# Architectural Metadata for Memory Safety

Dr. Nathaniel “nwf” Filardo

2024 Dec 4

## Who am I?

- Former CMU physics then CS undergrad, 410 student & TA, 213 instructor, ...
  - You can blame me for `swexn()`
- Contractor for SCI Semi, previously postdoc at Cambridge and researcher at Microsoft
  - I am not speaking on behalf of any employers.  Opinions herein are mine.
  - None of this should be taken to be information about product plans.
- I prefer talks with interrupts enabled; please ask questions as they arise

## Outline

- Software security (or: “how are buffer overflows *still* a thing?”)
- Pointer *authentication* (ARMv8.3, ~2017)
- Pointer *coloring* (ARMv9 MTE, ~2023)
  - Newly on market: Google Pixel 8 (“Tensor G3” CPU), October 2023
- *Upgrading* pointers (CHERI; commercial availability in 6 months to ~5 years)
- Safe languages on safe architectures

## Learning Goals

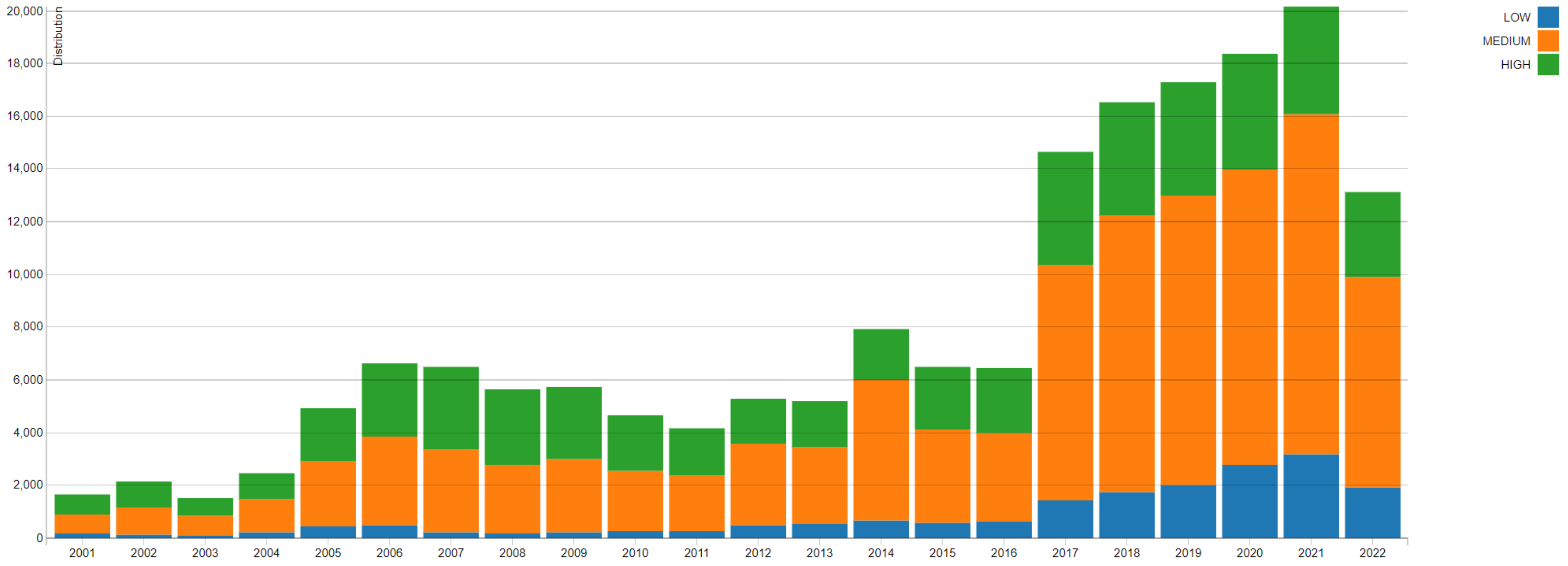
- Memory safety (esp. spatial, temporal)
- Metadata in pointers: authentication & coloring
- Memory capability

# Modern Computer Architecture: Unsafe at Any Speed?

Ralph Nader. *Unsafe at Any Speed.* (1965)    Kamp. *Linear Address Spaces: Unsafe at Any Speed.* (2022)

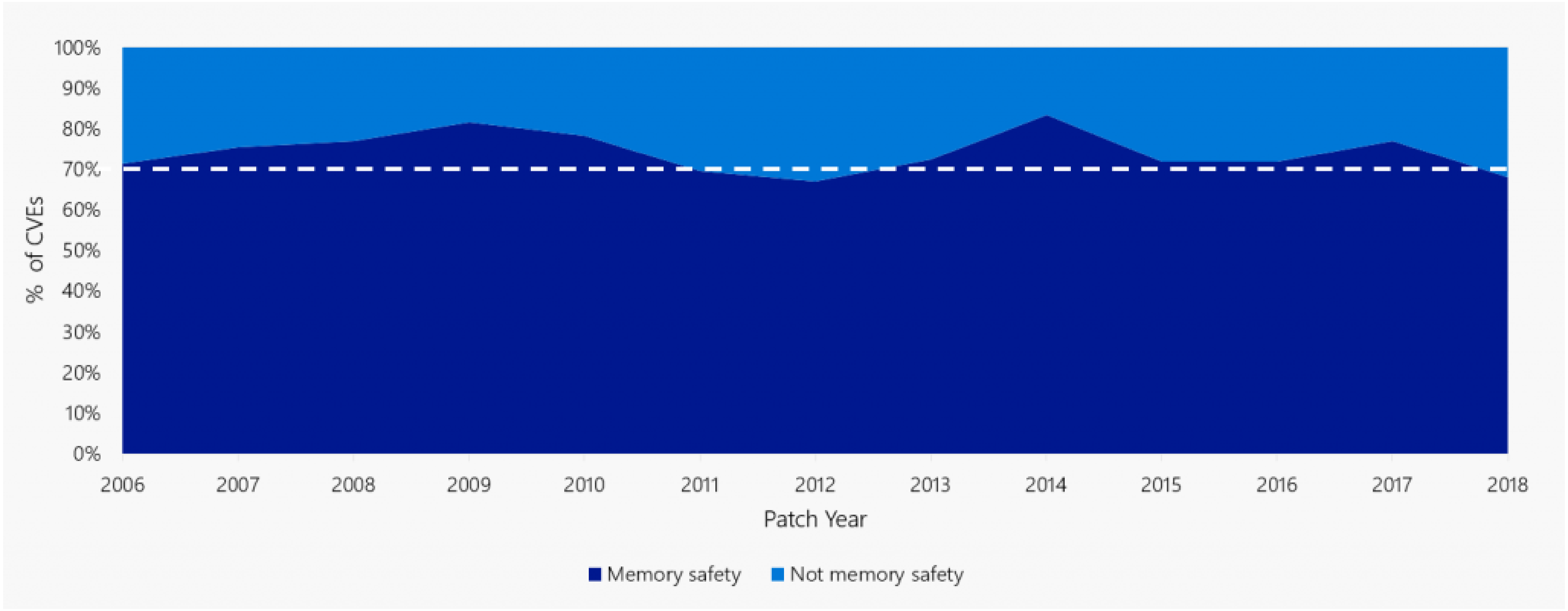
## CVEs and High Severity Bugs from (Lack of) Memory Safety

CVSS Severity Count Over Time (as of 22 Jul 2022)



NVD - CVSS Severity Distribution Over Time (nist.gov)

## CVEs and High Severity Bugs from (Lack of) Memory Safety



## Modern Architecture Unsafety

Very Short, Not At All Comprehensive, Examples

## Misbehaving C Program: Spatial & Referential Safety Violations

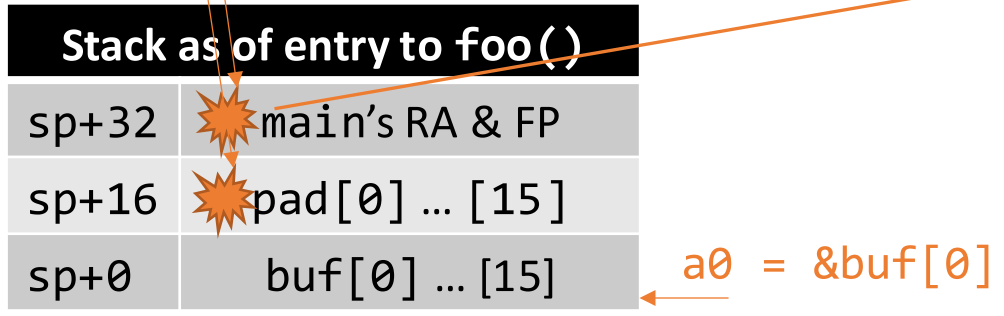
```c
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64

```
foo:
  mov     w8, #65
  strb    w8, [x0, #16]
  strb    w8, [x0, #32]
  ret
main:
  sub     sp, sp, #48
  stp     x29, x30, [sp, #32]
  add     x29, sp, #32
  mov     x0, sp

