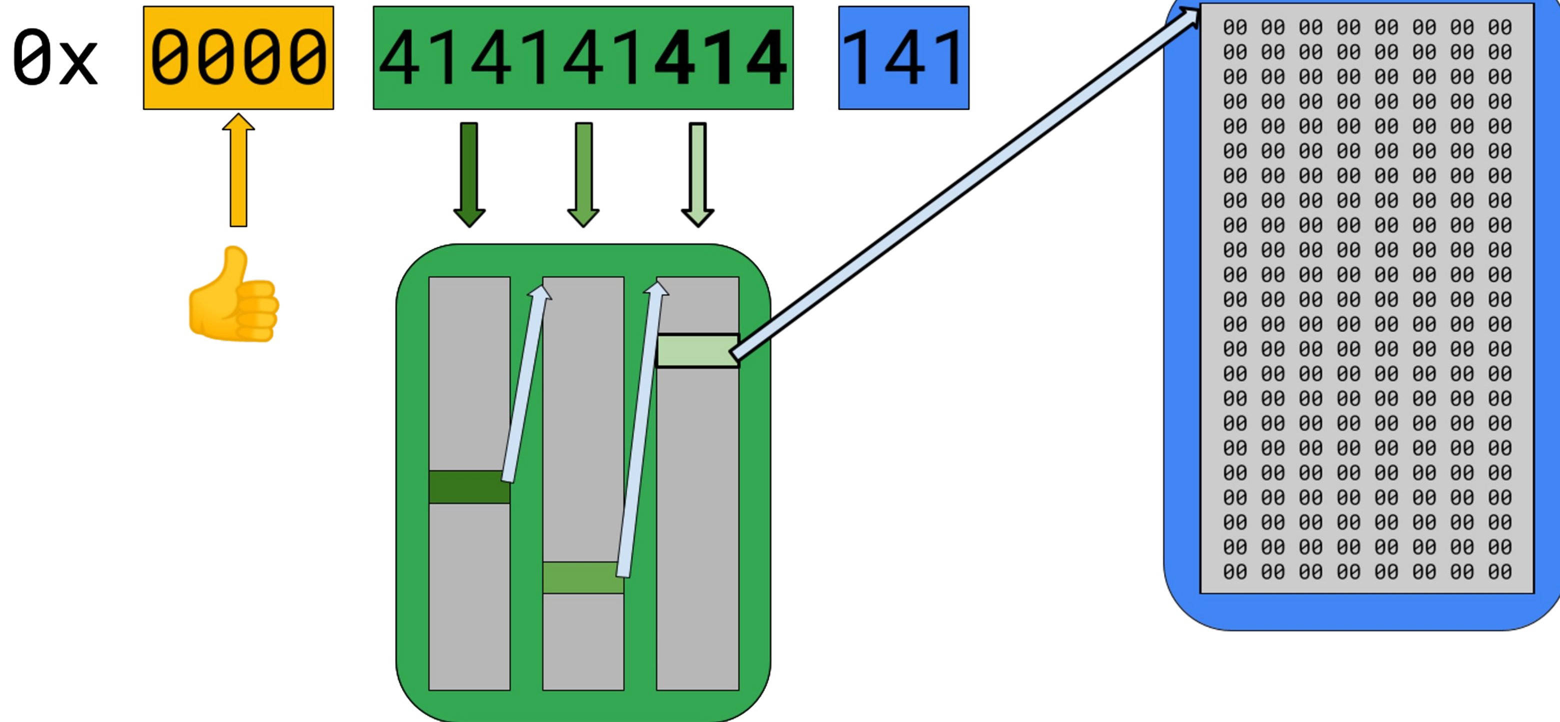
About ~~MTE~~ 64-bit virtual addressing



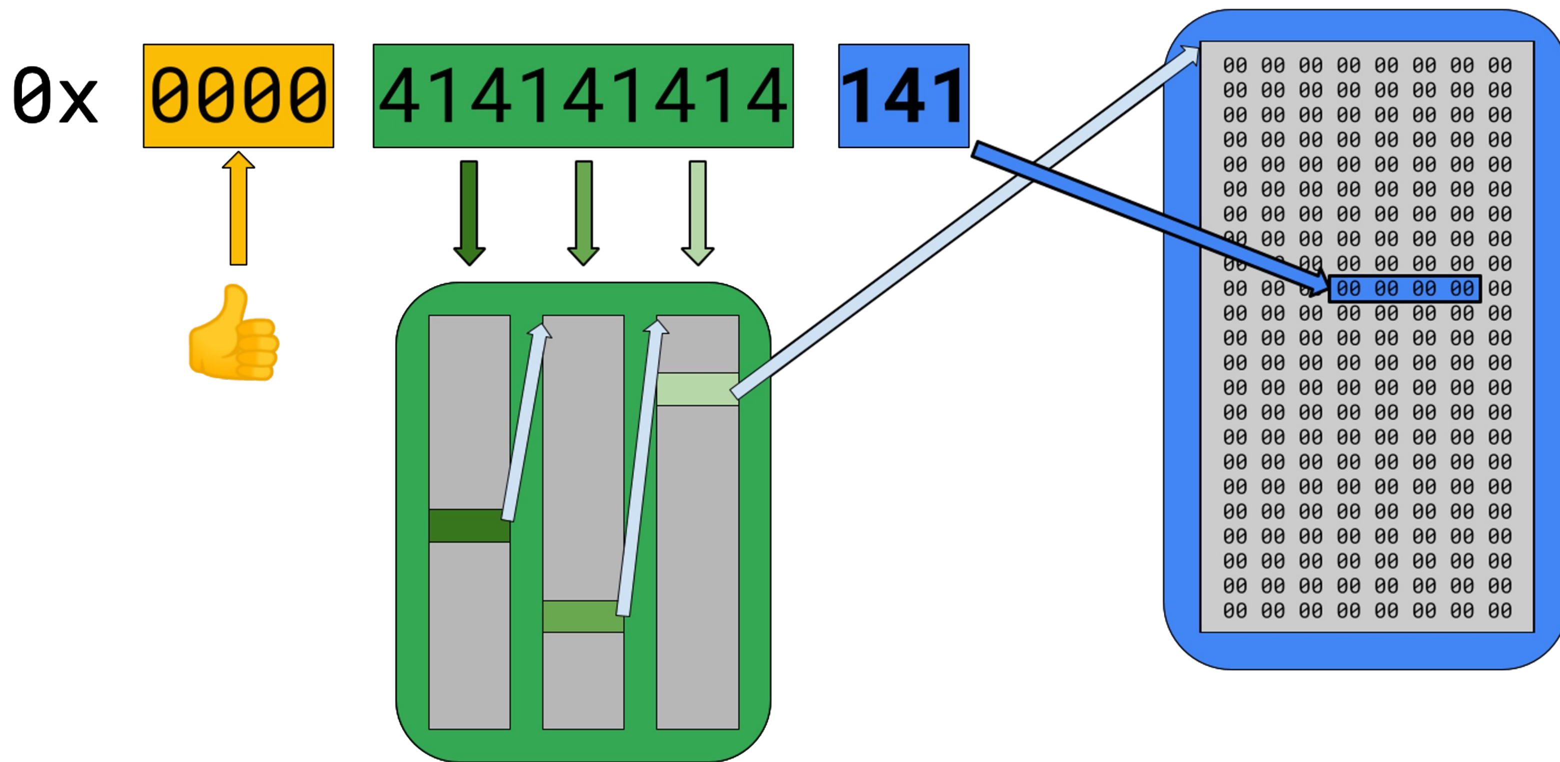
About ~~MTE~~ 64-bit virtual addressing



About ~~MTE~~ 64-bit virtual addressing

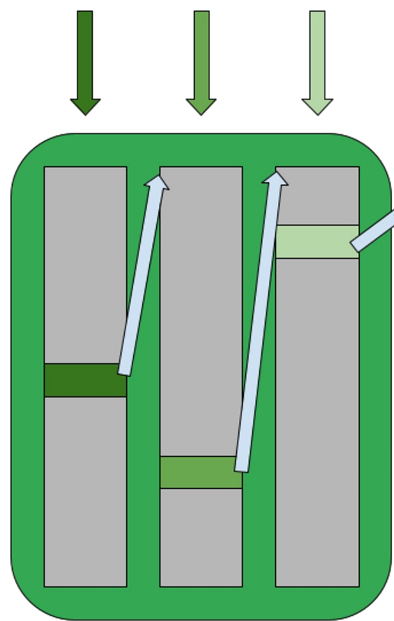


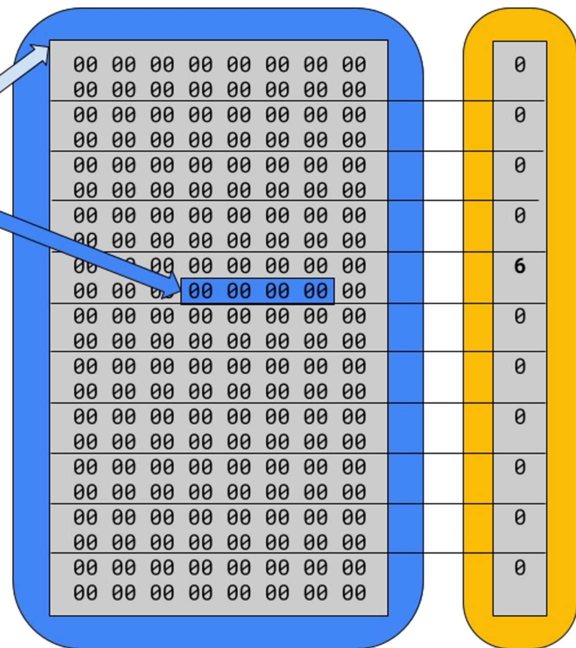
About ~~MTE~~ 64-bit virtual addressing



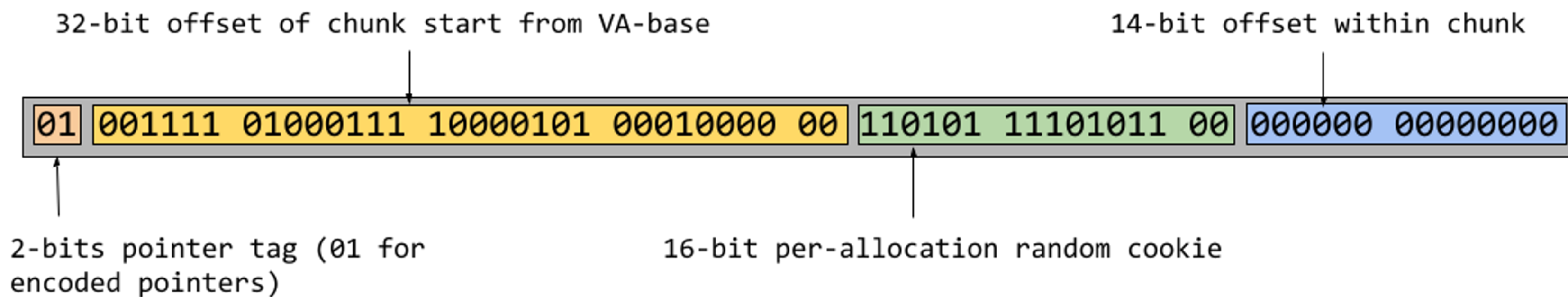
About MTE

0x 6 000 414141414 141

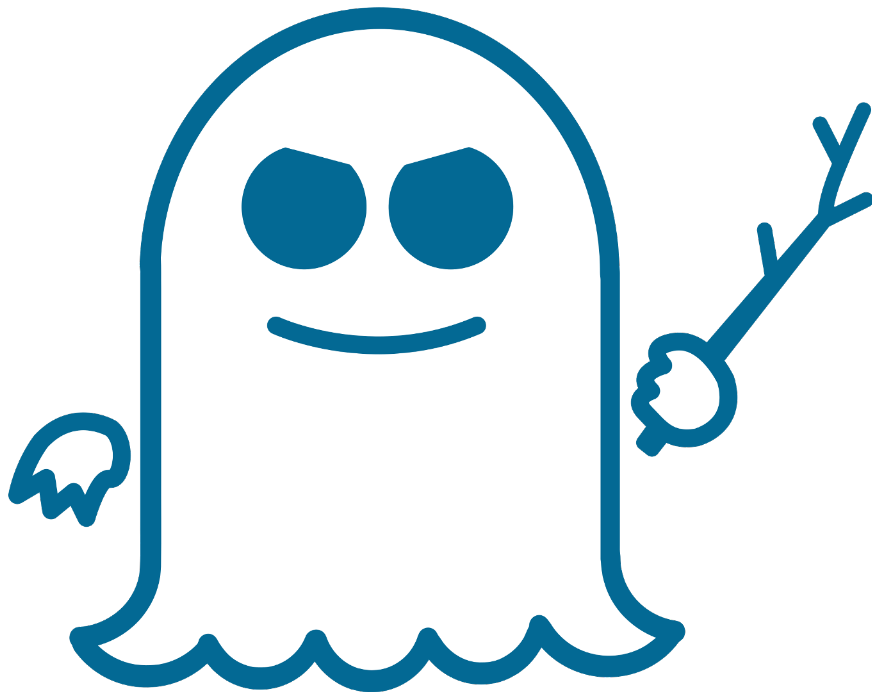




MTE and Me: Kernel Heap Protector



MTE and Me: Speculative Execution



MTE as a security "boundary"

Direct MTE induced speculative oracle?

Direct MTE induced speculative oracle?

ASYNC limitations?

Direct MTE induced speculative oracle?

Software limitations?

ASYNC limitations?