  bl      foo

  ldp     x29, x30, [sp, #32]
  mov     w0, wzr
  add     sp, sp, #48
  ret     // x30
```

Stores relative to address in x0

x0 holds *address* of buf on stack



Several things go wrong:

1. Write outside of allocation (lack of *spatial safety*)
2. *Corrupt* saved return address (lack of *integrity*)
3. Jump to corrupted address when `main()` "returns" (lack of *referential safety*)

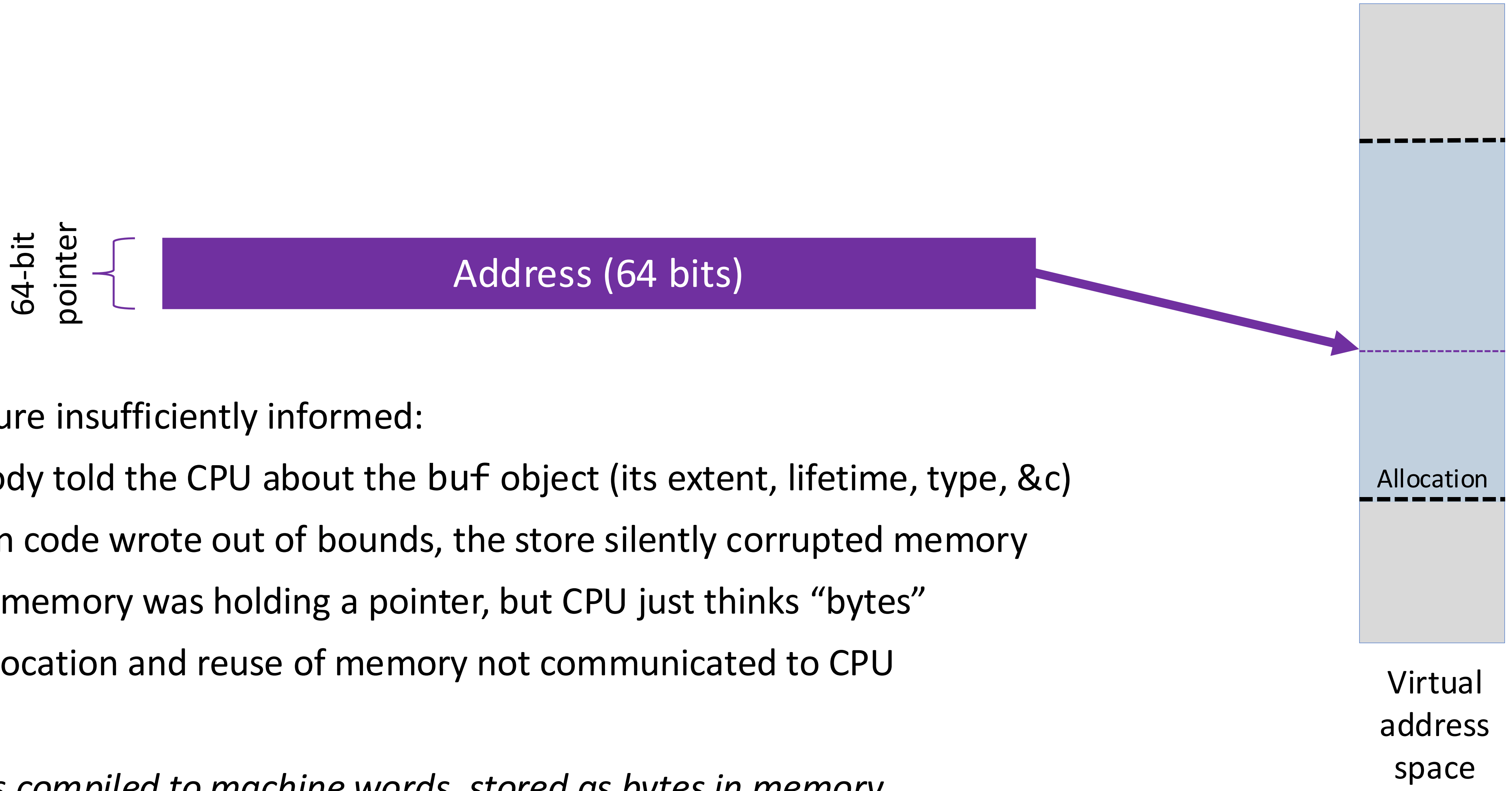
## Temporal Safety

- Memory is there to be (re)used
  - C language and compiler reuses stack memory aggressively by design
  - Heap allocator reuses freed objects for new ones

- What about use-after-free?

```
char *p = malloc(1024); // say: p == 0x15410DE0U
free(p);
char *q = malloc(1024); // quite likely: q == 0x15410DE0U

strcpy(p, “oh no”); // p == q, but different objects!
```

## Architecture Enables Safety Violations



Architecture insufficiently informed:

1. Nobody told the CPU about the buf object (its extent, lifetime, type, &c)
2. When code wrote out of bounds, the store silently corrupted memory
3. That memory was holding a pointer, but CPU just thinks “bytes”
4. Deallocation and reuse of memory not communicated to CPU

*C pointers compiled to machine words, stored as bytes in memory*.

## OK, But That’s Just C!

- Rewrite the world in a *safe language*!
  - LISP, Scheme, Rust, Java, JavaScript, ML, Ur/Web, Haskell…
  - Different data representations, operational semantics, static type systems…

- Safe?
  - Array index errors throw exceptions; other spatial errors impossible*
  - Temporal errors impossible*

* Some assumptions apply; see next slide

## What About All The Stuff We Can't Rewrite?

- A staggeringly large amount of software *already exists*.
  - OpenHub.net estimates ~10B LoC of C, ~3B LoC of C++ just in the open world.
    - That probably works out to $130G - $1.3T to rewrite everything.

- A lot of effort in optimizing that software! FFI bridges for the stuff we like?
    - Hand-tuned, specialized implementations... like `xz`!
    - Correctness can be subverted by foreign code!

- Language correctness often depends on (huge) runtime systems!
  - Written in C (or something like it)!
    - By humans!

## What Have We Tried Doing?

- Lots of people have tried lots of things:
  - Software tricks: stack canaries, guard pages, ASLR, W^X, fat pointers, …
  - Static analyses: symbolic execution, fuzzing, …
  - Languages: Ada, ML, Haskell, Java, JavaScript, C#/.Net, Rust, …
  - Computers: System/36, iAPX 432/BiiN, …
  - Architectural edits: BTI, continual excavation below ring 0, …

## What Now?

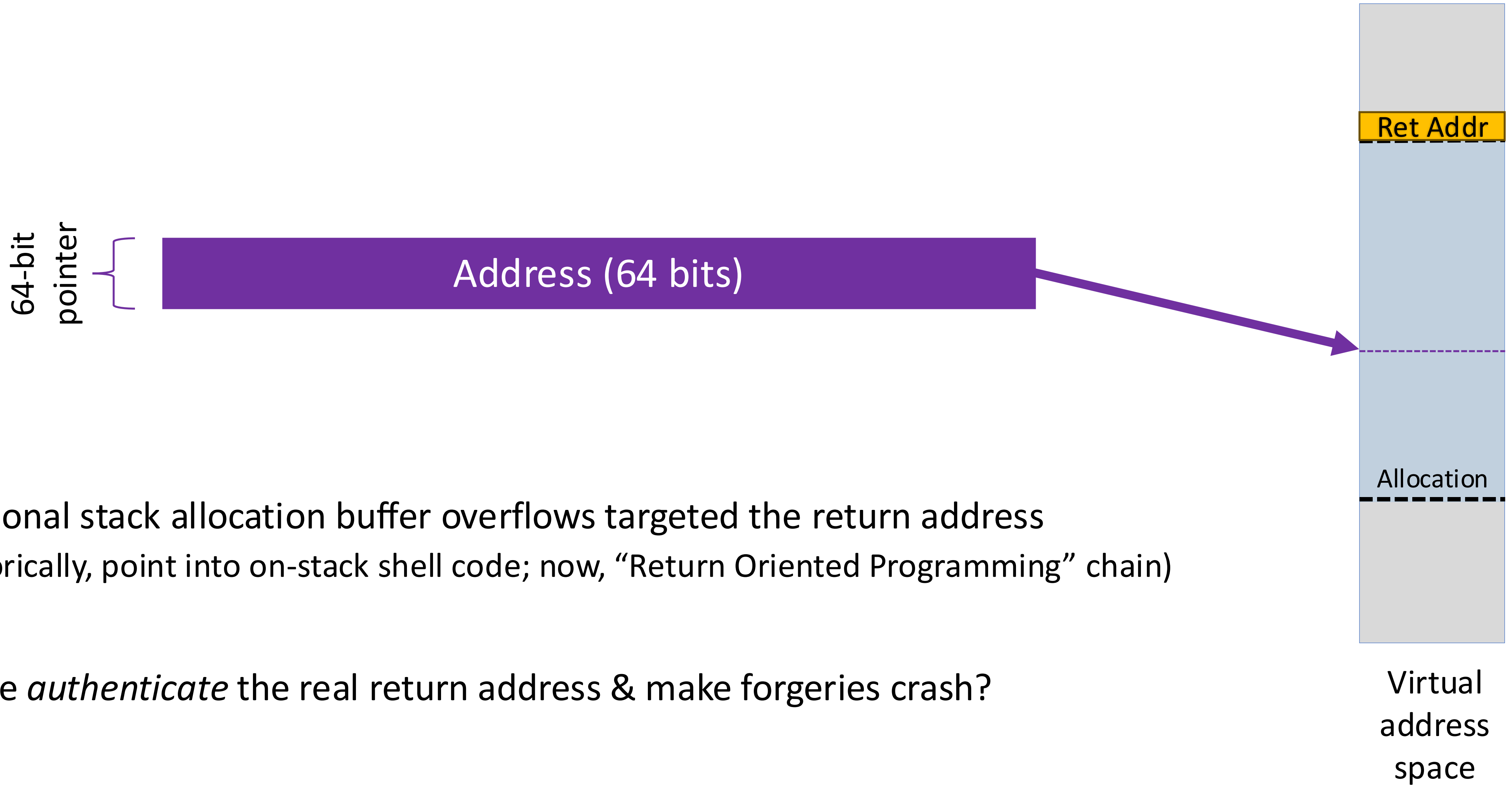
Increasingly popular “new old thing” is to add *metadata* to existing architecture:

- Arm “**Pointer Authentication Code**” (PAC) integrity checks (commercialized ~2017)
  - Make it harder to “forge” pointers / easier to detect forgeries
- Arm “**Memory Tagging Extension**” (MTE) “lock and key” covariance (~2023)
  - Make it harder to access memory *out of bounds* or *after free*
- **CHERI** memory *capability system* (2025?)
  - Deterministic memory safety and *software compartmentalization*

## Arm’s Pointer Authentication

Embedding Cryptographic Signatures

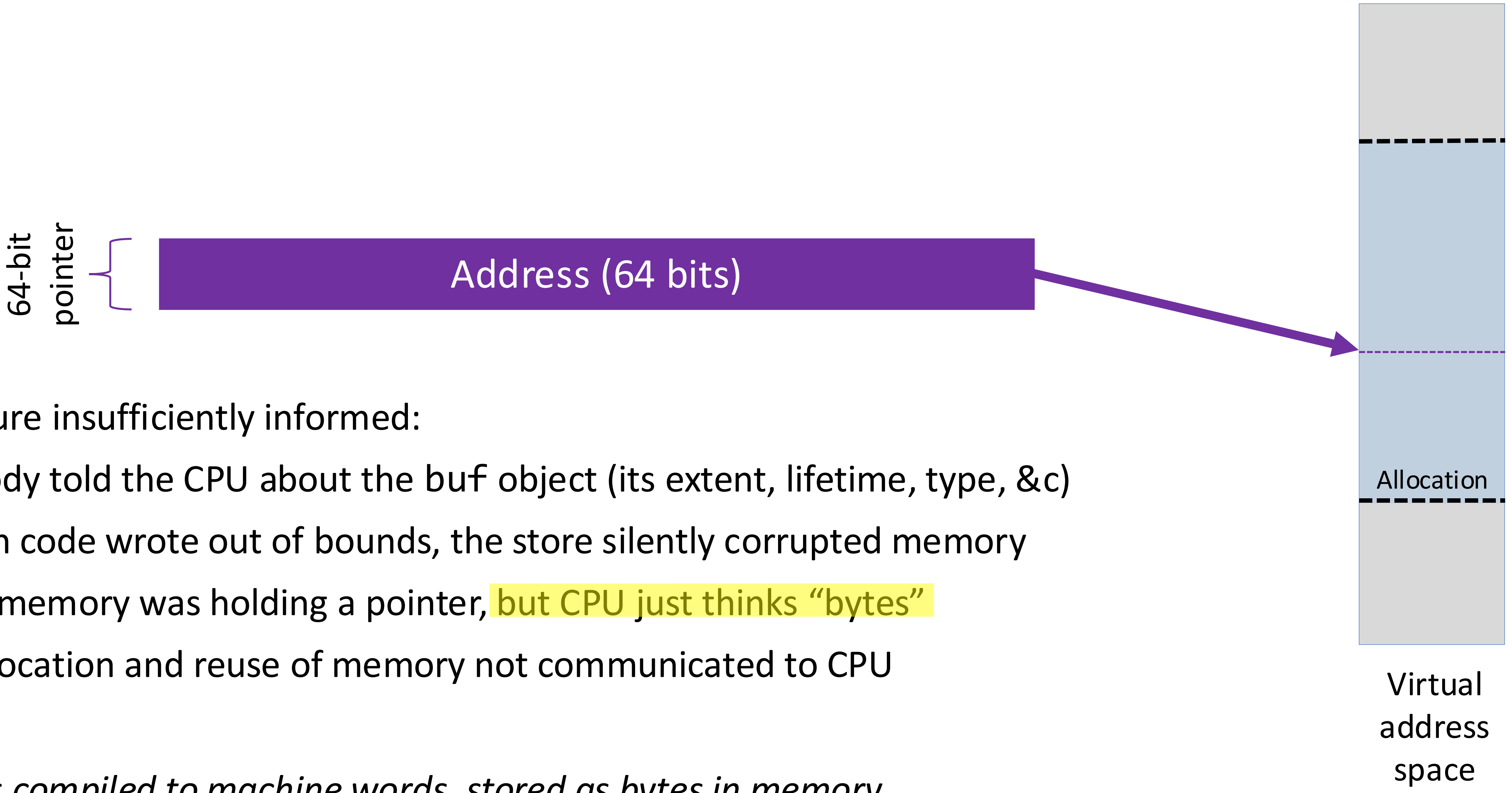
## Recall: Architecture Enables Safety Violations



- Traditional stack allocation buffer overflows targeted the return address
  (Historically, point into on-stack shell code; now, “Return Oriented Programming” chain)

- Can we *authenticate* the real return address & make forgeries crash?

## Architecture Enables Safety Violations

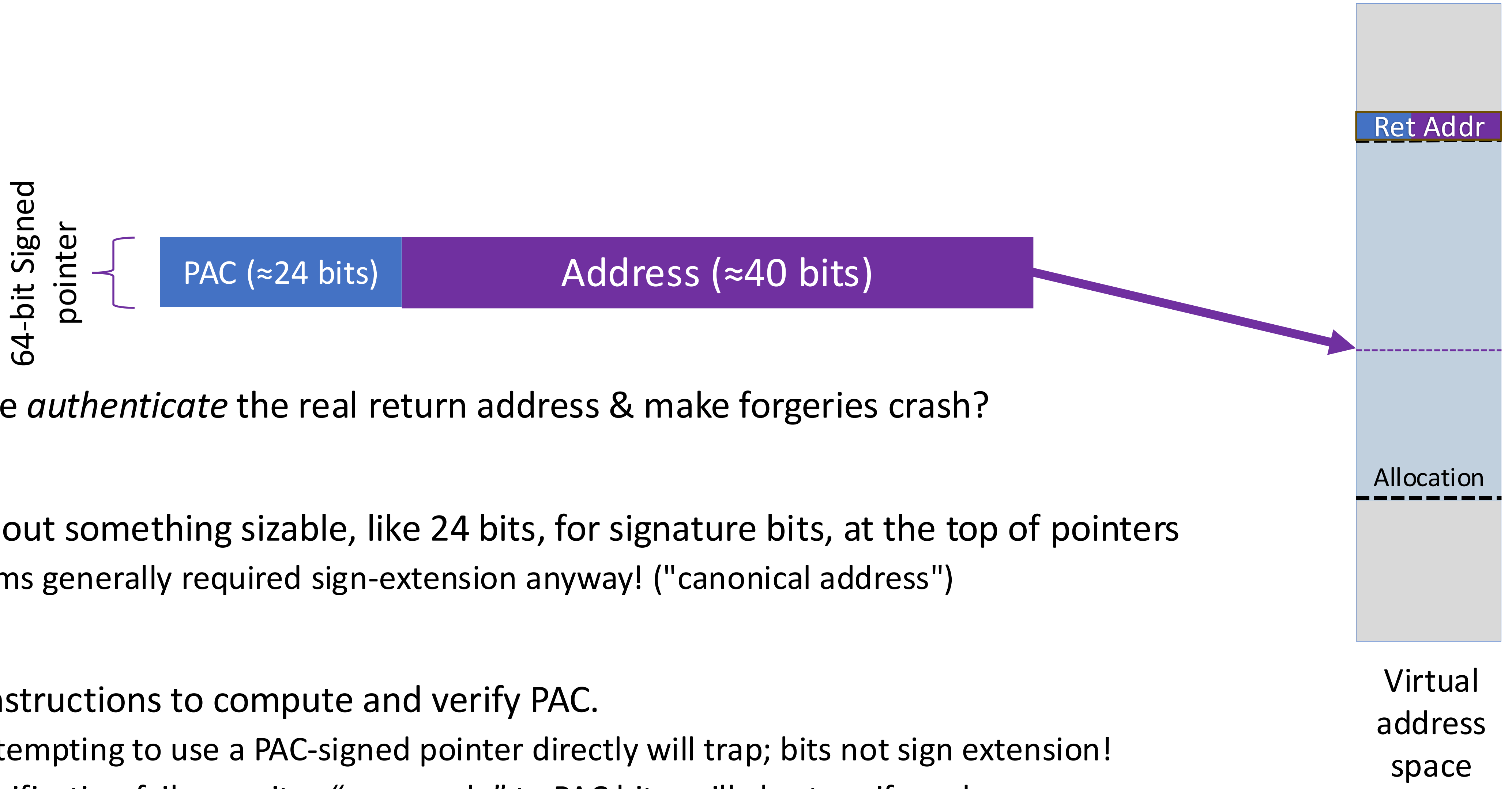


Architecture insufficiently informed:

1. Nobody told the CPU about the buf object (its extent, lifetime, type, &c)
2. When code wrote out of bounds, the store silently corrupted memory
3. That memory was holding a pointer, but CPU just thinks "bytes"
4. Deallocation and reuse of memory not communicated to CPU

*C pointers compiled to machine words, stored as bytes in memory.*

## PAC-ing Extra Bits



- Can we *authenticate* the real return address & make forgeries crash?

- Carve out something sizable, like 24 bits, for signature bits, at the top of pointers
  Systems generally required sign-extension anyway! ("canonical address")

- Add instructions to compute and verify PAC.
  - Attempting to use a PAC-signed pointer directly will trap; bits not sign extension!
  - Verification failure writes "error code" to PAC bits; will also trap if used.

Arm - Understand Arm Pointer Authentication    Qualcomm - Pointer Authentication on ARMv8.3

## What to PAC?

- Cryptographically combine:
  1. The pointer’s value
  2. A secret value (from a kernel-managed control register)
  3. A “context” word (TBD)

- Cryptography?  Secrets?
  - Make it hard to “forge” pointers, *even if some have leaked*
  - More than one secret: sw sign for different purposes (stack pointer, function pointer, data pointer, …)

- Context?
  - Further differentiation of authentication tag, without requiring more and more secrets
  - “Not just *any* return address, the one *right here* on the stack.”
  - “Not just *any* pointer, but one that points to type 0x15410DE0U”

## Spilled Return Address: Without PAC

```c
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64

```
foo:
  mov     w8, #65
  strb    w8, [x0, #16]
  strb    w8, [x0, #32]
  ret
main:

  sub     sp, sp, #48
  stp     x29, x30, [sp, #32]
  add     x29, sp, #32
  mov     x0, sp
  bl      foo
  ldp     x29, x30, [sp, #32]
  mov     w0, wzr
  add     sp, sp, #48

  ret     // x30
```

Spill and restore "link register" x30

| Stack as of entry to foo() | |
|---|---|
| sp+32 | main's saved RA |
| sp+16 | pad[0] … [15] |
| sp+0 | buf[0] … [15] |

Canonical Pointer

0x0000_0000_0010_CAFE

a0 = &buf[0]

## Spilled Return Address: With PAC

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64

```
foo:
  mov     w8, #65
  strb    w8, [x0, #16]
  strb    w8, [x0, #32]
  ret
main:
  pacia   x30, sp
  sub     sp, sp, #48
  stp     x29, x30, [sp, #32]
  add     x29, sp, #32
  mov     x0, sp
  bl      foo
  ldp     x29, x30, [sp, #32]
  mov     w0, wzr
  add     sp, sp, #48
  autia   x30, sp
  ret     // x30
```

Sign link register w/ stack pointer

Spill and restore "link register" x30

Verify signature on link register

| Stack as of entry to foo() | |
|---|---|
| sp+32 | main's PAC'd RA |
| sp+16 | pad[0] … [15] |
| sp+0 | buf[0] … [15] |

PAC'd Pointer

`0xE7A3_F000_0010_CA`

`a0 = &buf[0]`

autia sets x30 to (say) `0xDEAD_0000_0010_CAFE`.
ret faults on noncanonical value; hooray!

## PAC Deployment

PAC “needs to get everywhere”: potentially *every* creation and use of a pointer!  How?