Known-tag attacks
vs.
Unknown-tag attacks

Known-tag attacks (SYNC + ASYNC)

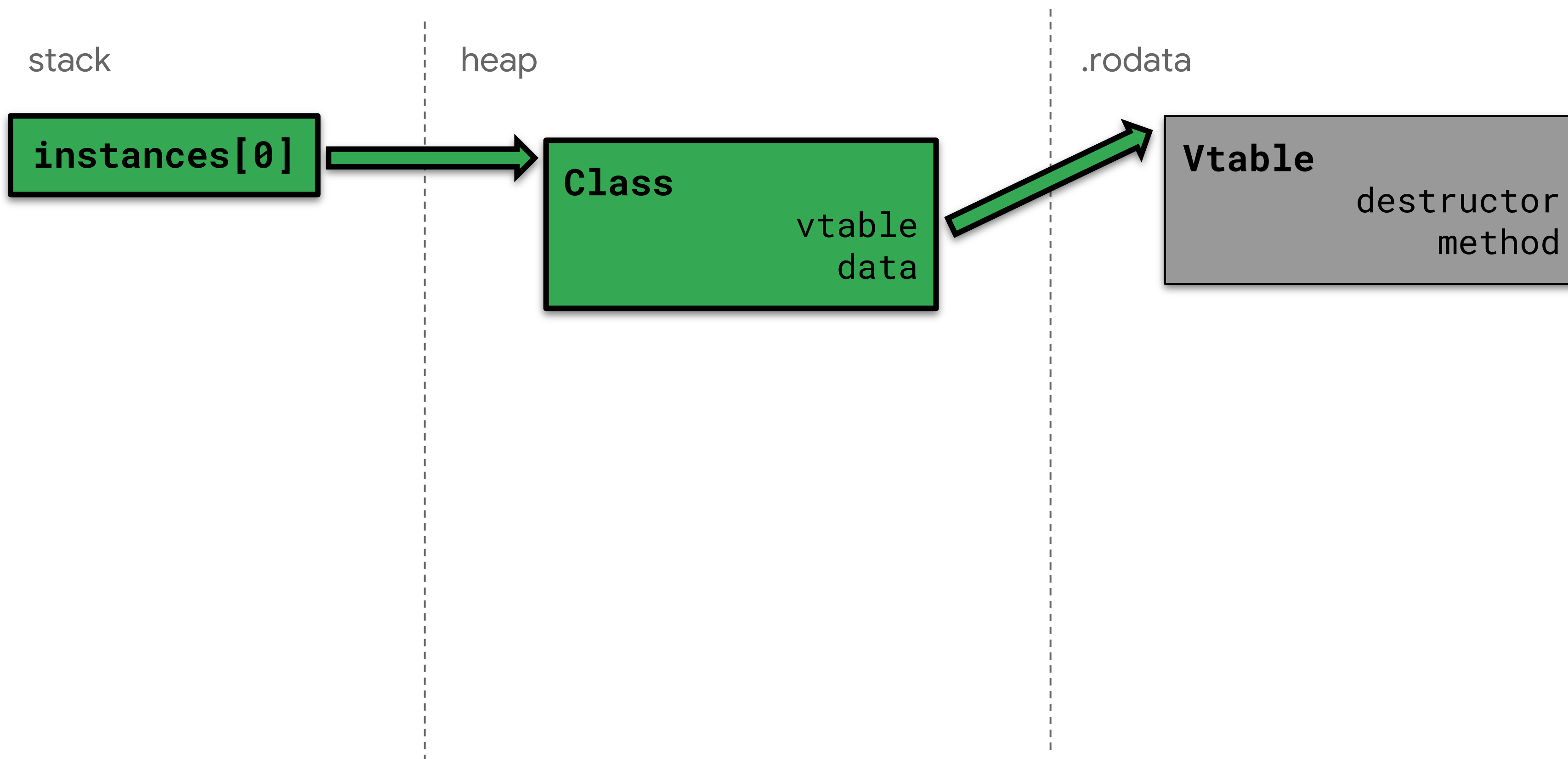
```
case '1': // "Alloc"
  idx = ipc_read(in_pipe);
  if (idx < 0) {
    break;
  }

  if (instances[idx]) {
    instances[idx]->vtable->destructor(instances[idx]);
    free(instances[idx]);
  }

  data = ipc_read_string(in_pipe);

  instances[idx] = malloc(sizeof(struct Class));
  fprintf(stderr, "instances[%i] = %p (%p)\n", idx, instances[idx], data);
  Class_constructor(instances[idx], data);

  break;
```

```
case '2': // "Free"
  idx = ipc_read(in_pipe);
  if (idx < 0 || !classes[idx]) {
    break;
  }

  if (instances[idx]) {
    instances[idx]->vtable->destructor(instances[idx]);
    free(instances[idx]);
  }

  // Bug: we don't set class to NULL, so we're left with a dangling
  // pointer.

  break;
```

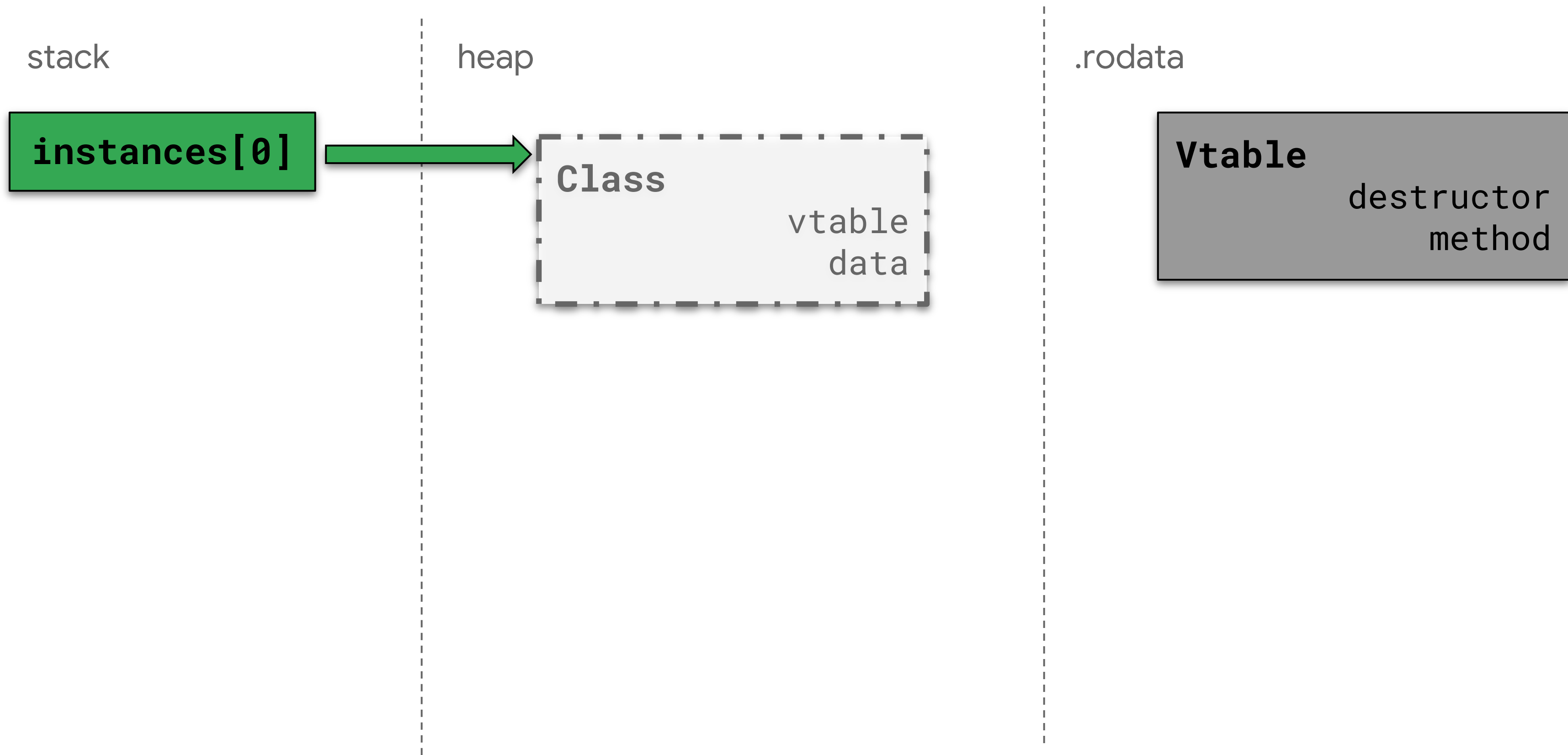
```
case '3': // "Replace"
  if (replacement) {
    replacement->vtable->destructor(replacement_instance);
    free(replacement);
  }

  data = ipc_read_string(in_pipe);

  replacement = malloc(sizeof(struct ReplacementClass));
  fprintf(stderr, "replacement = %p\n", replacement);
  ReplacementClass_constructor(replacement, data);
  data = NULL;

  break;
```

```
case '4': // "Write"
  idx = ipc_read(in_pipe);
  if (idx < 0 || !instances[idx]) {
    break;
  }

  if (ipc_write_ready(out_pipe)) {
    instances[idx]->vtable->write(instances[idx], out_pipe);
  }

  break;
```

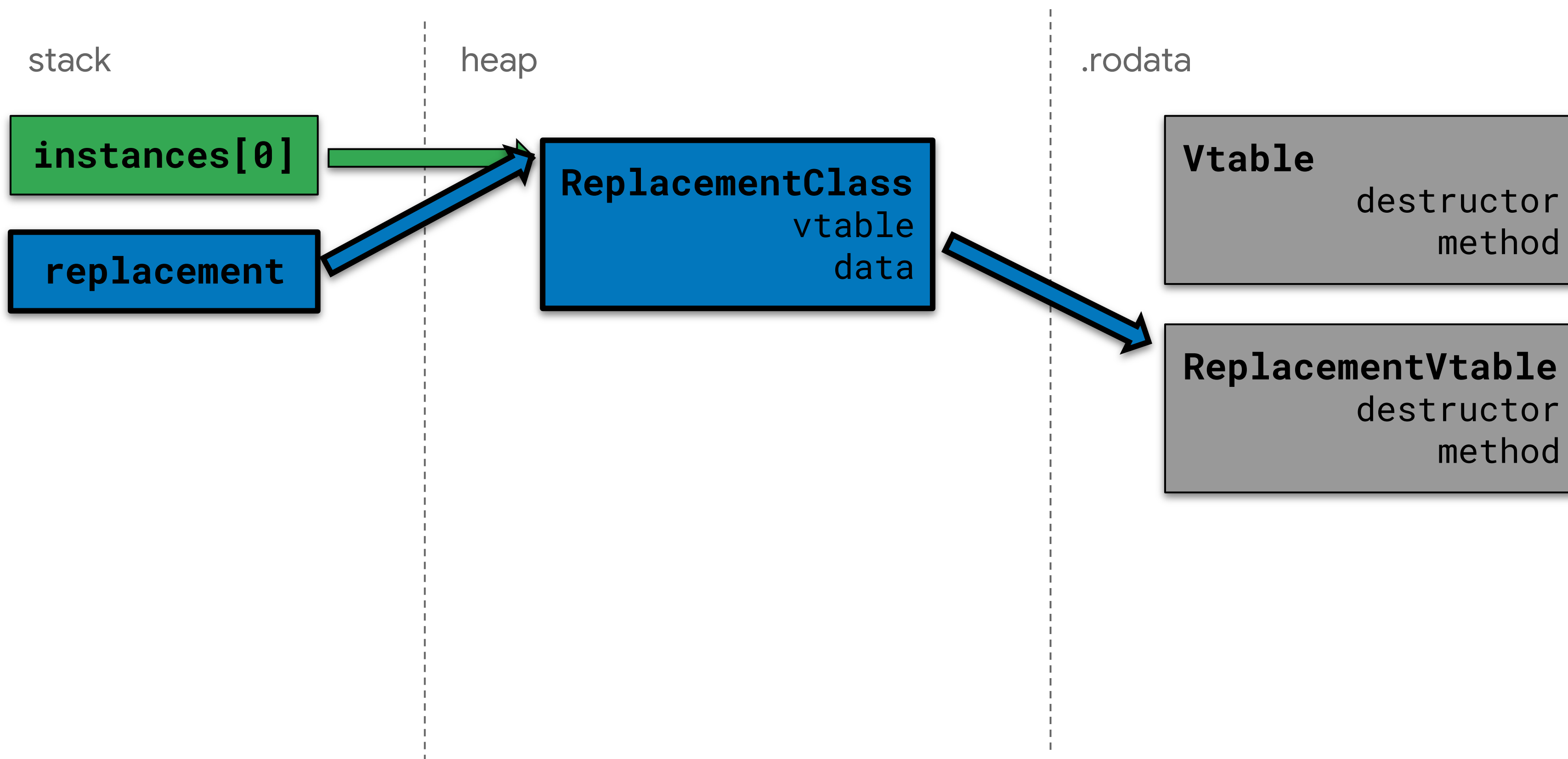
Exploit Flow: Alloc



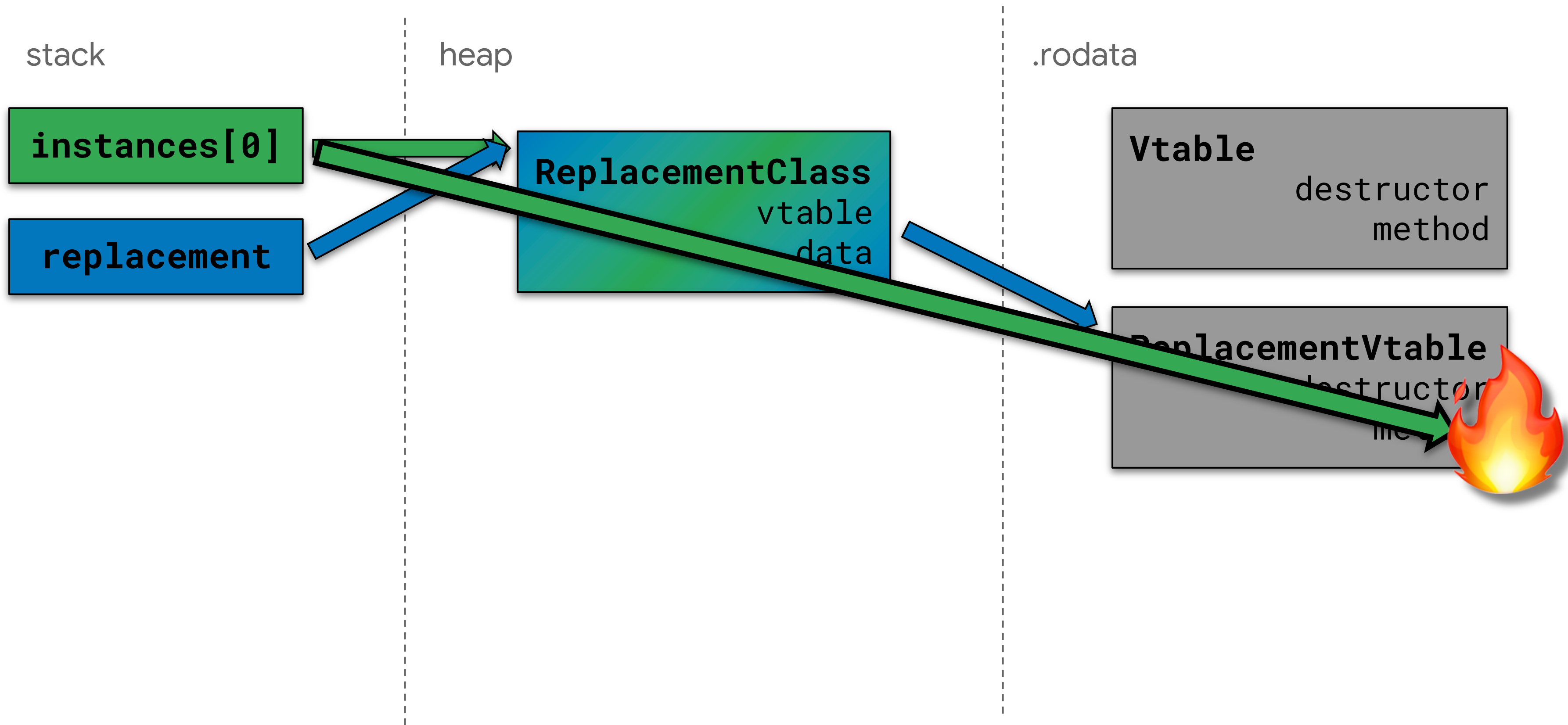
Exploit Flow: Free



Exploit Flow: Replace



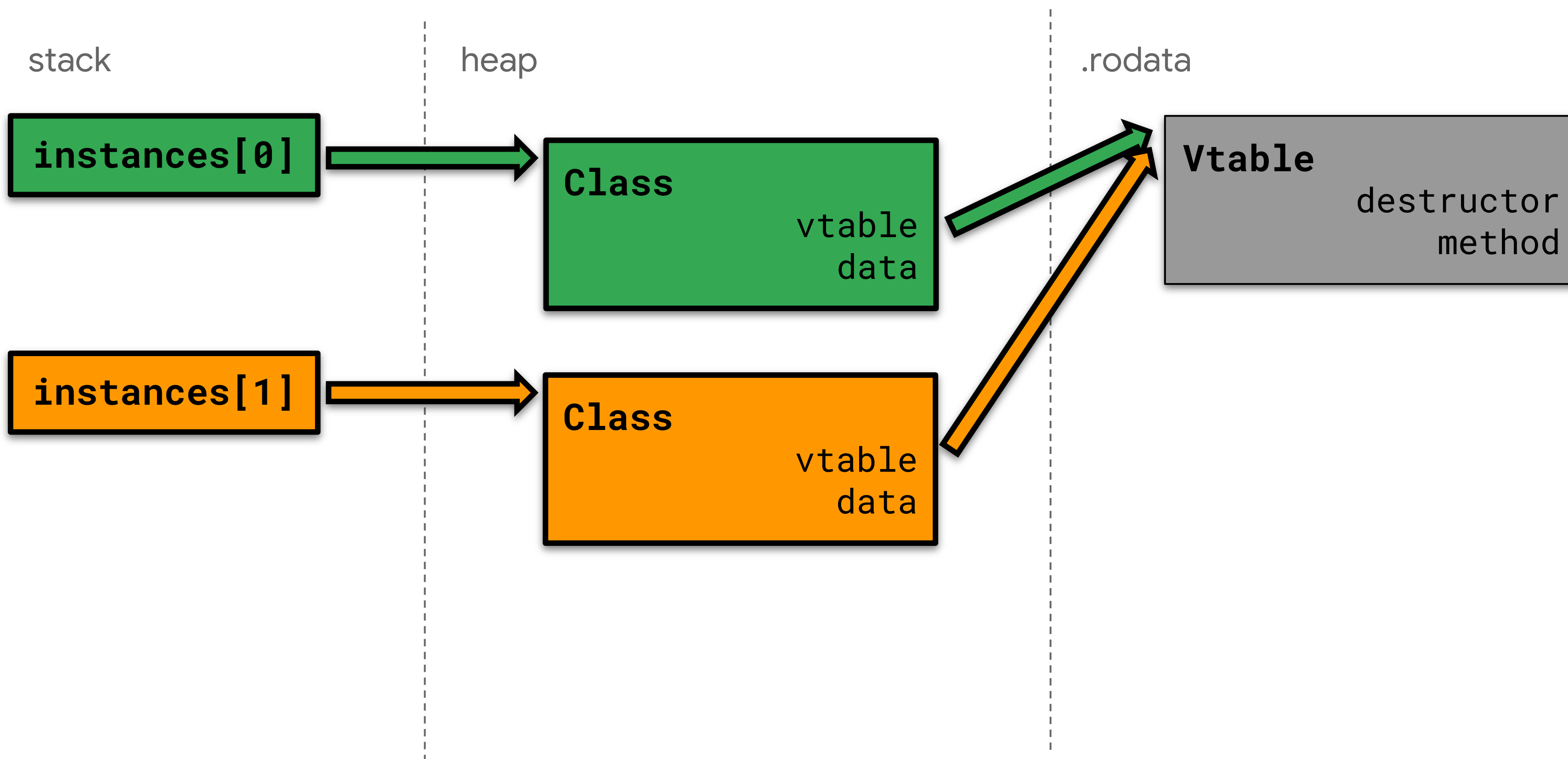
Exploit Flow: Use (type-confusion)



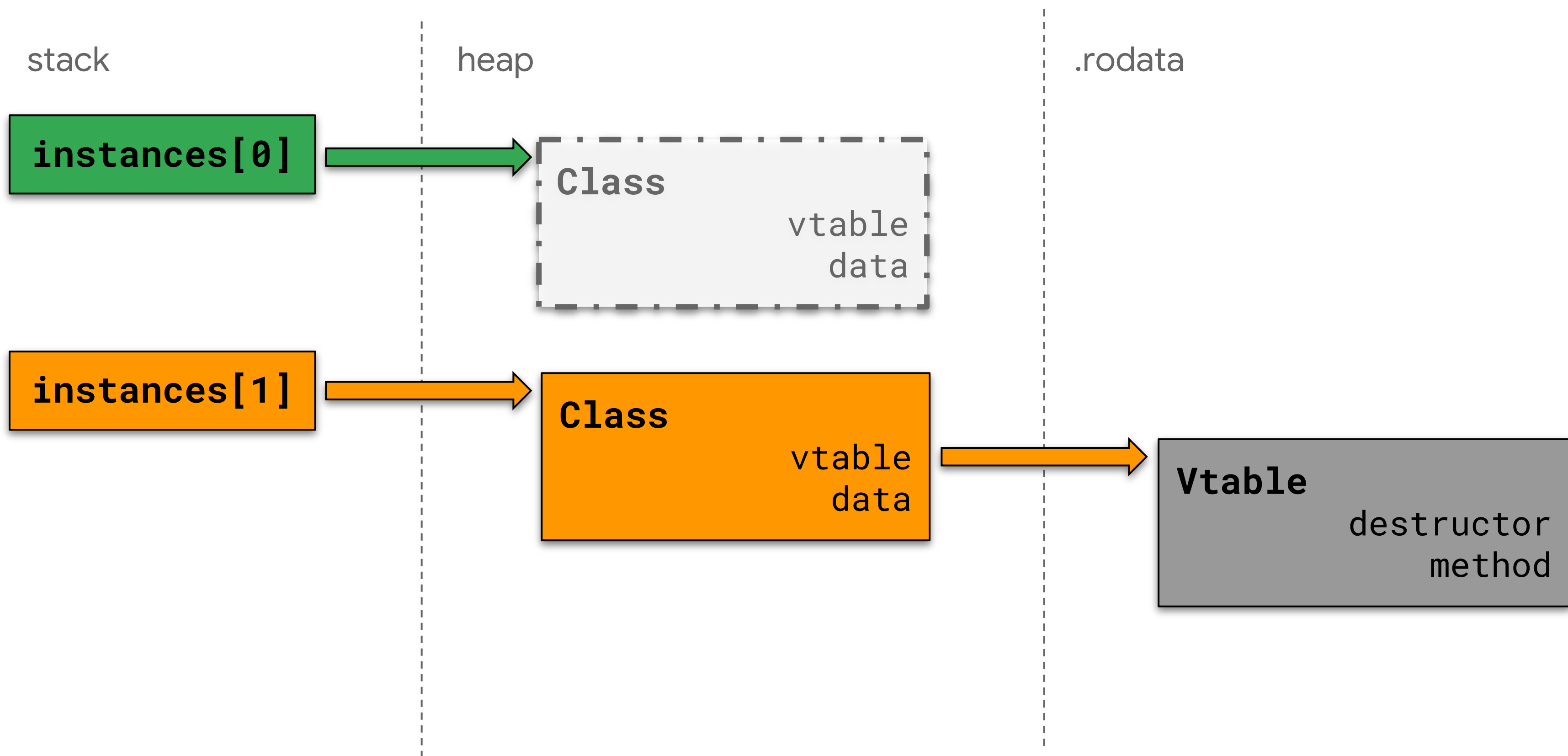
Demo: No tagging

```
shiba:/ $ cd /data/local/tmp
shiba:/data/local/tmp $
```