Staged deployment strategy:

1. Recompile binaries with a subset of pointer signing
    - Instructions are cleverly encoded as “no-op hints” on old machines
2. Make kernel changes to turn on feature for binaries requesting it
    - Recompiled binaries get more secure
3. For new software targeting new CPUs, can use more pointer signing features
    - Easier for some (Apple, Android) than others (Microsoft, mainstream Linux, *BSD)

## PAC Summary

- Increasingly deployed in practice, especially in Apple’s ecosystem
- Easy to take first steps
- Generally effective in its niche

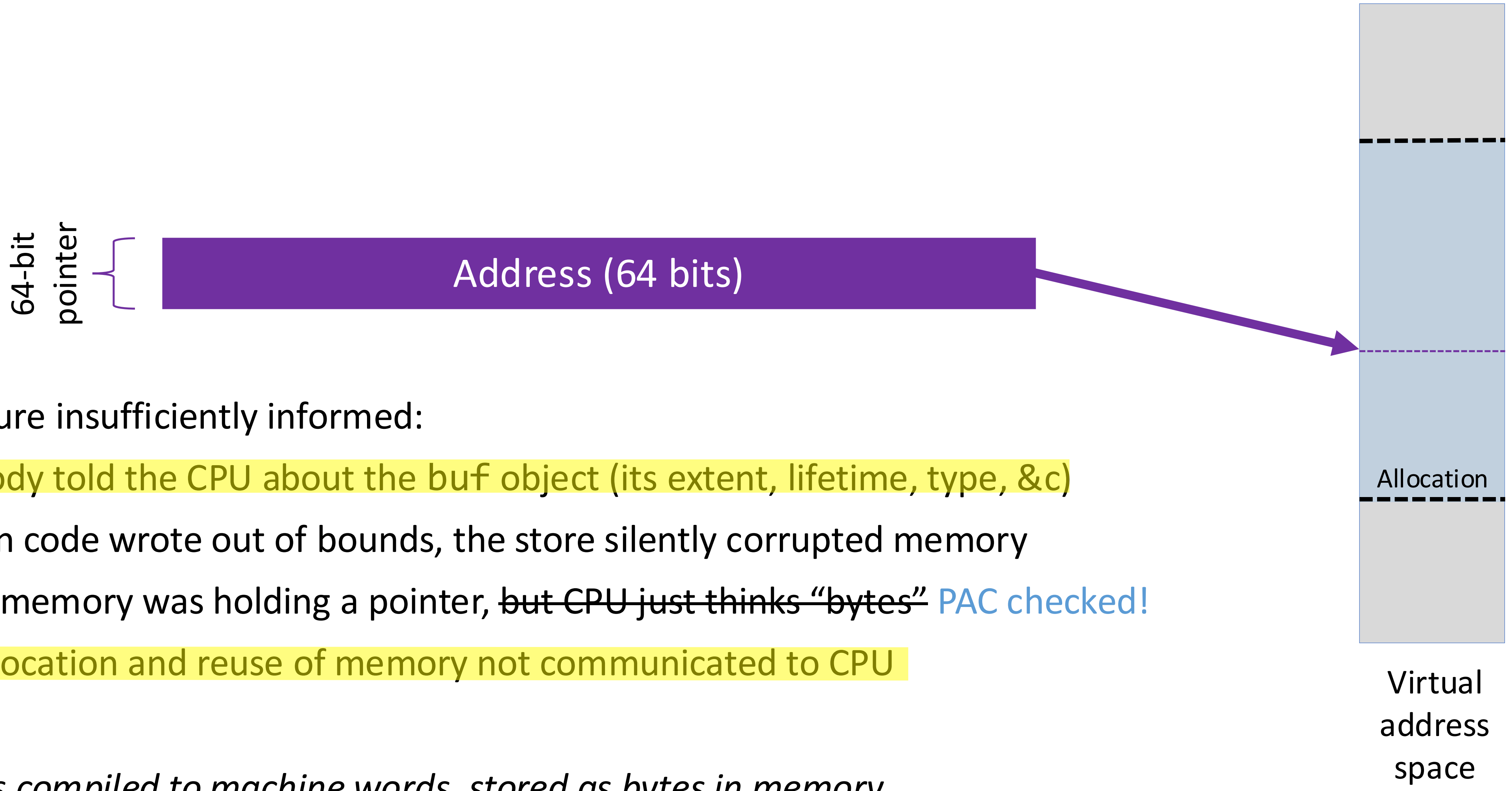
- Bypasses do still happen:
  - If attacker can repeatedly try a guess at a forged pointer, $2^{24}$ is not a lot of guesses.
  - If attacker has cross-context access to a “signing gadget”, may not need to guess

Google Project Zero - Examining Pointer Authentication on the iPhone XS (2019)

# Arm's Memory Tagging Extension

Coloring Pointers and Memory

## Architecture Enables Safety Violations

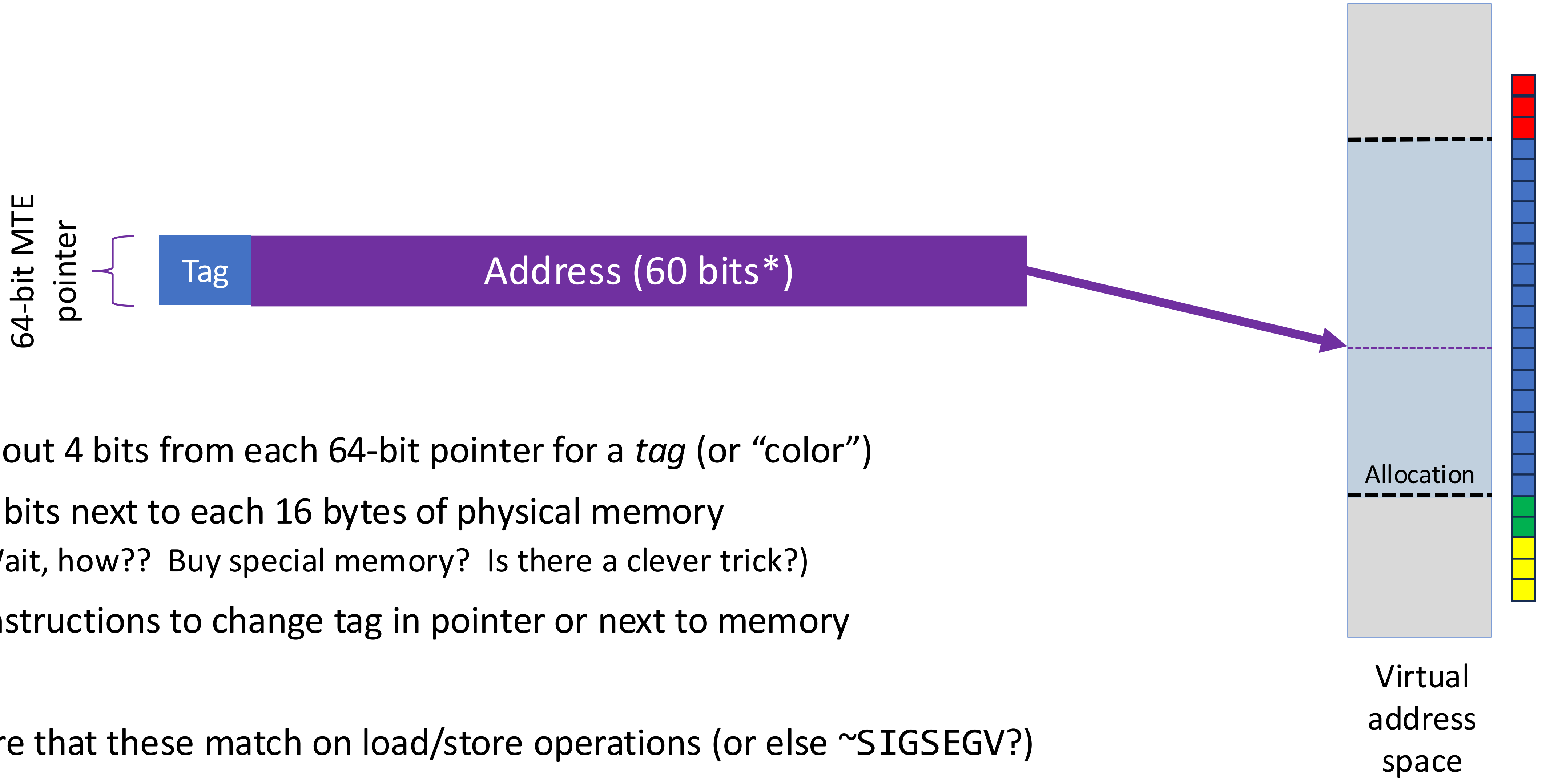


Architecture insufficiently informed:

1. Nobody told the CPU about the buf object (its extent, lifetime, type, &c)
2. When code wrote out of bounds, the store silently corrupted memory
3. That memory was holding a pointer, ~~but CPU just thinks "bytes"~~ PAC checked!
4. Deallocation and reuse of memory not communicated to CPU

*C pointers compiled to machine words, stored as bytes in memory.*

## MTE Architecture

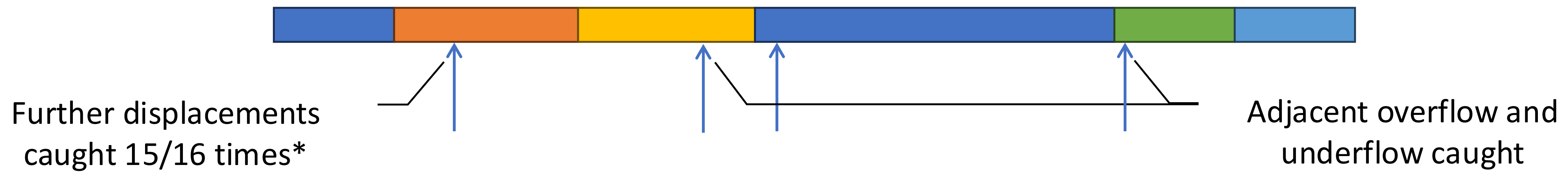


- Carve out 4 bits from each 64-bit pointer for a *tag* (or "color")
- Bolt 4 bits next to each 16 bytes of physical memory
  - (Wait, how?? Buy special memory? Is there a clever trick?)
- Add instructions to change tag in pointer or next to memory
- Require that these match on load/store operations (or else ~`SIGSEGV`?)