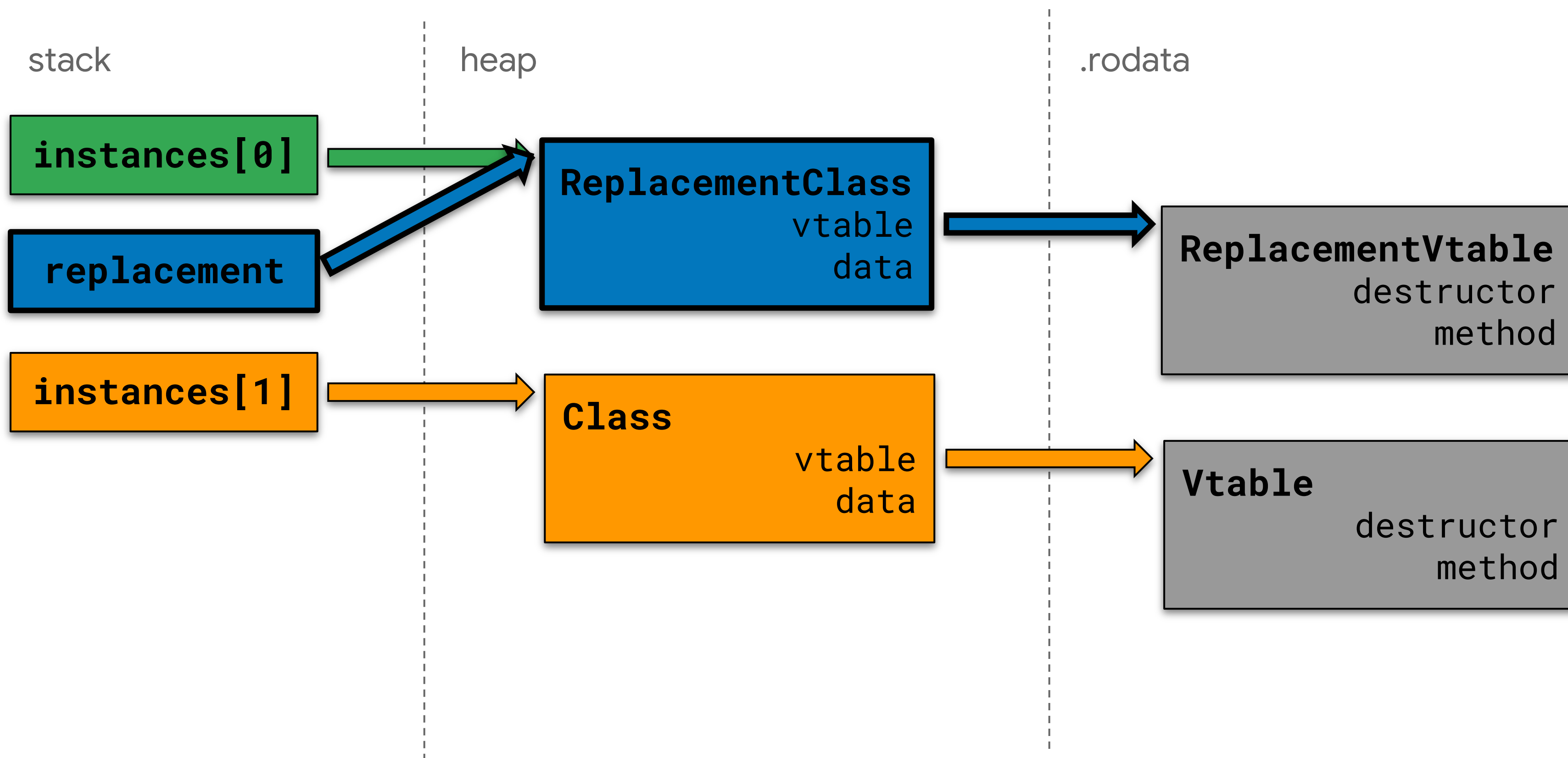
Exploit Flow: Alloc



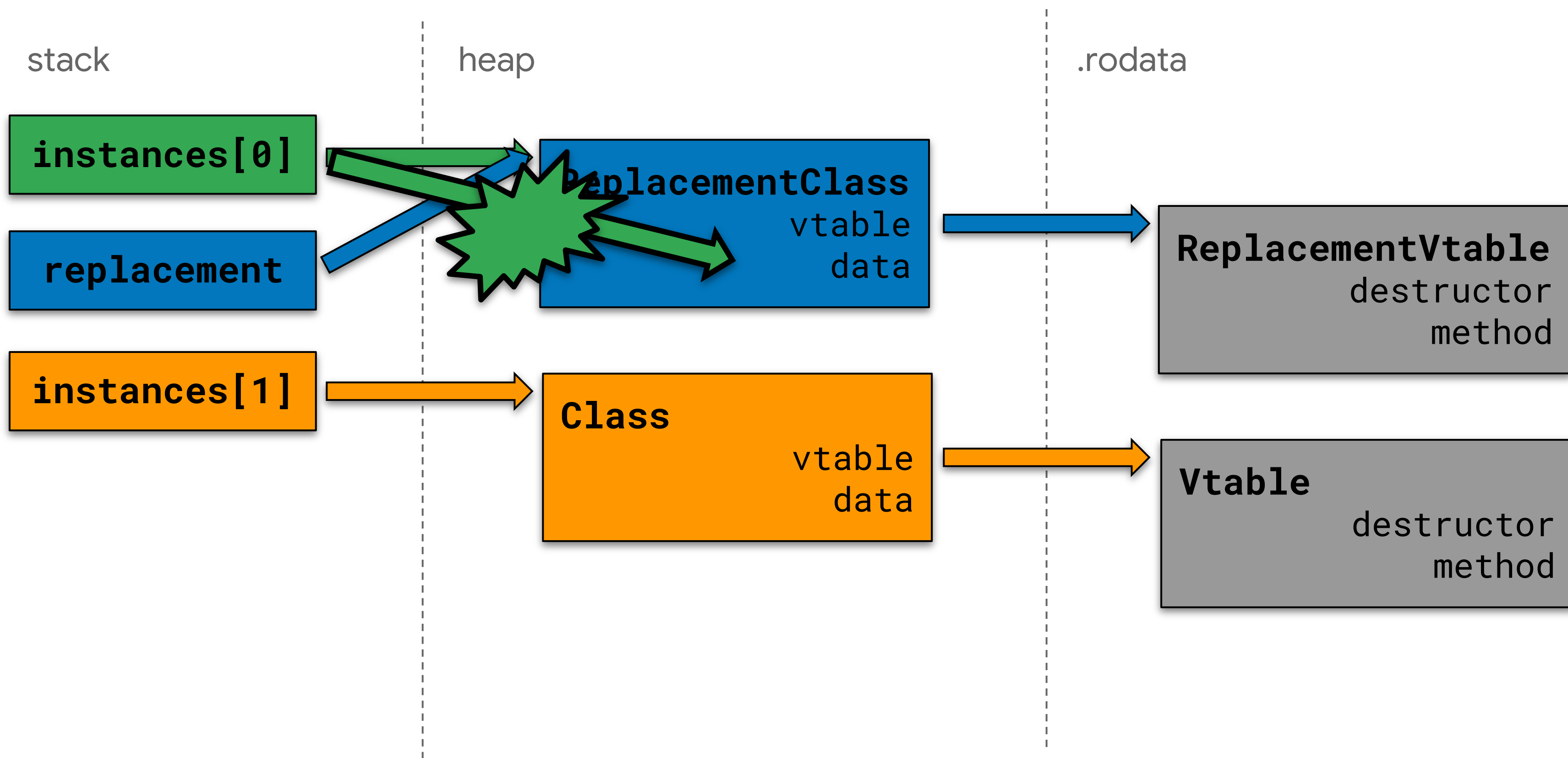
Exploit Flow: Free



Exploit Flow: Replace

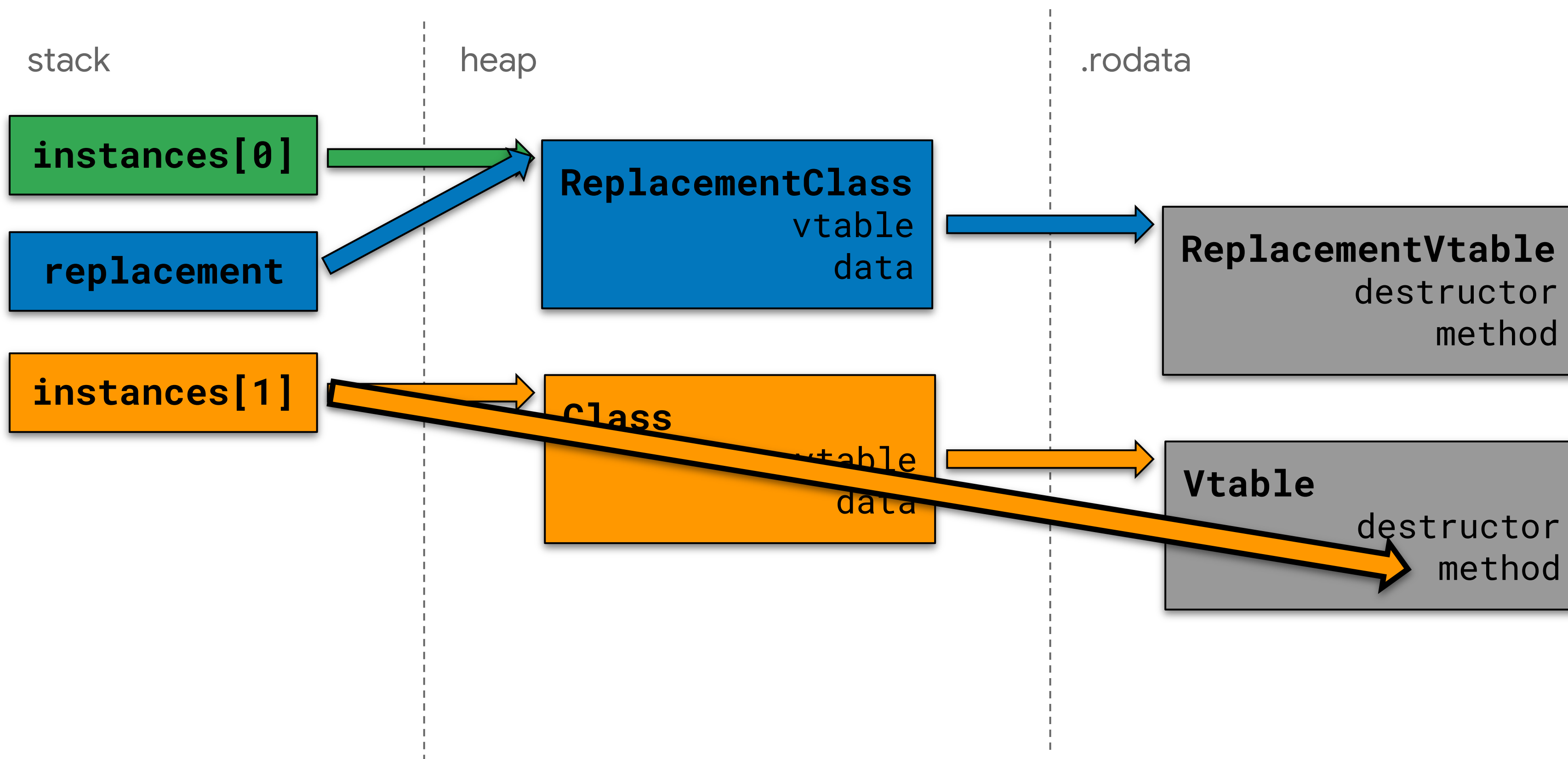


Exploit Flow: Use?

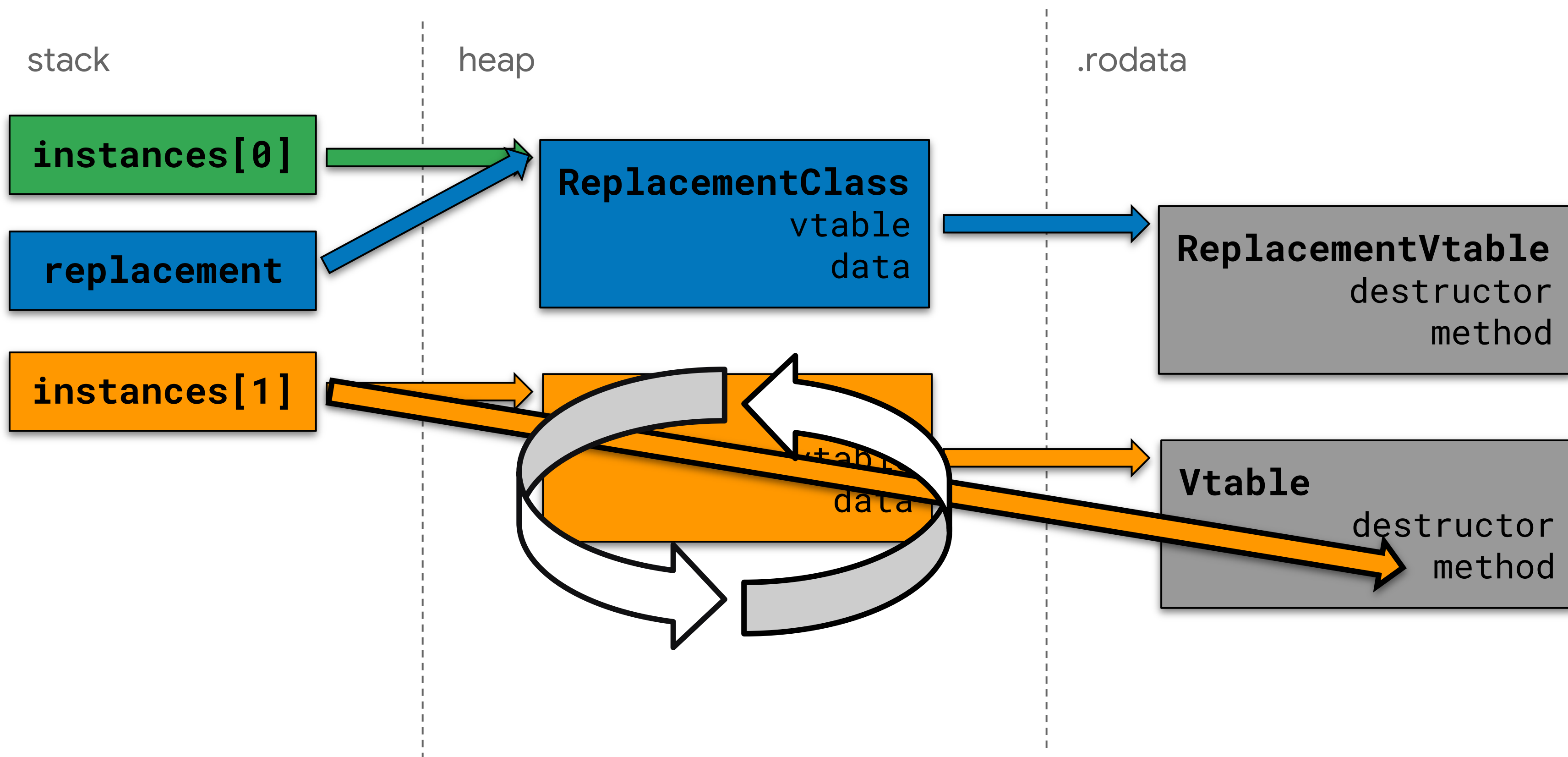


```
case '4': // "Write"
  idx = ipc_read(in_pipe);
  if (idx < 0 || !classes[idx]) {
    break;
  }

  if (ipc_write_ready(out_pipe)) {
    classes[idx]->vtable->write(classes[idx], out_pipe);
  }

  break;
```

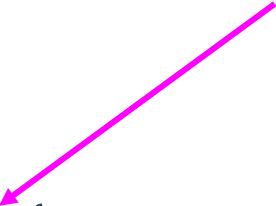
Exploit Flow: Train



Exploit Flow: Train

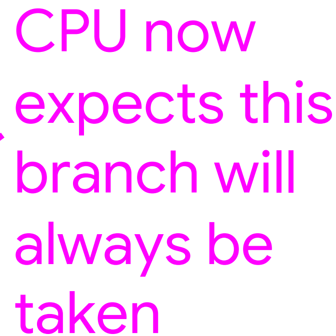


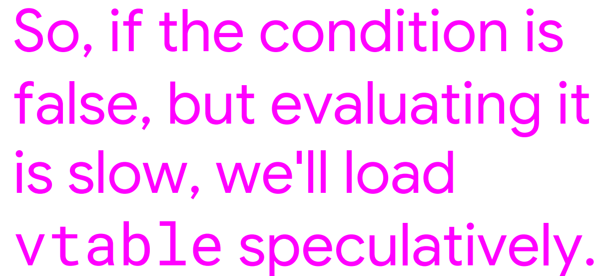
```
case '4': // "Write"
  idx = ipc_read(in_pipe);
  if (idx < 0 || !classes[idx]) {
    break;
  }

   if (ipc_write_ready(out_pipe)) {
     classes[idx]->vtable->write(classes[idx], out_pipe);
   }

  break;
```



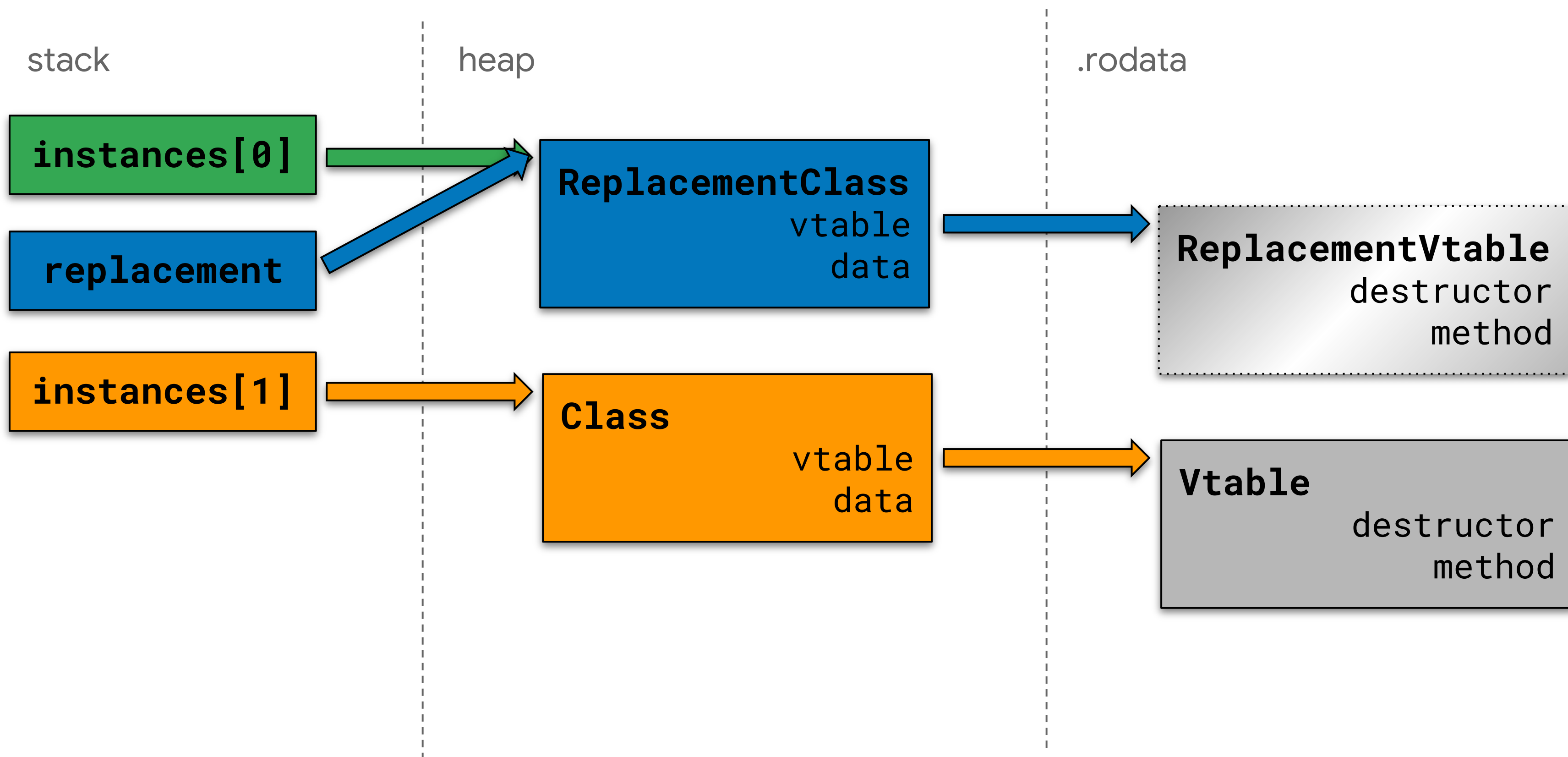
CPU now expects this branch will always be taken

```
case '4': // "Write"
  idx = ipc_read(in_pipe);
  if (idx < 0 || !classes[idx]) {
    break;
  }

    if (ipc_write_ready(out_pipe)) {
      classes[idx]->vtable->write(classes[idx], out_pipe);
    }

  break;
```

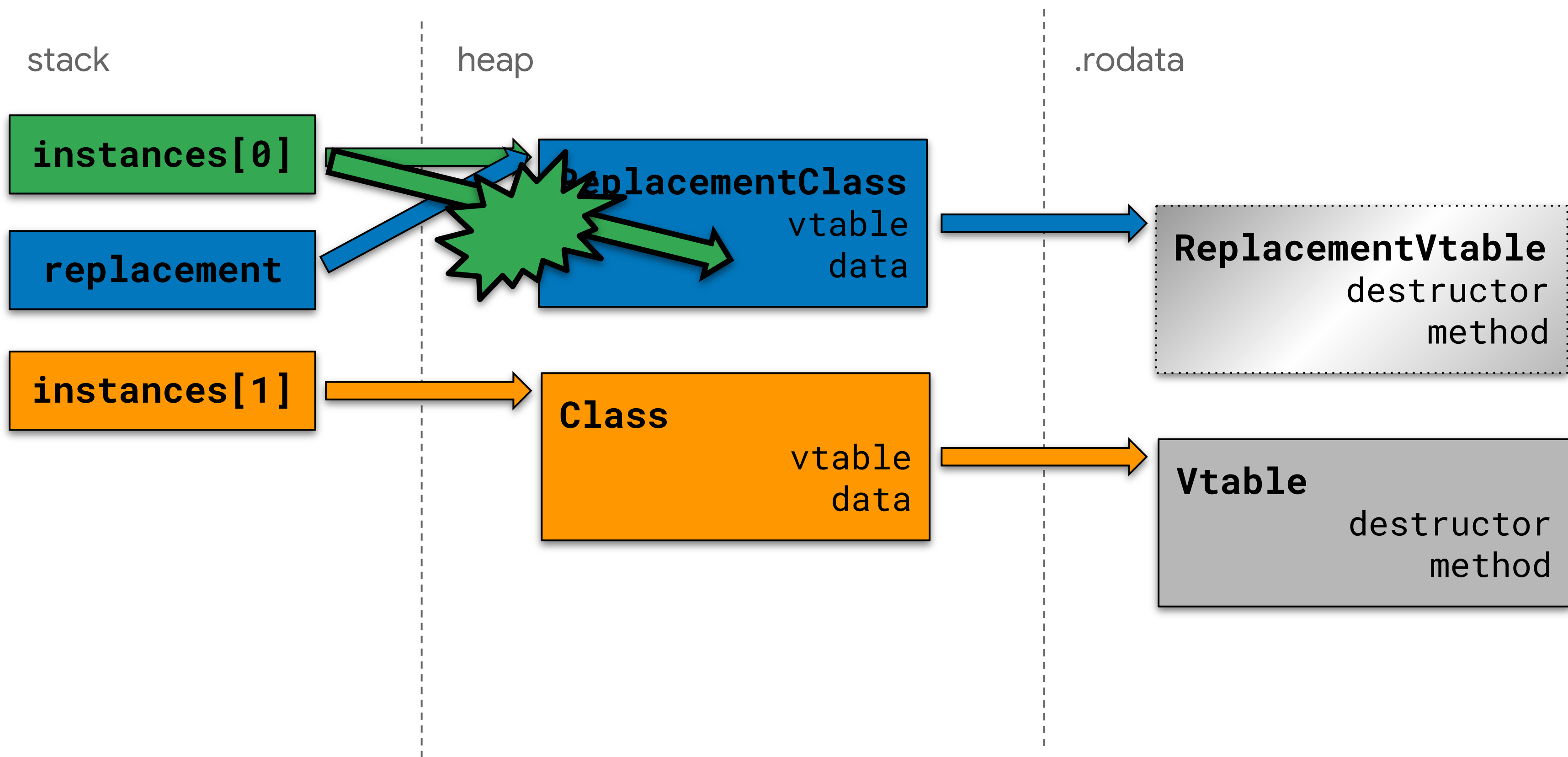




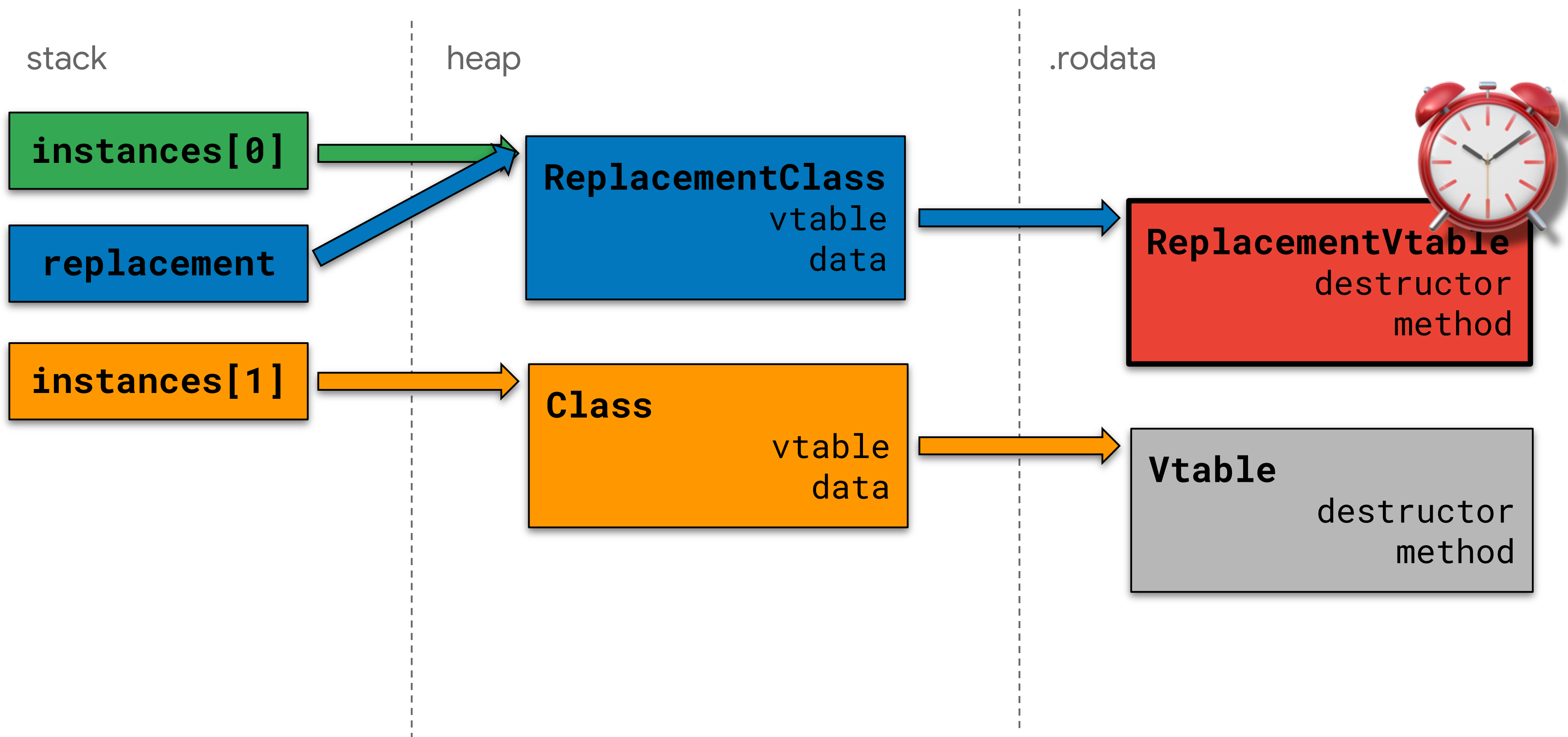
Exploit Flow: Flush



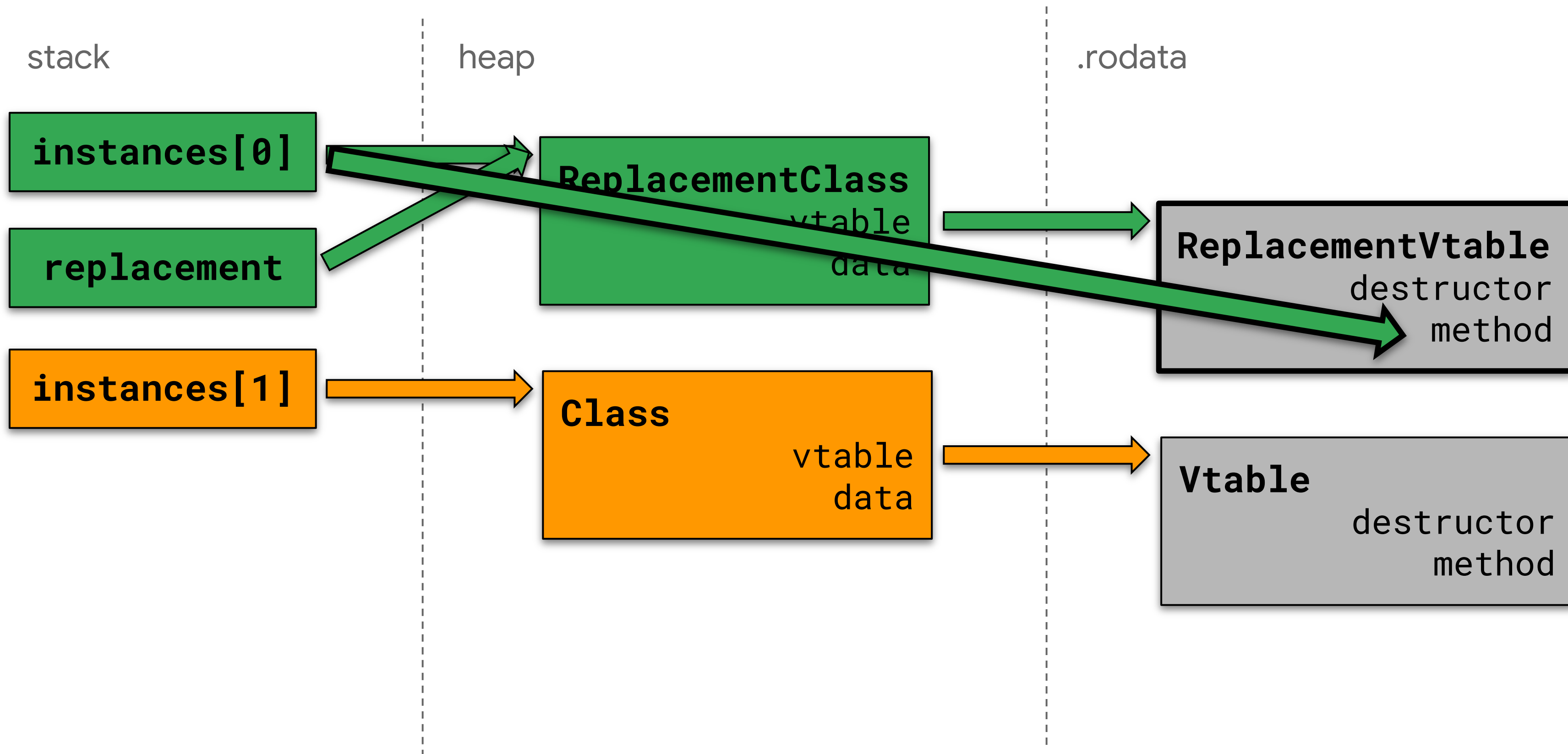
Exploit Flow: Speculative Use [tag mismatch]



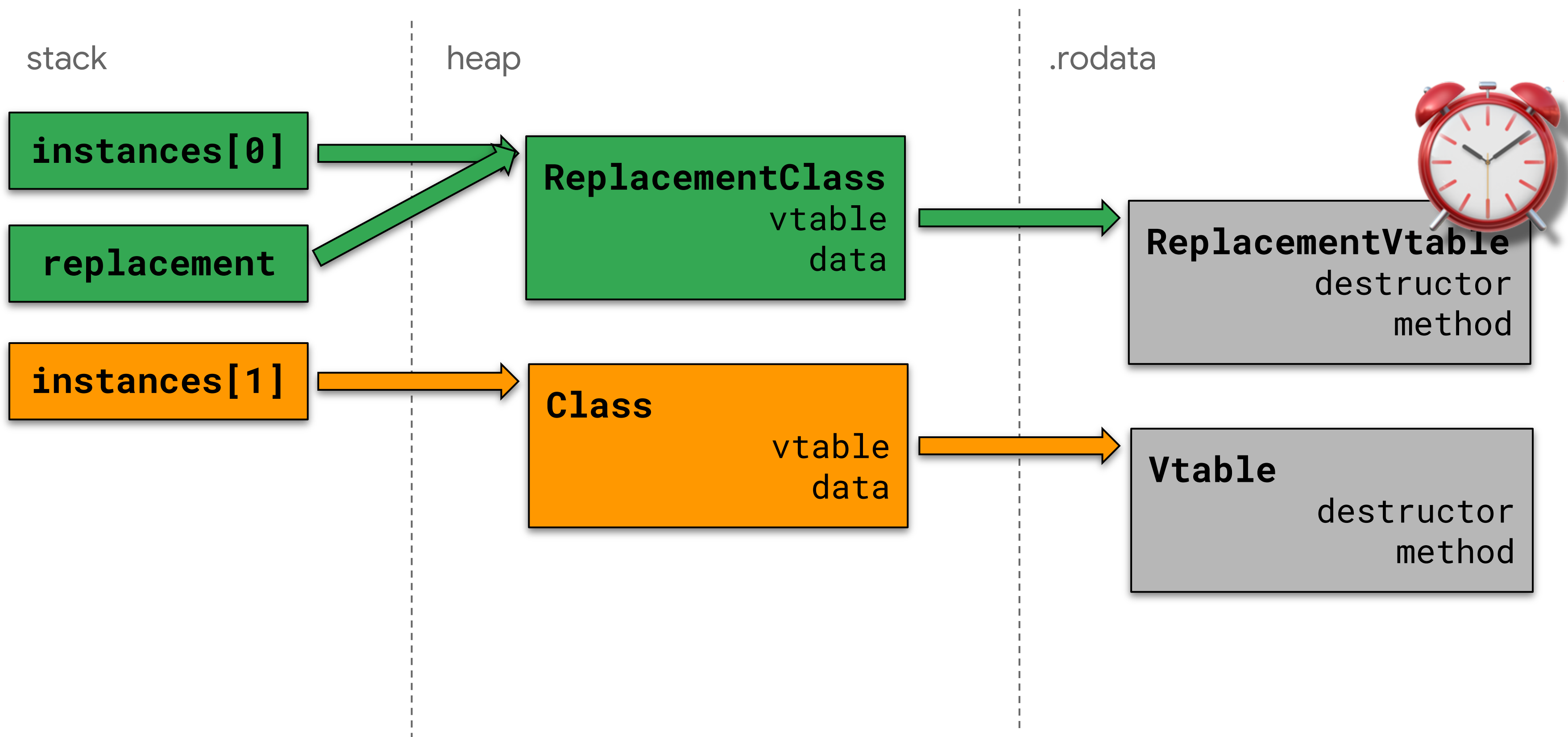
Exploit Flow: Reload



Exploit Flow: Speculative use [tag match]



Exploit Flow: Reload



Demo: Software tagging

```
shiba:/ $ cd /data/local/tmp
shiba:/data/local/tmp $
```

Demo: Hardware tagging (MTE)

```
shiba:/ $ cd /data/local/tmp
shiba:/data/local/tmp $
```

Speculation window length

- With speculative side-channels, we're (typically) using branch misprediction to speculatively execute some instructions that do not execute architecturally.

- The number of instructions executed depends on how long it takes for the misprediction to resolve.

- However, does CPU continue to execute if the instructions are nonsense?