Arm - Delivering enhanced security through Memory Tagging Extension (2021) Arm MTE whitepaper

## MTE for Spatial Safety

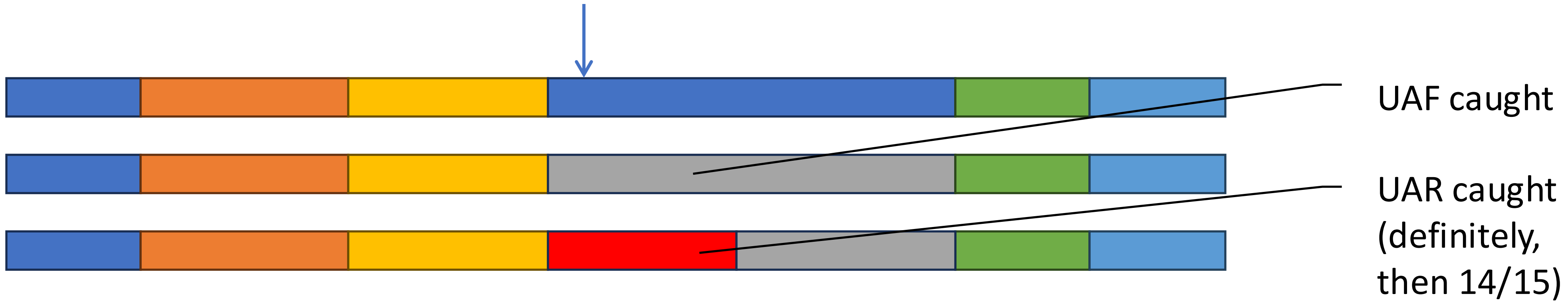
- Software (heap, compiler) can ensure that adjacent objects never the same color:


Further displacements caught 15/16 times*
Adjacent overflow and underflow caught

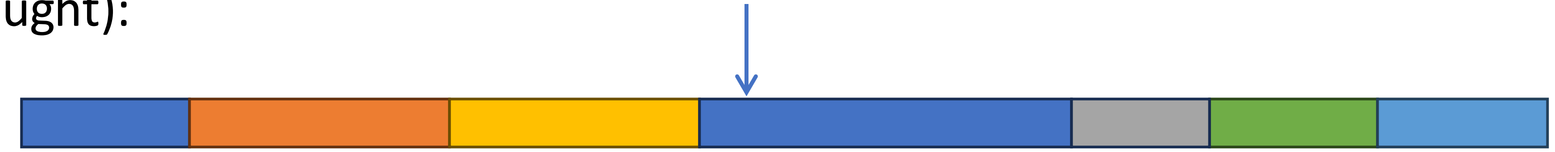
- Easy in `malloc`; some subtlety in stack handling; globals (`.data`) a little tricky

## MTE for Temporal Safety

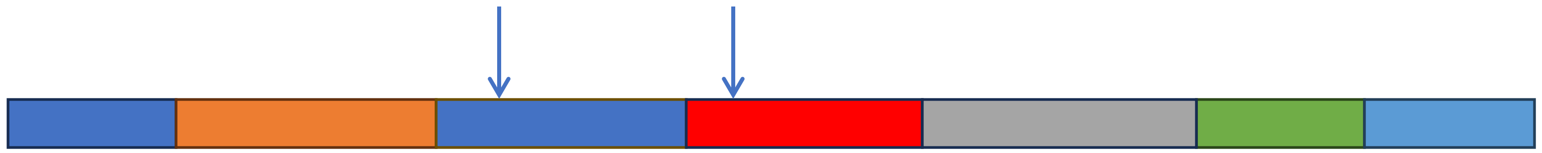
- Heap temporal safety: freed and (re)allocated objects' colors changed
  - Easiest to pick allocated object color (not neighbors!) at random; will *most likely* be a different color.
  - Might reserve one color for free objects & always exclude previous color



- Eventually, we'll run out of colors (pigeon-hole principle) and somewhere we'll have a collision (UAR not caught):



- Neighbors could also collide colors: wouldn't detect simultaneous UAF & OOB access:



## Misbehaving C Program: Spatial & Referential Safety Violations

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64

```
foo:
  mov     w8, #65
  strb    w8, [x0, #16]
  strb    w8, [x0, #32]
  ret
main:
  sub     sp, sp, #48
  stp     x29, x30, [sp, #32]
  add     x29, sp, #32
  mov     x0, sp

  bl      foo

  ldp     x29, x30, [sp, #32]
  mov     w0, wzr
  add     sp, sp, #48
  ret     // x30
```

Stores relative to address in x0

x0 holds *address* of buf on stack

| Stack as of entry to foo() | |
|---|---|
| sp+32 | main's saved RA |
| sp+16 | pad[0] … [15] |
| sp+0 | buf[0] … [15] |

a0 = &buf[0]

# Misbehaving C Program: Spatial Safety Violations, with Slideware MTE

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64 + MTE

```
foo:
  mov     w8, #65
  strb    w8, [x0, #16]
  strb    w8, [x0, #32]
  ret
main:
  sub     sp, sp, #48
  stp     x29, x30, [sp, #32]
  add     x29, sp, #32
  irg     x0, sp
  stg     x0, [x0]
  bl      foo
  stg     sp, [sp]
  ldp     x29, x30, [sp, #32]
  mov     w0, wzr
  add     sp, sp, #48
  ret     // x30
```

Stores unchanged

Copy sp to x0, *insert random tag*

Set tag in memory

Put "sp" tag back

Mismatch reported *here*: Tag in x0 != tag at [x0, #16]

| Stack as of entry to `foo()` | | Tag |
|---|---|---|
| sp+32 | main's saved RA | 1 |
| sp+16 | pad[0] … [15] | 1 |
| sp+0 | buf[0] … [15] | 7 |

`x0 = &buf[0], tag 7`

## MTE Enforcement

MTE has three enforcement strategies, trading security for performance:

- **Synchronous**: each load and store will check tags before committing, will trap (SIGSEGV) on mismatch.
- **Symmetric asynchronous**: loads or stores commit regardless of tags, mismatches set a flag
  - Kernel expected to check flag and kill process on each entry (syscall, trap, or interrupt).
- **Asymmetric asynchronous**: loads synchronous, stores asynchronous.
  - Synchronous loads “easy” to do fast: data coming from cache/RAM anyway.
  - Synchronous stores slow: performance needs stores to complete without loading cache line.

Intended deployment scenarios look like “accelerated debugging”:

1. At scale, in production:
   a. an async mode to answer, “is there a bug?”;
   b. once “yes”, switch to sync
2. Under fuzzing, in sync mode.

## MTE Weaknesses

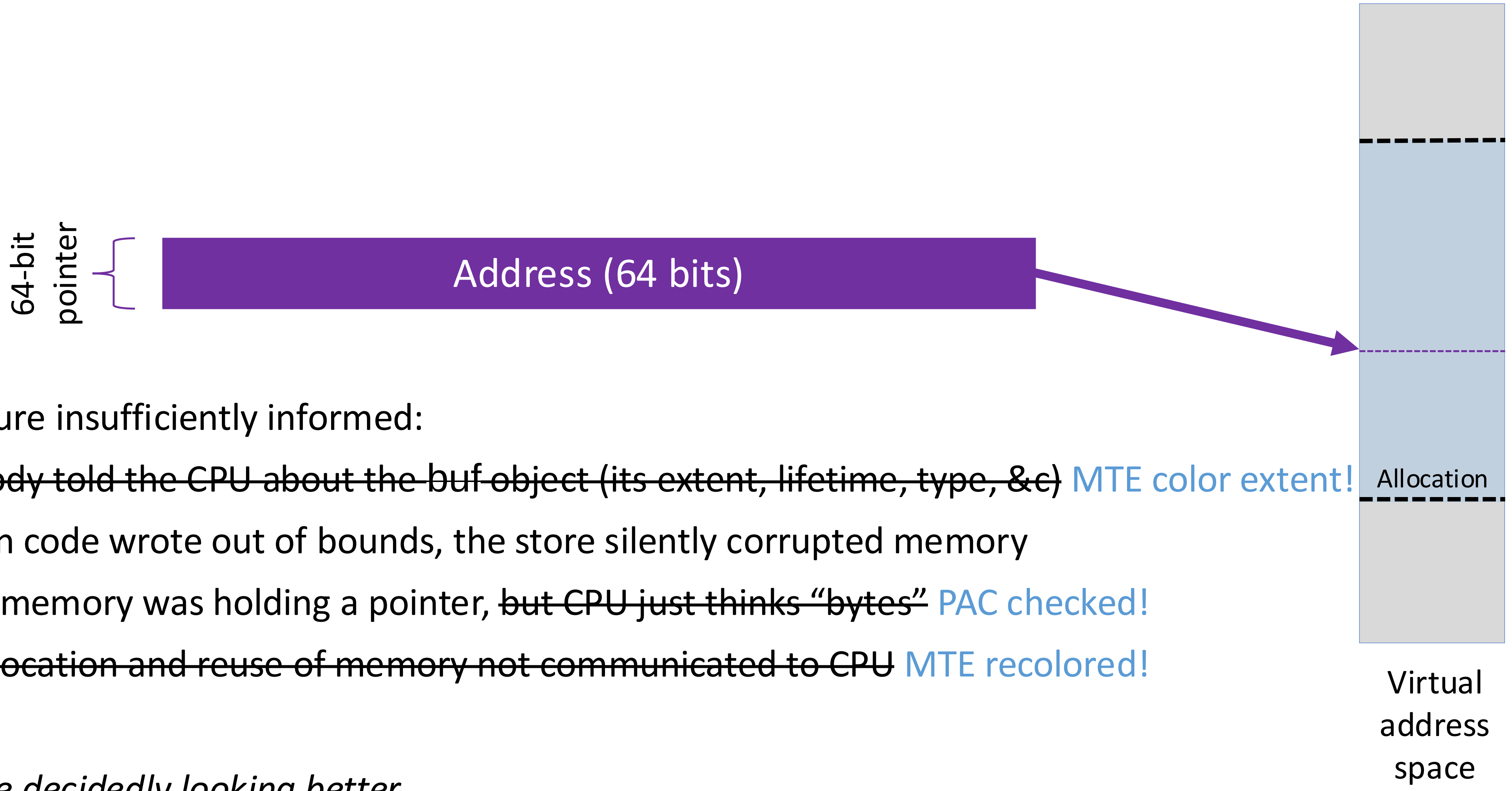
- Kernel access to user memory is generally not (at present) mediated by MTE.
- Probabilistic arguments (“15/16”) fall if the attacker can *forge tagged pointers* of the right color.
- Opinions vary, but: MTE is not generally considered viable defense against determined attackers.