- Can tag-check failure during speculation influence the length of speculation after a failed tag-check?

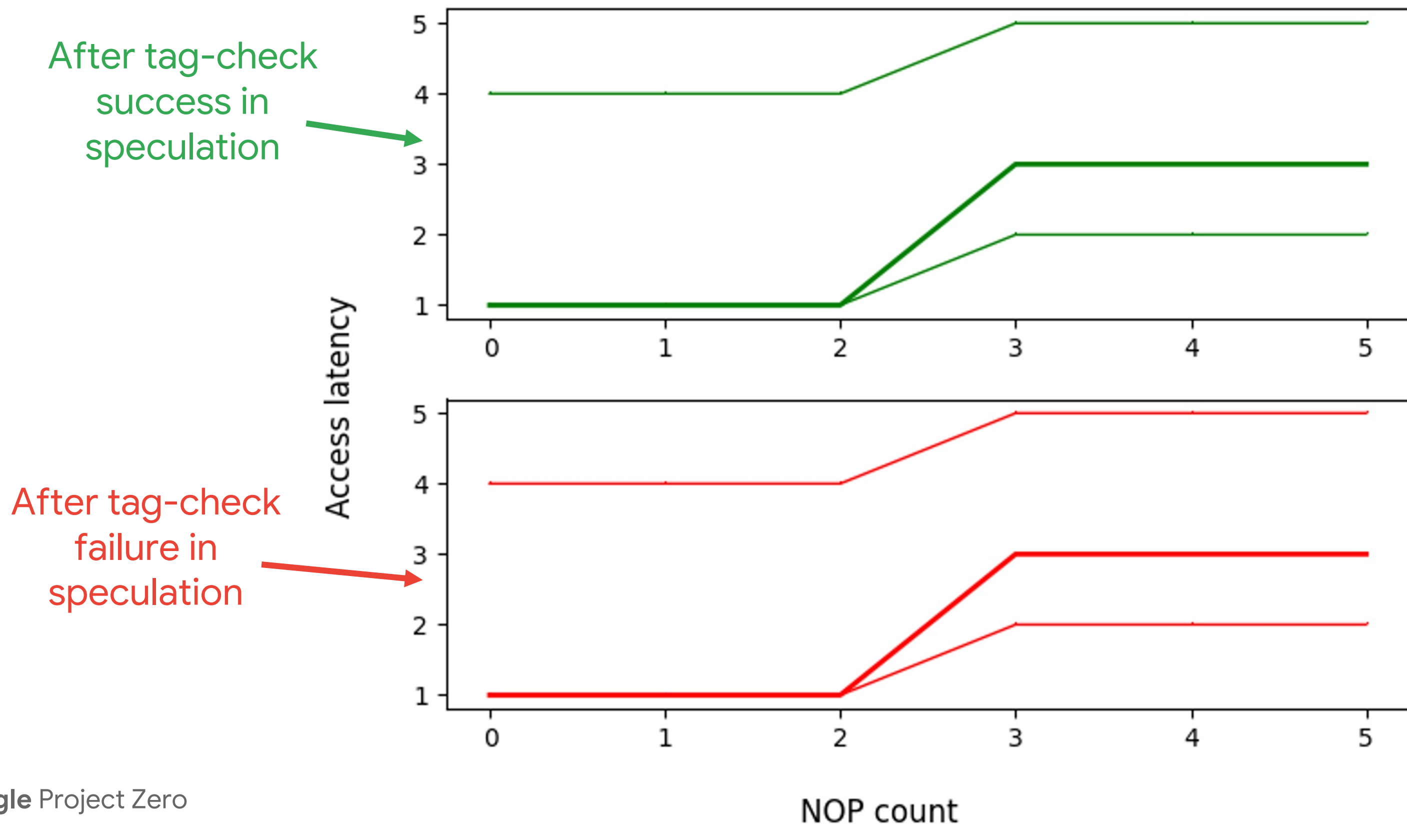
```
  ldr  x0, [x0]            ; this load is slow (*x0 is uncached)
  cbnz x0, speculation:    ; this branch is always taken during warmup
  ret

speculation:
  ldr  x1, [x1]            ; this load is fast (*x1 is cached)
                           ; the tag-check success or fail will happen on
                           ; this access, but during warmup the tag-check
                           ; will always be a success.

  orr  x2, x2, x1          ; this is a no-op (as x1 is always 0) but it
  ... n times ...          ; maintains a data dependency between the
  orr  x2, x2, x1          ; loads (and the no-ops), hopefully preventing
                           ; too much re-ordering.

  ldr  x2, [x2]            ; *x2 is uncached, if it is cached later then
                           ; this instruction was (probably) executed.

  ret
```

```
  ldr  x0, [x0]            ; this load is slow (*x0 is uncached)
  cbnz x0, speculation:    ; this branch is always taken during warmup
  ret

speculation:
  ldr  x1, [x1]            ; this load is fast (*x1 is cached)
                           ; the tag-check success or fail will happen on
                           ; this access, but during warmup the tag-check
                           ; will always be a success.

  orr  x2, x2, x1          ; this is a no-op (as x1 is always 0) but it
  ... n times ...          ; maintains a data dependency between the
  orr  x2, x2, x1          ; loads (and the no-ops), hopefully preventing
                           ; too much re-ordering.

  ldr  x2, [x2]            ; *x2 is uncached, if it is cached later then
                           ; this instruction was (probably) executed.

  ret
```

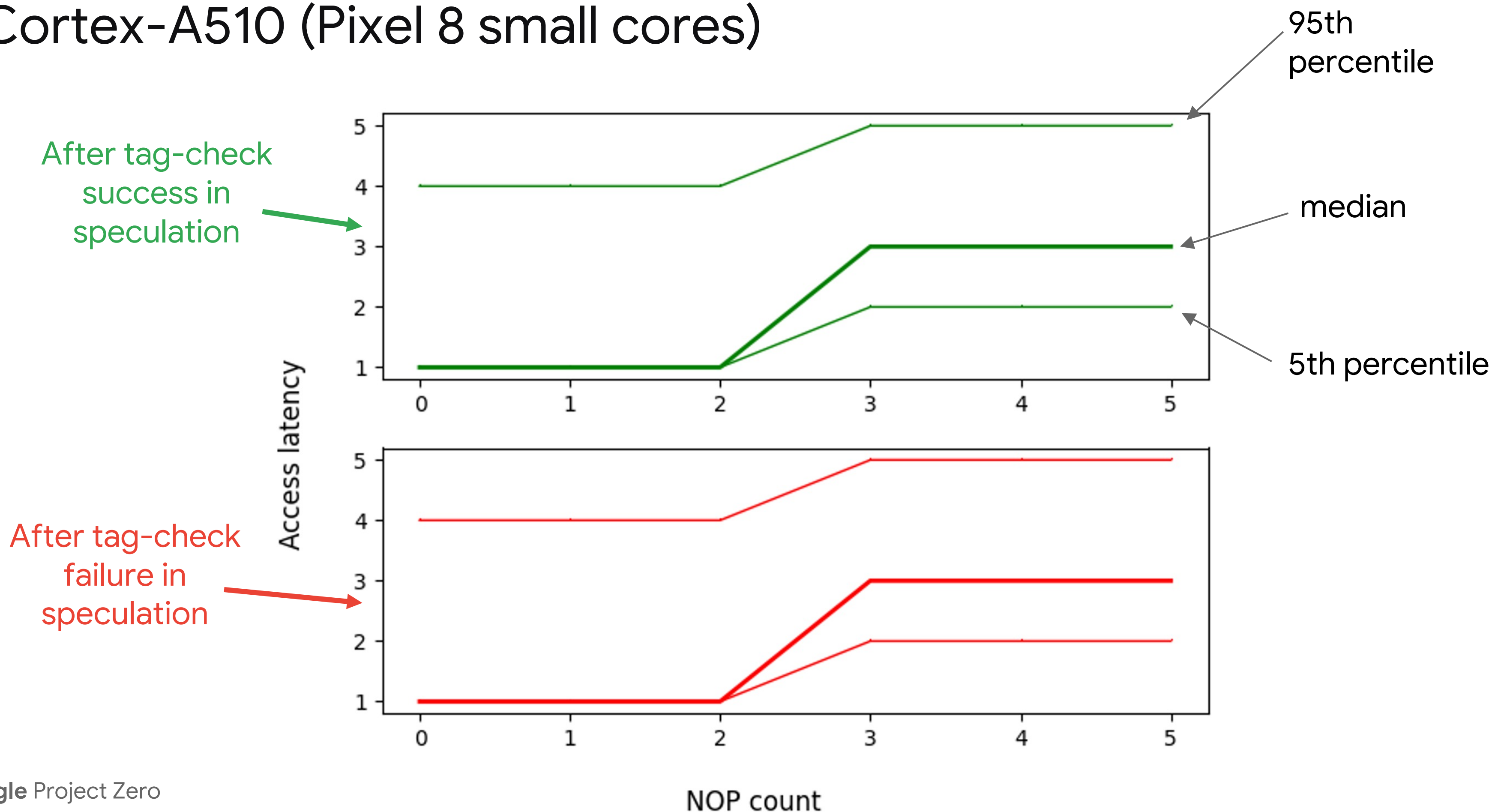
```
  ldr  x0, [x0]          ; this load is slow (*x0 is uncached)
  cbnz x0, speculation:  ; this branch is always taken during warmup
  ret

speculation:
  ldr  x1, [x1]          ; this load is fast (*x1 is cached)
                         ; the tag-check success or fail will happen on
                         ; this access, but during warmup the tag-check
                         ; will always be a success.

  orr  x2, x2, x1        ; this is a no-op (as x1 is always 0) but it
  ... n times ...        ; maintains a data dependency between the
  orr  x2, x2, x1        ; loads (and the no-ops), hopefully preventing
                         ; too much re-ordering.

  ldr  x2, [x2]          ; *x2 is uncached, if it is cached later then
                         ; this instruction was (probably) executed.

  ret
```

```
  ldr  x0, [x0]          ; this load is slow (*x0 is uncached)
  cbnz x0, speculation:  ; this branch is always taken during warmup
  ret

speculation:
  ldr  x1, [x1]          ; this load is fast (*x1 is cached)
                         ; the tag-check success or fail will happen on
                         ; this access, but during warmup the tag-check
                         ; will always be a success.

  orr  x2, x2, x1        ; this is a no-op (as x1 is always 0) but it
  ... n times ...        ; maintains a data dependency between the
  orr  x2, x2, x1        ; loads (and the no-ops), hopefully preventing
                         ; too much re-ordering.

  ldr  x2, [x2]          ; *x2 is uncached, if it is cached later then
                         ; this instruction was (probably) executed.

  ret
```

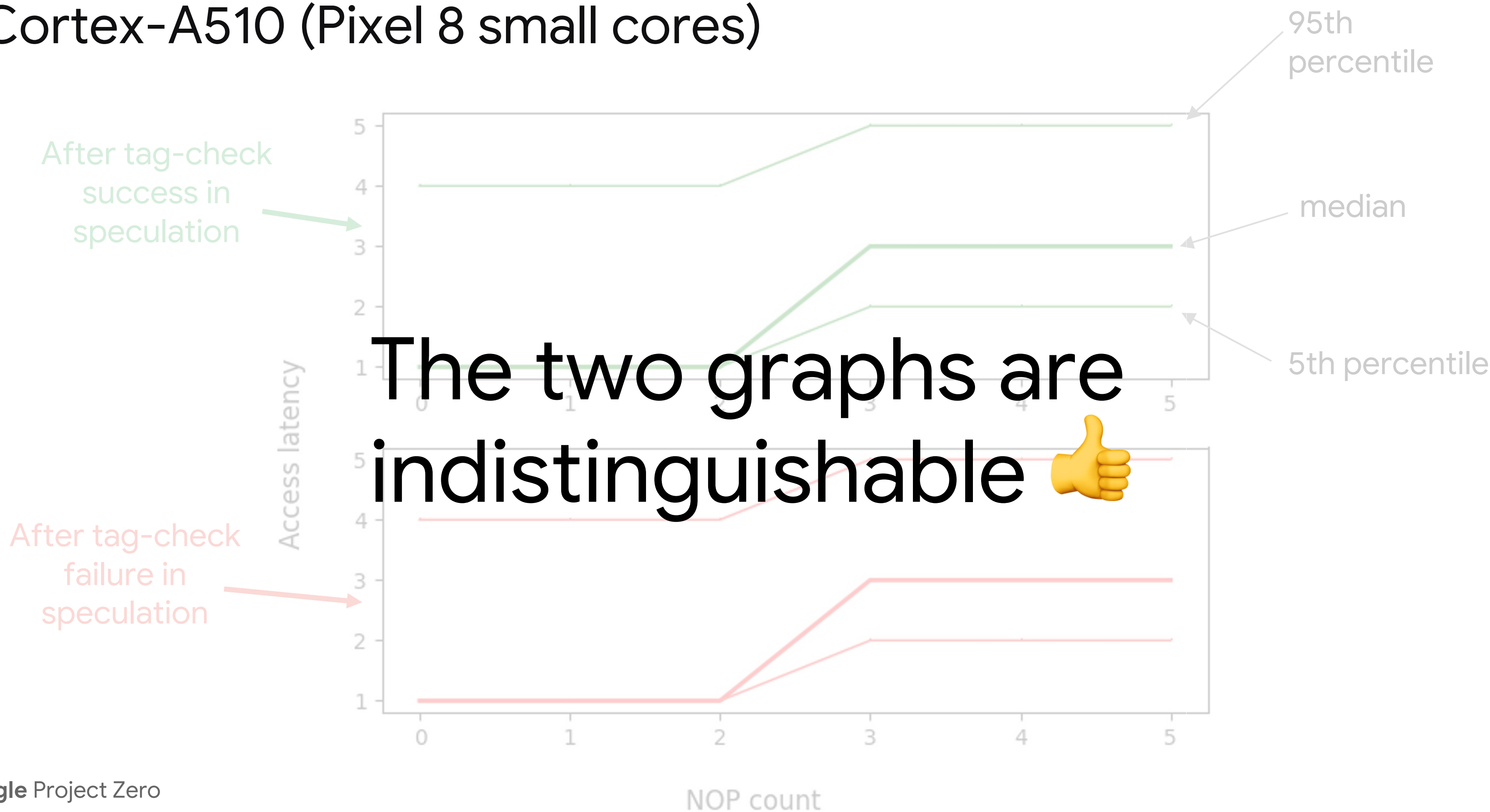
Cortex-A510 (Pixel 8 small cores)



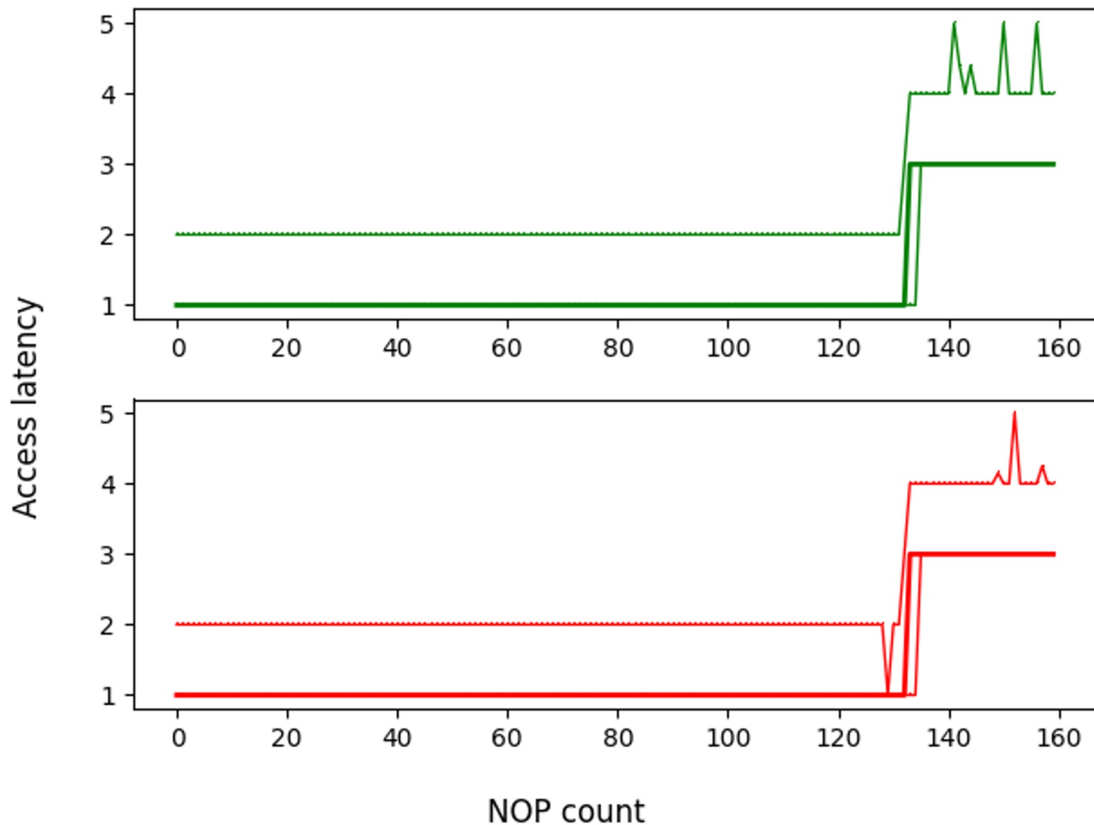
Cortex-A510 (Pixel 8 small cores)



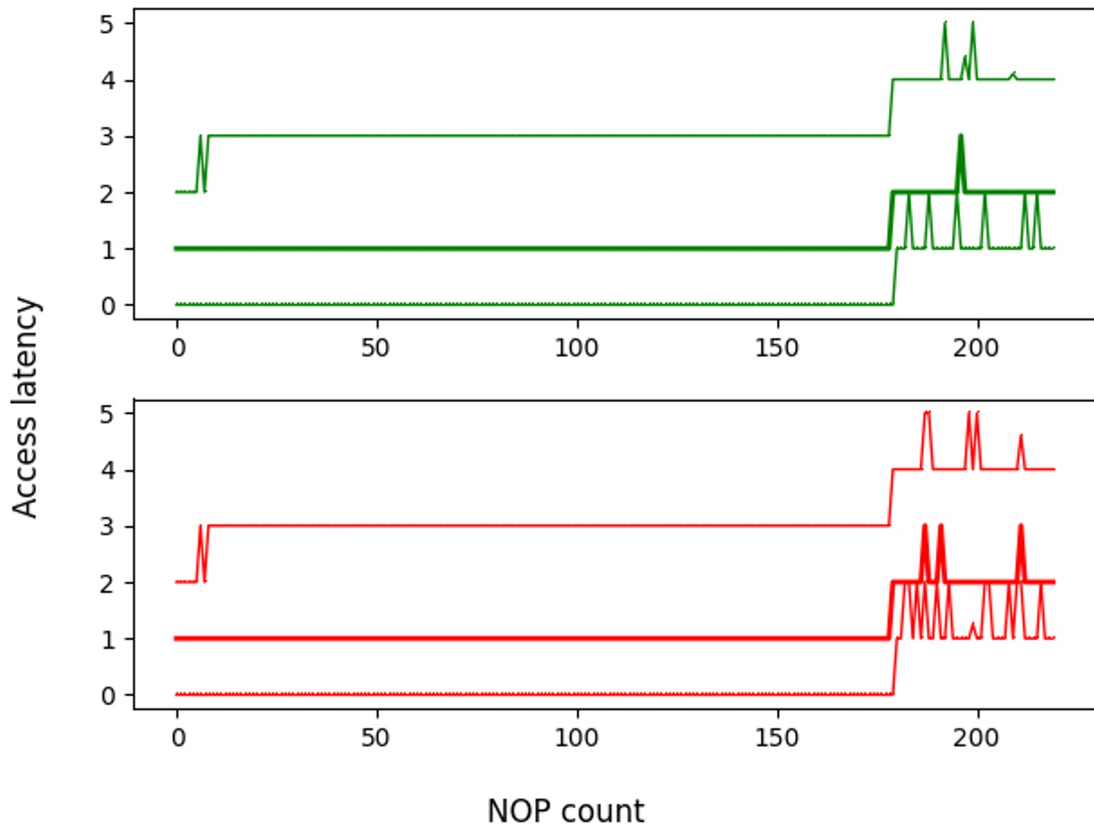
Cortex-A510 (Pixel 8 small cores)



Cortex-A715 (Pixel 8 middle cores)



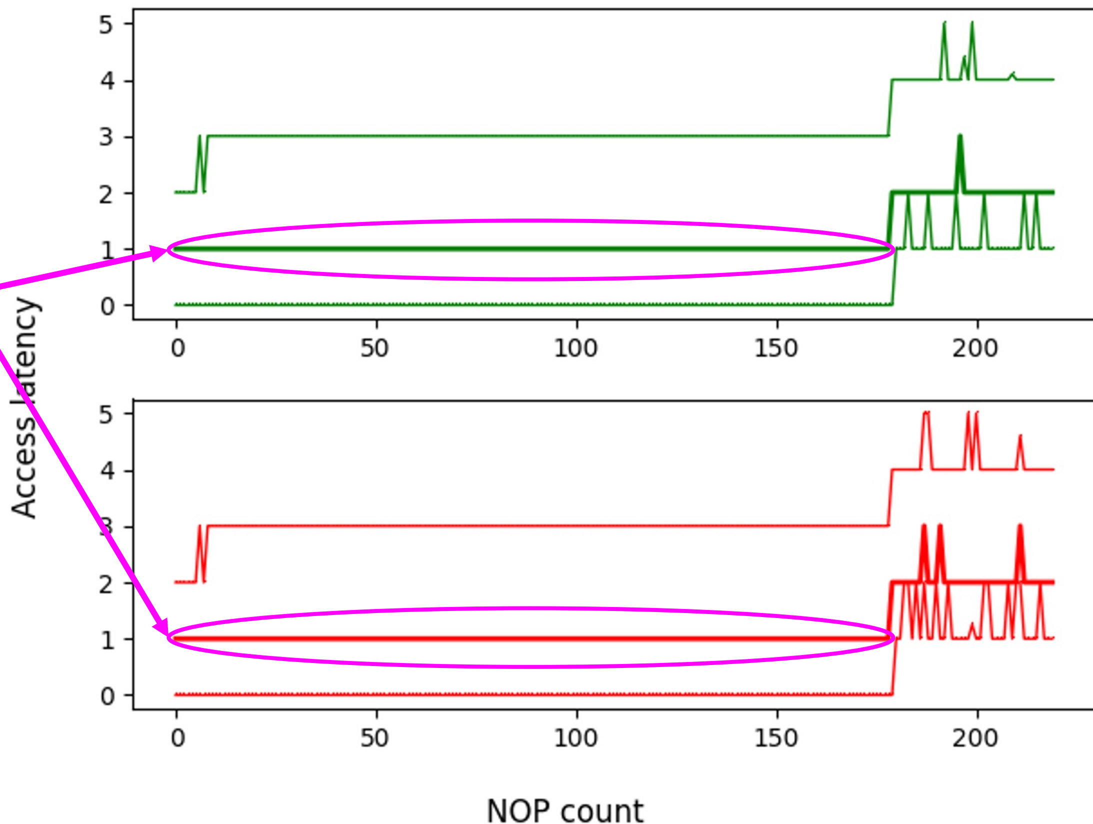
Cortex-X3 (Pixel 8 large core)



The limits of measurement

The virtual timer on the biggest core has really, really poor resolution.

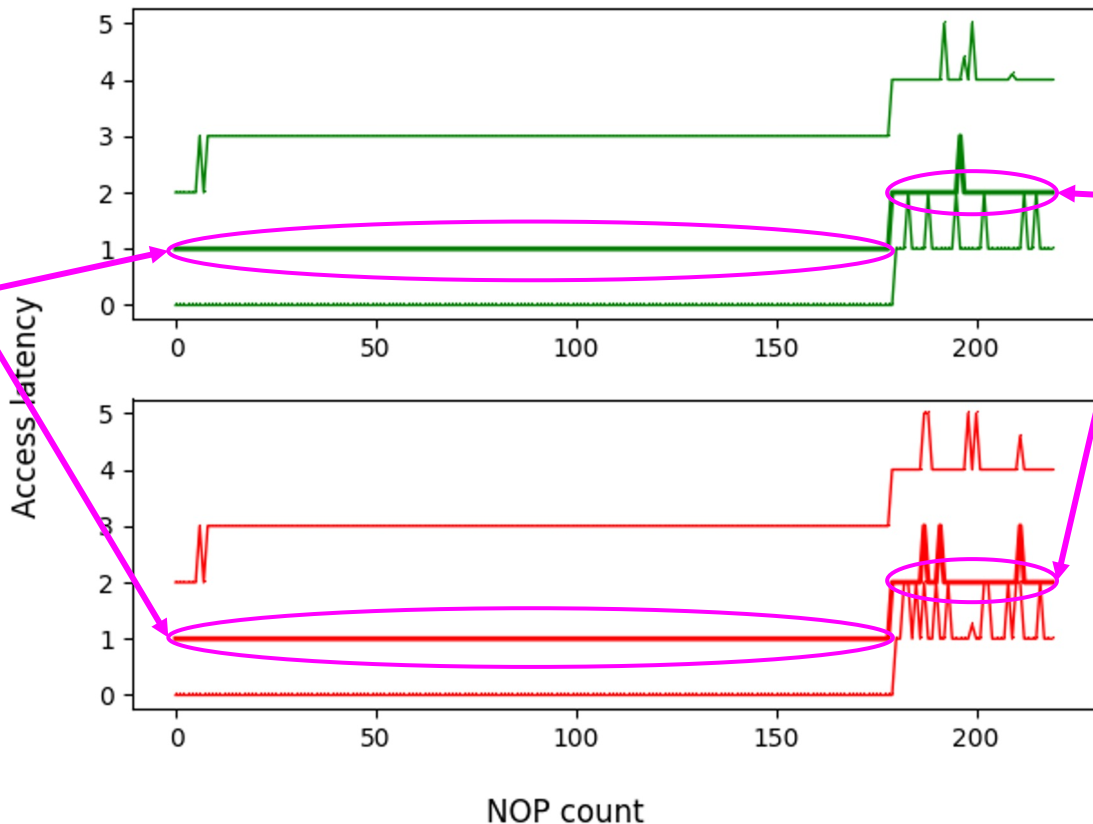
A hot (l1) cache hit has a median access latency of '1'



The limits of measurement

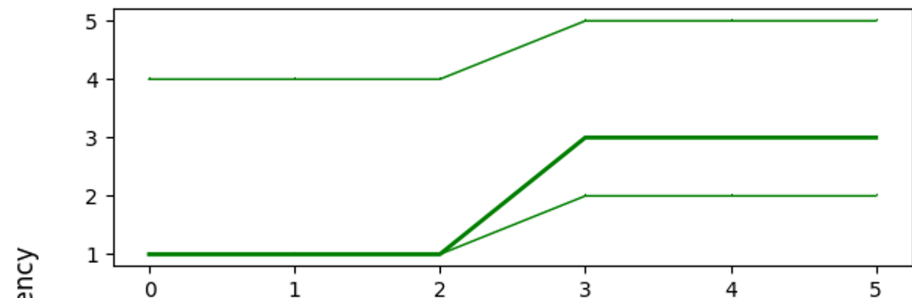
The virtual timer on the biggest core has really, really poor resolution.

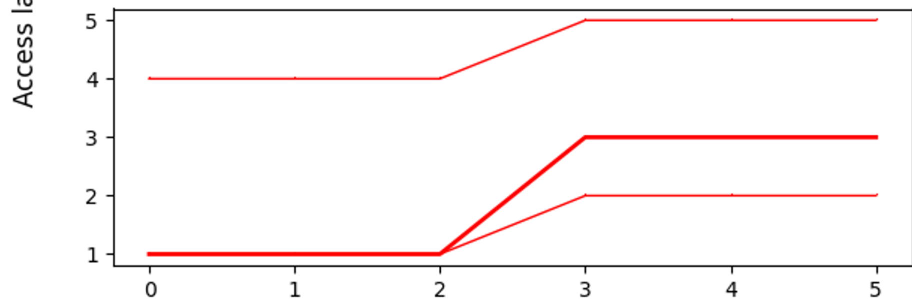
A hot (l1) cache hit has a median access latency of '1'



A cold (llc) cache miss has a median access latency of '2'!

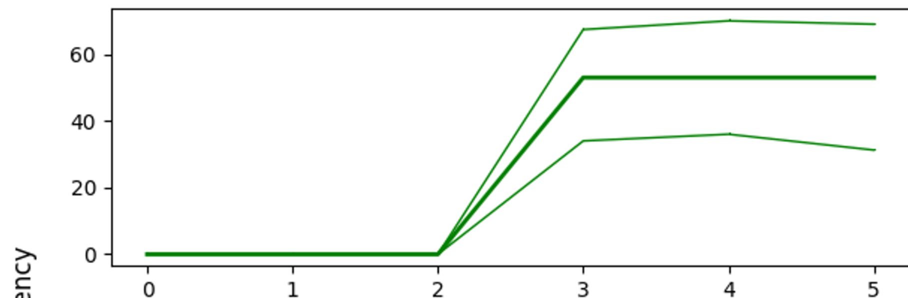
Is a shared memory timer better?