# MTE Summary

- In shipping arm cores!
- At-scale audit & debug
- High probability of finding *bugs*

- High cost of synchronous mode
- Weak against directed *attacks*

## Architecture Enables Safety Violations



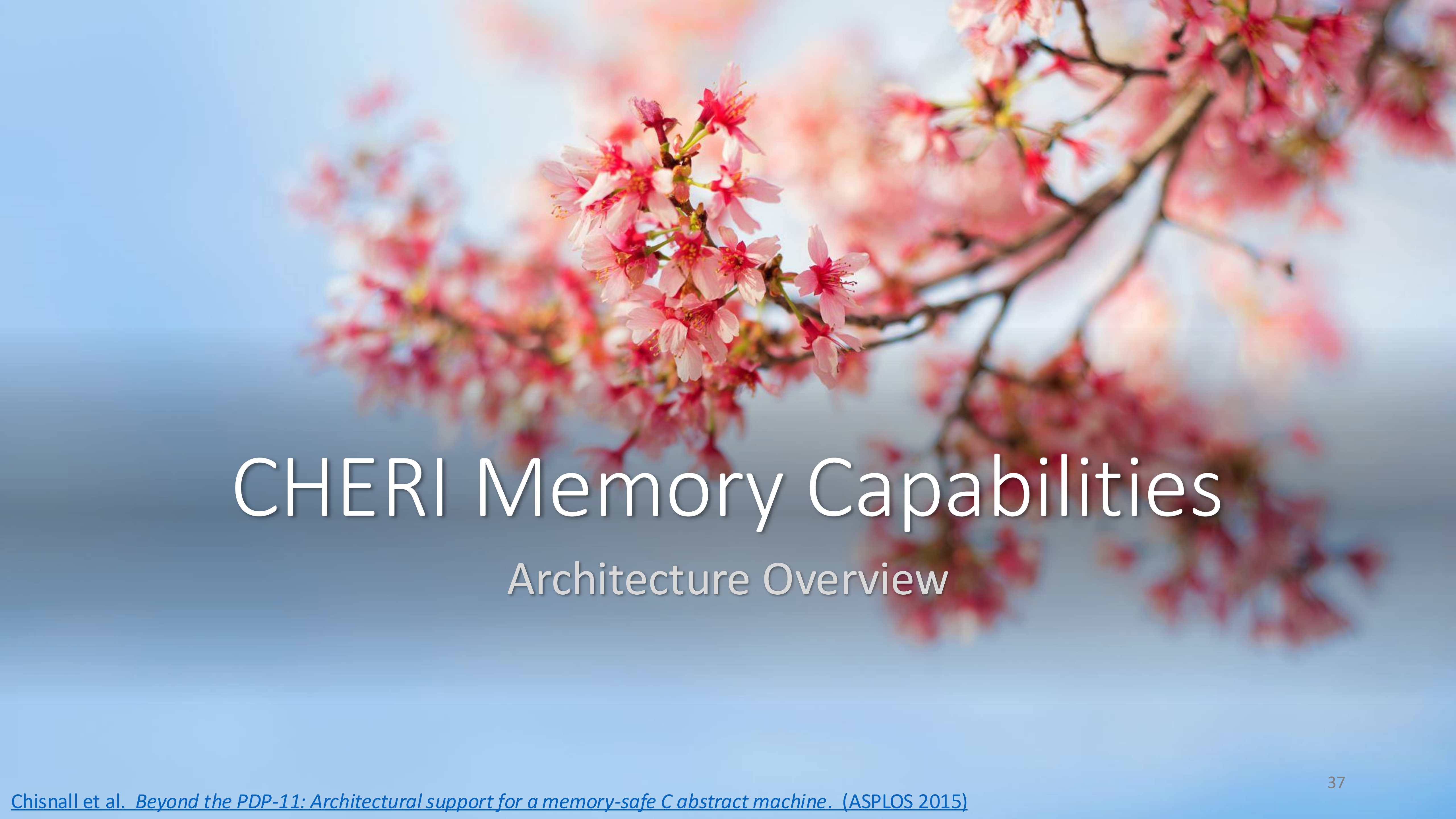
Architecture insufficiently informed:

1. ~~Nobody told the CPU about the buf object (its extent, lifetime, type, &c)~~ MTE color extent!
2. When code wrote out of bounds, the store silently corrupted memory
3. That memory was holding a pointer, ~~but CPU just thinks "bytes"~~ PAC checked!
4. ~~Deallocation and reuse of memory not communicated to CPU~~ MTE recolored!

*Things are decidedly looking better…*

## Probabilistic Defenses Should Be A Last Resort

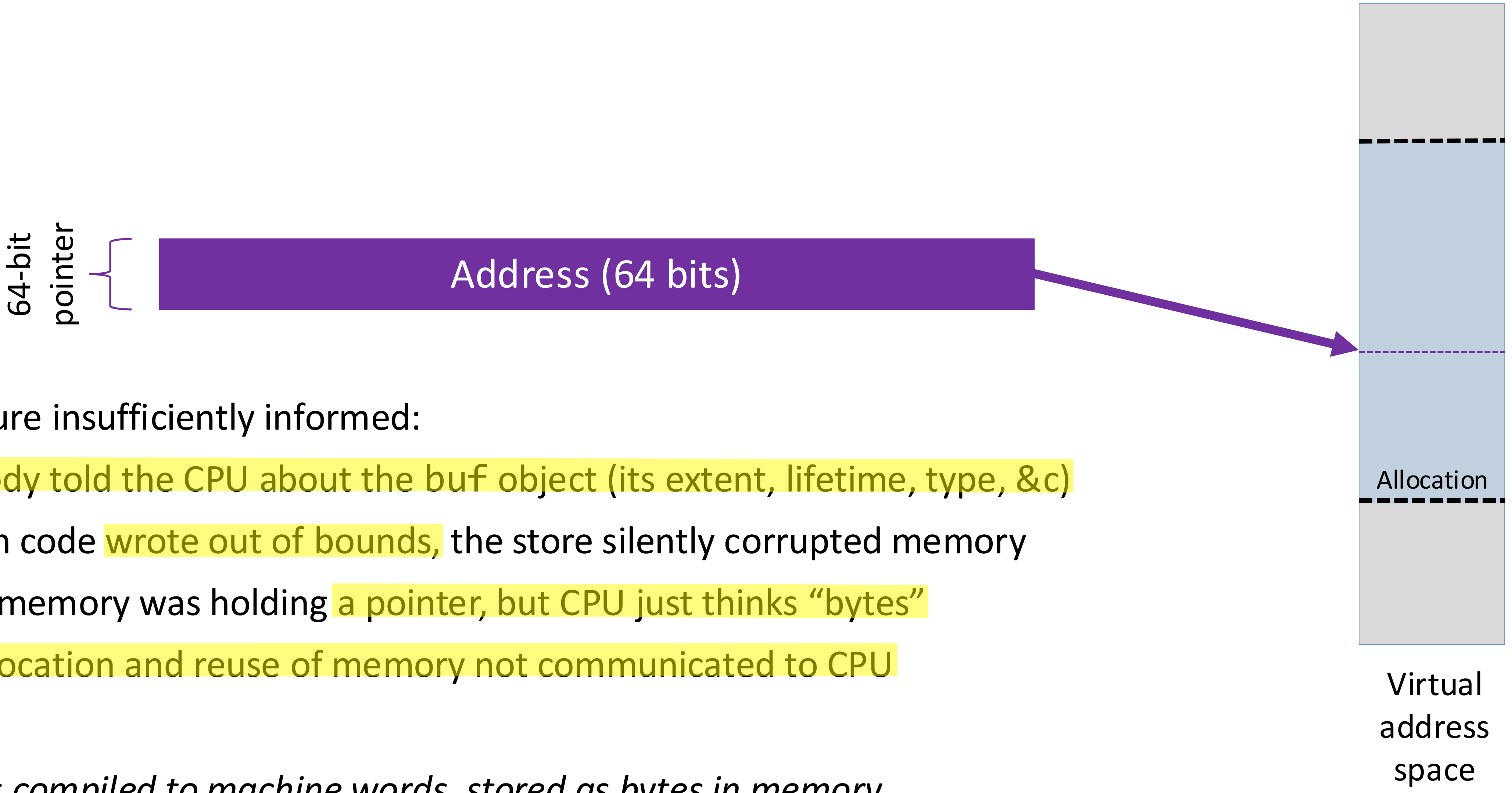
- PAC and MTE both fail *if the adversary "knows" the right secrets to forge a pointer*
- This is not an idle threat:
  - MTE weakens PAC in combination: 20-bit PAC + 4-bit color
  - Information disclosure vulnerabilities
  - Speculative side channels
  - Maybe use forging gadgets (PAC signing code, MTE memory or pointer recoloring code)
  - Might have repeated ability to guess (1M guesses is not a lot)
  - *Sometimes we call the adversary* ("library dependency", "foreign code", "plugin", "JIT-ed code")
- PAC and MTE show willingness to increase security by…
  - *getting new computers (adding metadata and new instructions to the architecture),*
  - *changing system software*, and
  - *recompiling*.
- If we're willing to do all that, can we do better than *probabilistic* defenses?