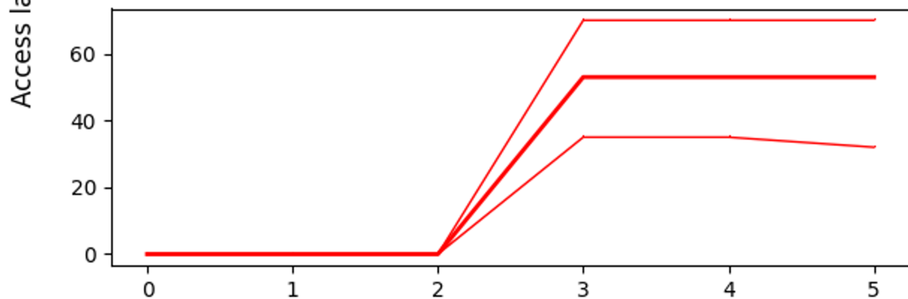




Access latency

NOP count

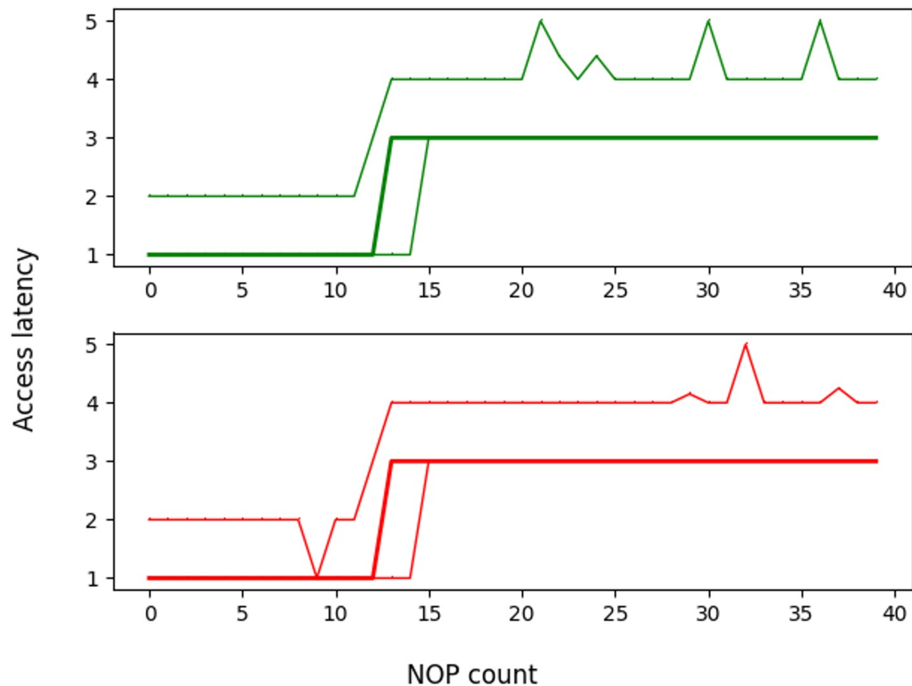


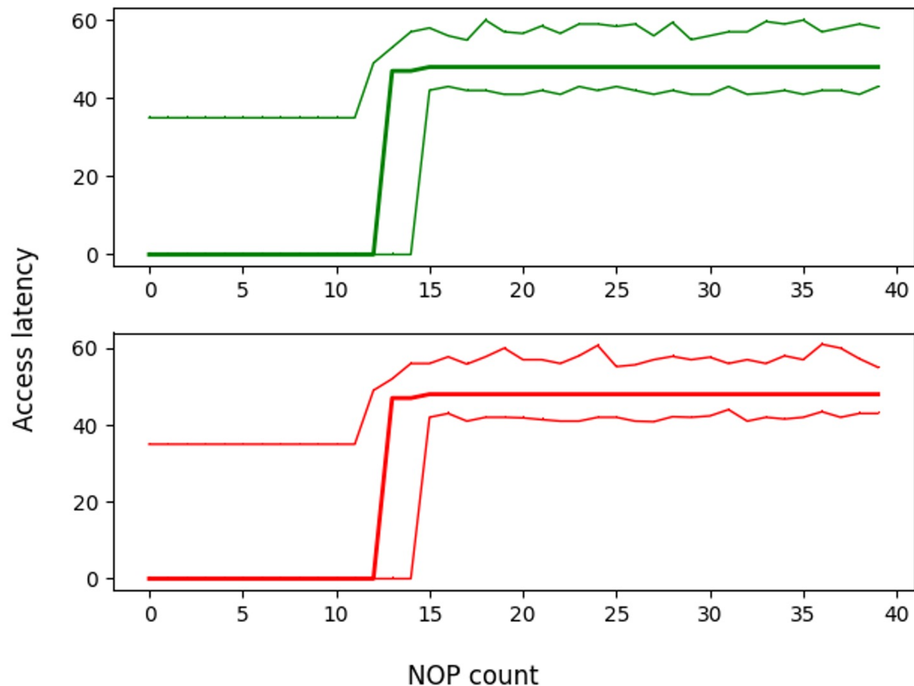


Access latency

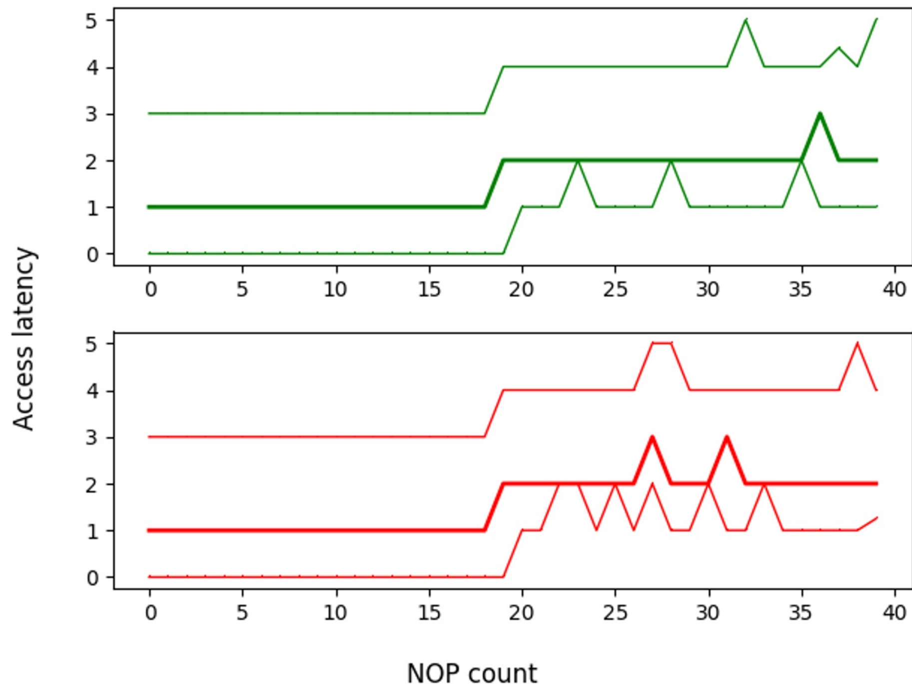
NOP count

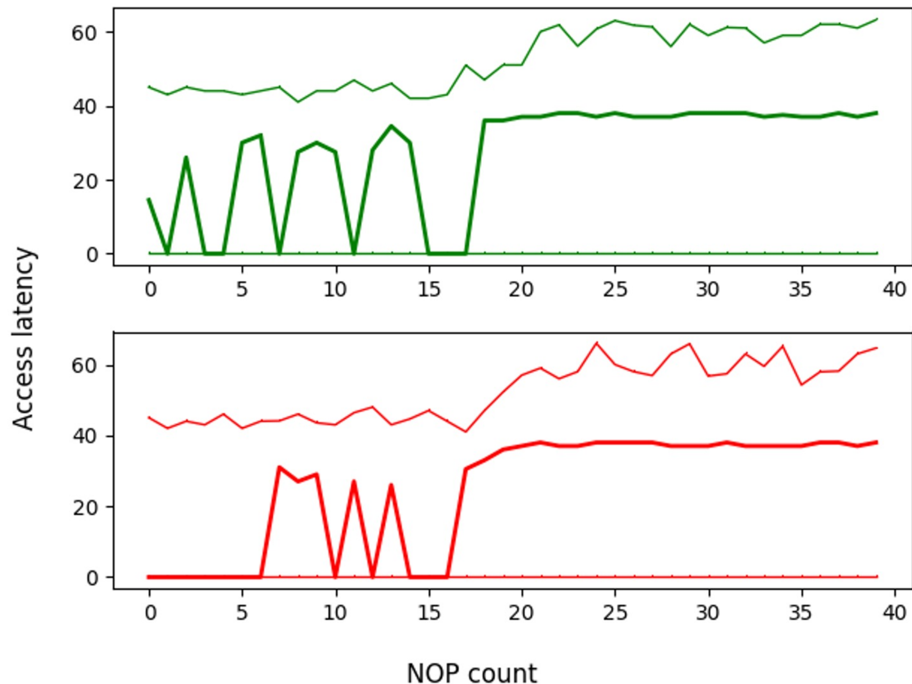
Is a shared memory timer better?



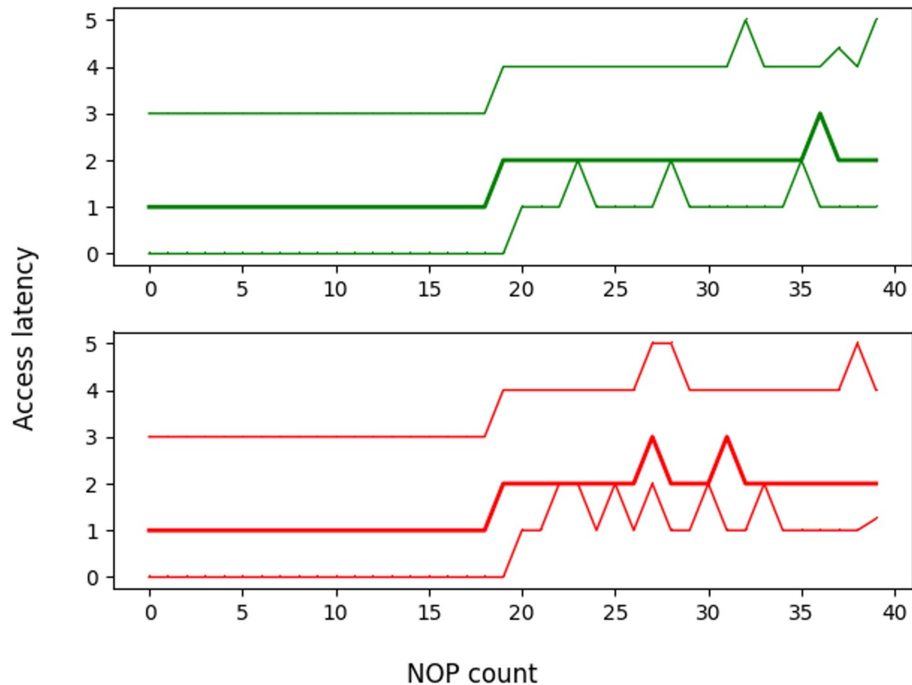


Is a shared memory timer better?

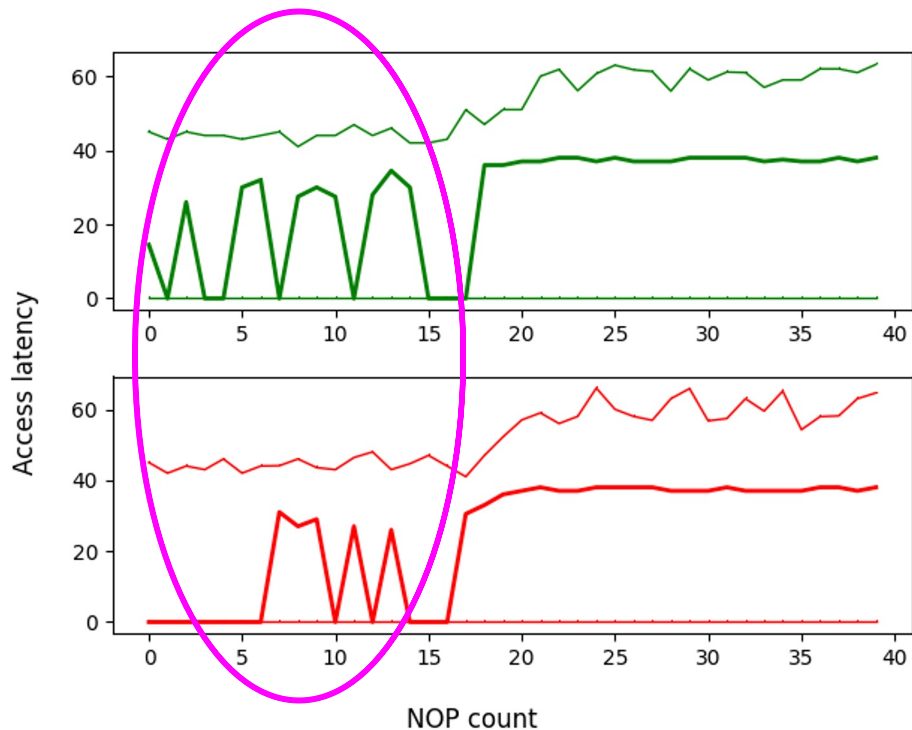




Is a shared memory timer better?



WHAT?!?



Realisation

- Measurements on the fastest core are noisy enough that the signal/noise ratio is poor.

- Repeated experiments consistently show (inconsistent) results that **appear** to differentiate tag-check pass and failure.

- Small-scale experiments and eyeballing graphs is not enough to be confident about the behaviour of the fastest core.

- **Proving a negative…**

Unknown-tag attacks (Mostly ASYNC)

Revisiting signal-safety...

```
// This function runs in a compromised context: see the top of the
file.
// Runs on the crashing thread.
// static
void ExceptionHandler::SignalHandler(int sig, siginfo_t* info, void*
uc) {
  // Give the first chance handler a chance to recover from this
signal
  if (g_first_chance_handler_ != nullptr &&
      g_first_chance_handler_(sig, info, uc)) {
    return;
  }
```

Demo: Initial exploit

```
shiba:/ $ cd /data/local/tmp
shiba:/data/local/tmp $
```

Classic exploit code

```
// XXX: This is a load-bearing string concatenation.
var do_not_remove = "A".concat("B");
```

Demo: Initial exploit reliability

```
markbrand@markbrand:~/poc2023$
```

Demo: Optimized exploit reliability

```
markbrand@markbrand:~/poc2023$ python3 ./reliability.py ./poc_demo_6.js
```

Context	Mode	Bypass techniques	
		known-tag-bypass	unknown-tag-bypass
Chrome: Renderer Exploit	async	Trivial ♻️	Likely trivial ♻️
	sync	Trivial ♻️	Bypass techniques should be rare 🛠️
Chrome: IPC Sandbox Escape	async	Likely possible in many cases ♻️	Likely possible in many cases 🐛 *
	sync	Likely possible in many cases ♻️	Bypass techniques should be rare 🛠️
Android: Binder Sandbox Escape	async	Difficulty will depend on service	Difficulty will depend on service 🐛 *
	sync	Difficulty will depend on service	Bypass techniques should be rare 🛠️
Android: Messaging App Oneshot	async	Likely impossible in most cases	Good enough bugs will be very rare 🐛 *
	sync	Likely impossible in most cases	Bypass techniques should be rare 🛠️

Questions?