# CHERI Memory Capabilities

Architecture Overview

## Architecture Enables Safety Violations

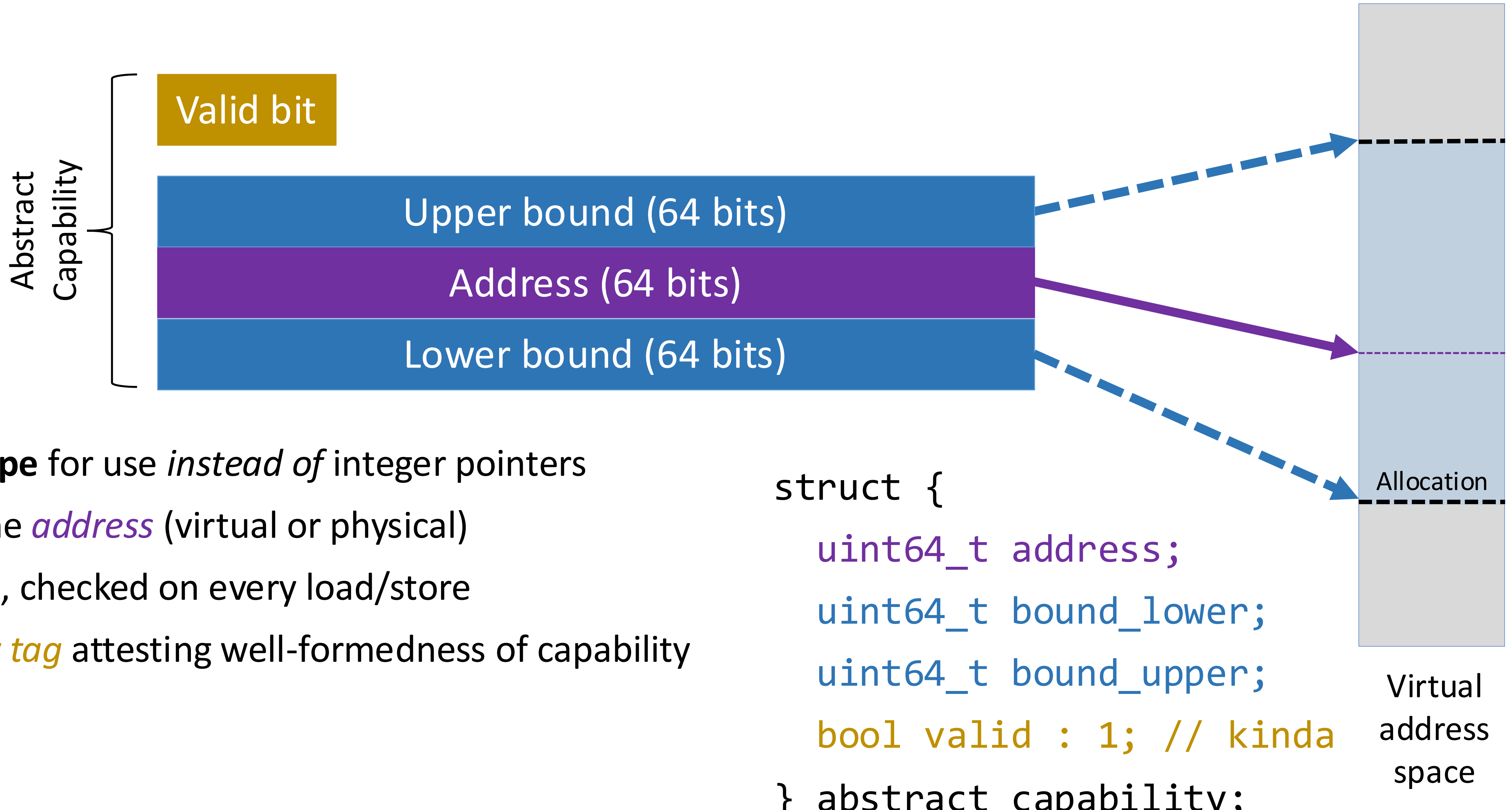


Architecture insufficiently informed:

1. Nobody told the CPU about the buf object (its extent, lifetime, type, &c)
2. When code wrote out of bounds, the store silently corrupted memory
3. That memory was holding a pointer, but CPU just thinks “bytes”
4. Deallocation and reuse of memory not communicated to CPU

*C pointers compiled to machine words, stored as bytes in memory.*

## Spatially-Safe C/C++ with Memory Capabilities



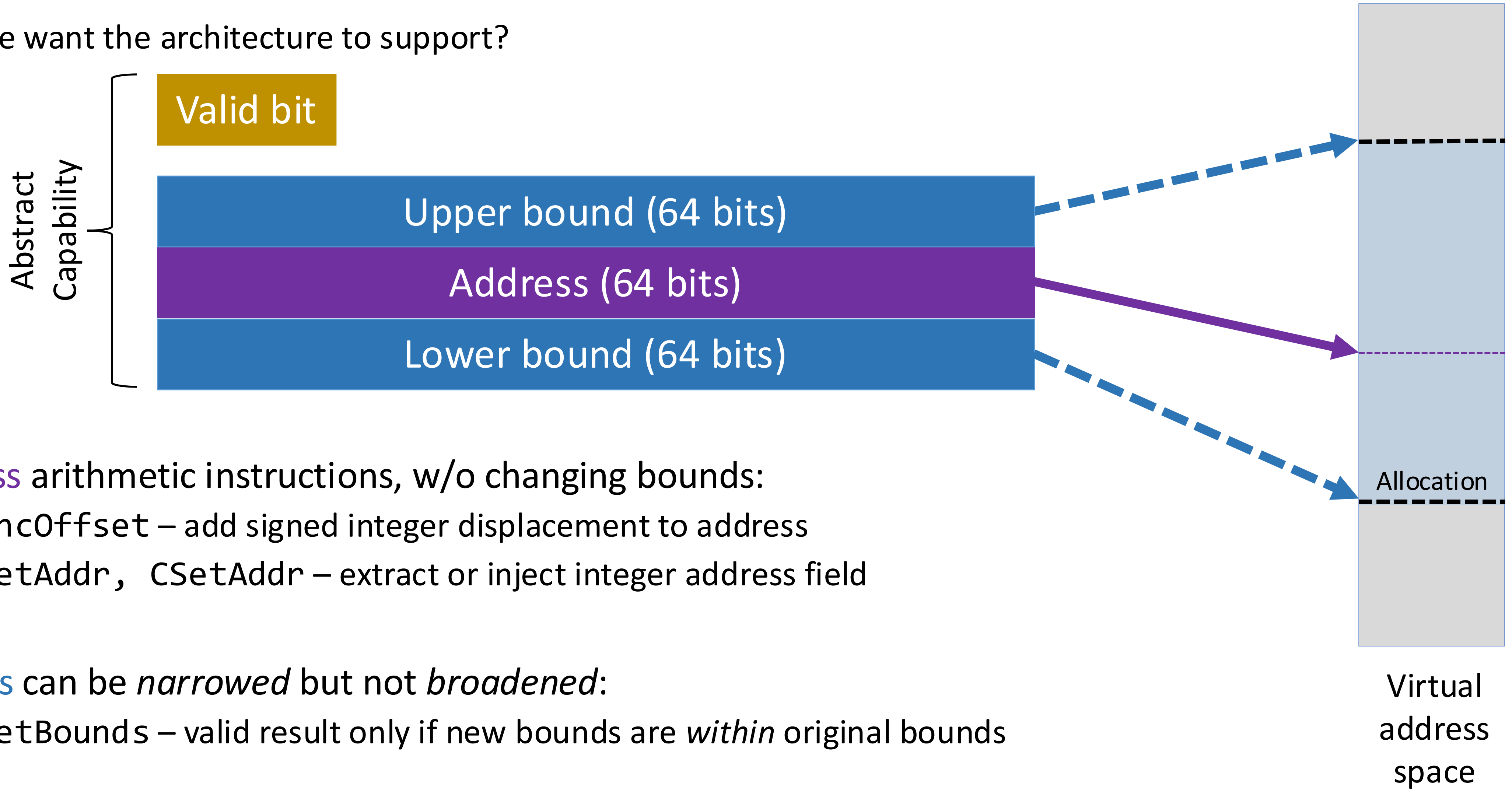
- New **datatype** for use *instead of* integer pointers
- Still need the *address* (virtual or physical)
- Add *bounds*, checked on every load/store
- Add *validity tag* attesting well-formedness of capability

```
struct {
  uint64_t address;
  uint64_t bound_lower;
  uint64_t bound_upper;
  bool valid : 1; // kinda
} abstract_capability;
```

Henry M. Levy. Capability-Based Computer Systems (1984)

## Operations on Capabilities

What do we want the architecture to support?

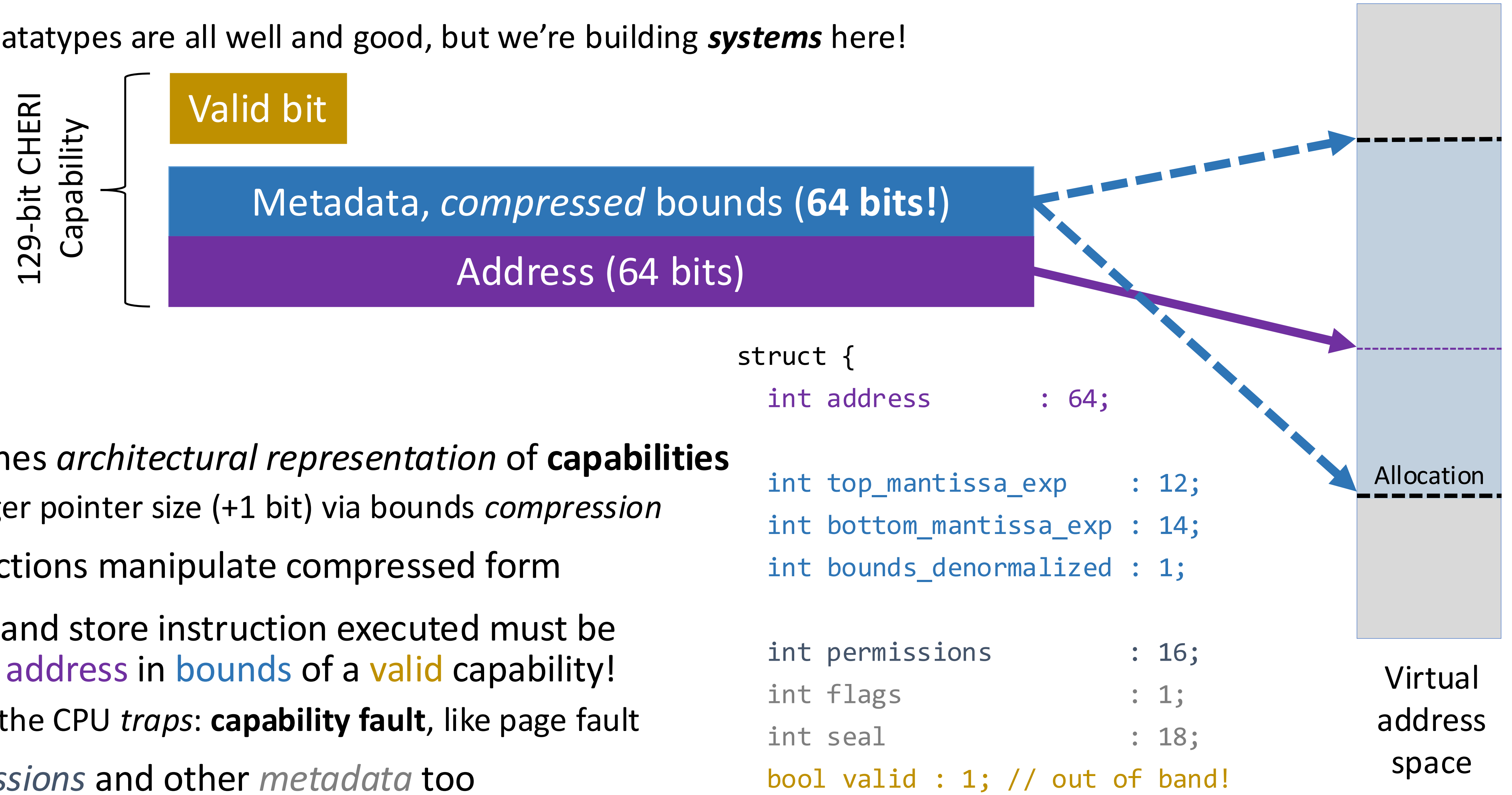


- Address arithmetic instructions, w/o changing bounds:
  - `CIncOffset` – add signed integer displacement to address
  - `CGetAddr, CSetAddr` – extract or inject integer address field
- Bounds can be *narrowed* but not *broadened*:
  - `CSetBounds` – valid result only if new bounds are *within* original bounds
- Validity tracking: capability valid only if it comes from another pointer via approved transforms

[CHERI Instruction-Set Architecture (Version 9)](#)

## CHERI: Memory Capabilities (For Real This Time)

Abstract datatypes are all well and good, but we're building ***systems*** here!

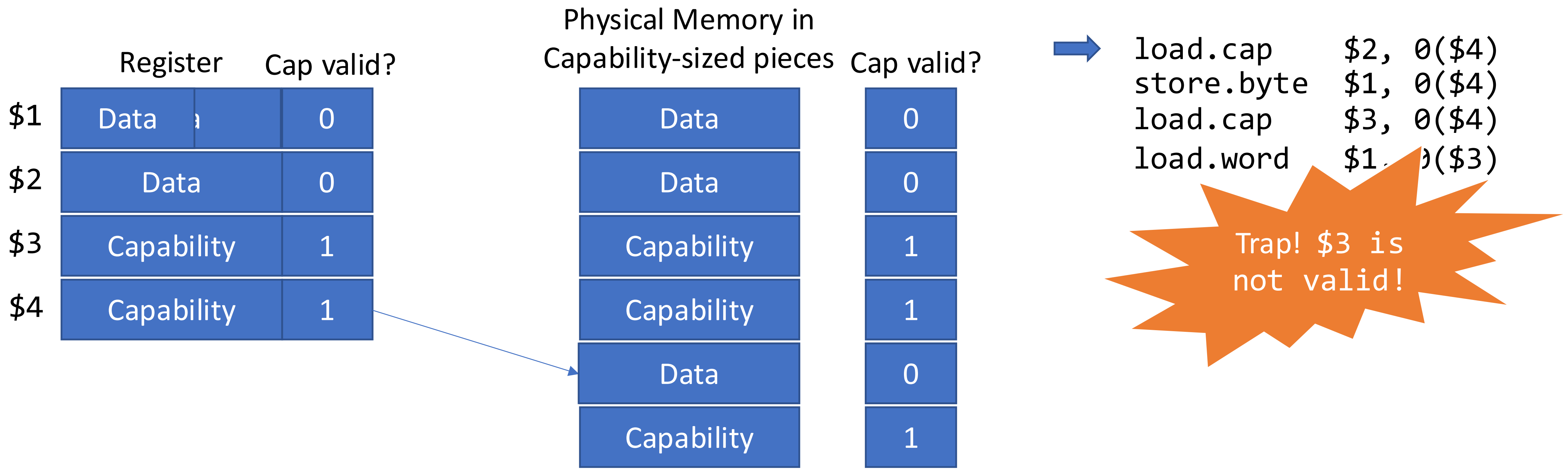


```
struct {
  int address            : 64;

  int top_mantissa_exp     : 12;
  int bottom_mantissa_exp  : 14;
  int bounds_denormalized  : 1;

  int permissions          : 16;
  int flags                : 1;
  int seal                 : 18;
  bool valid : 1; // out of band!
} CHERI_mem_cap;
```

- CHERI defines *architectural representation* of **capabilities**
  - 2x integer pointer size (+1 bit) via bounds *compression*
- CPU instructions manipulate compressed form
- Every load and store instruction executed must be to an address in bounds of a valid capability!
  - Or else the CPU *traps*: **capability fault**, like page fault
- Add *permissions* and other *metadata* too

CHERI Instruction-Set Architecture (Version 9)

## Capabilities in Registers and Memory



```
load.cap     $2, 0($4)
store.byte   $1, 0($4)
load.cap     $3, 0($4)
load.word    $1, 0($3)
```

- Program counter register also holds a capability!
- CHERI embodies a very simple (1-bit) “dynamic type” system:
  - Every word is *either* a capability *or* just an integer
  - Trap if integer used where a capability is required

## Misbehaving C Program: Spatial & Referential Safety Violations

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

AArch64

```
foo:
  mov      w8, #65
  strb     w8, [x0, #16]
  strb     w8, [x0, #32]
  ret
main:
  sub      sp, sp, #48
  stp      x29, x30, [sp, #32]
  add      x29, sp, #32
  mov      x0, sp

  bl       foo

  ldp      x29, x30, [sp, #32]
  mov      w0, wzr
  add      sp, sp, #48
  ret      // x30
```

Stores relative to address in x0

x0 holds *address* of buf on stack

| Stack as of entry to foo() | |
|---|---|
| sp+32 | main's saved RA |
| sp+16 | pad[0] … [15] |
| sp+0 | buf[0] … [15] |

a0 = &buf[0]

## Misbehaving C Program: Spatial & Referential Safety Violations

```c
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

Morello

```
foo:
  mov      w8, #65
  strb     w8, [c0, #16]
  strb     w8, [c0, #32]
  ret
main:
  sub      csp, csp, #64
  stp      c29, c30, [csp, #32]
  add      c29, csp, #32
  scbnds   c0, csp, #16

  bl       foo

  ldp      x29, x30, [sp, #32]
  mov      w0, wzr
  add      sp, sp, #48
  ret      // c30
```

Stores relative to capability in c0

x0 holds *cap to* buf on stack

| Stack as of entry to foo() | | V |
|---|---|---|
| sp+32 | main's saved RA | 1 |
| sp+16 | pad[0] … [15] | 0 |
| sp+0 | buf[0] … [15] | 0 |

ca0

gdb says:

```
Program received signal SIGPROT, CHERI protection violation
Capability bounds fault
… in foo (buf=0x3fffdfff70 [rwRW,0x3fffdfff70-0x3fffdfff80] …)
```

CHERI C/C++ Programming Guide (2020)

## Secret-Free, Deterministic Mechanism

- CHERI is *secret-free* and *deterministic*, in contrast to PAC and MTE.
- An adversary cannot forge a capability *even if they know every bit of system state*.
  - No MTE colors, PAC secrets, ASLR slide, …
  - Can't re-inject *data* as *pointers*: no more Smashing The Stack For Fun And Profit *even ignoring bounds*
  - *Speculative execution* not a threat to protection mechanism
- Out-of-bounds or invalid dereference *always* traps.
- Byte-level corruption or attempts to widen bounds *always* caught (clear tag or trap).

Amar et al. *An Armful of CHERIs*. (2022)

## CheriABI: Spatially Safe *NIX Processes

Significant *ambient authority* in modern *nix-like systems: **system calls**!

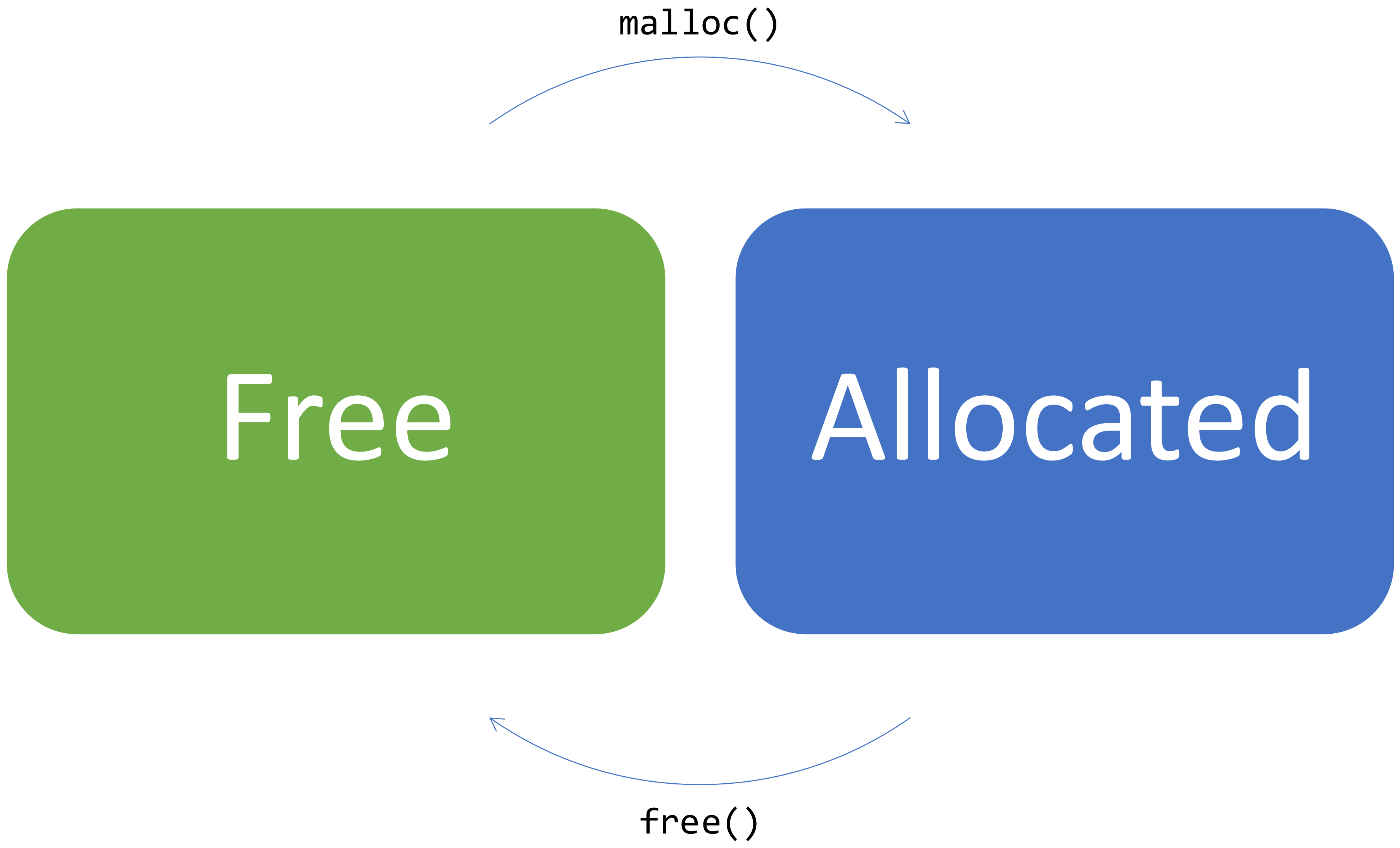
- Code might *mislead kernel* into violating spatial safety ("confused deputy"). Consider:

```
char buf[1024];
read(fd, buf, 2048);
```

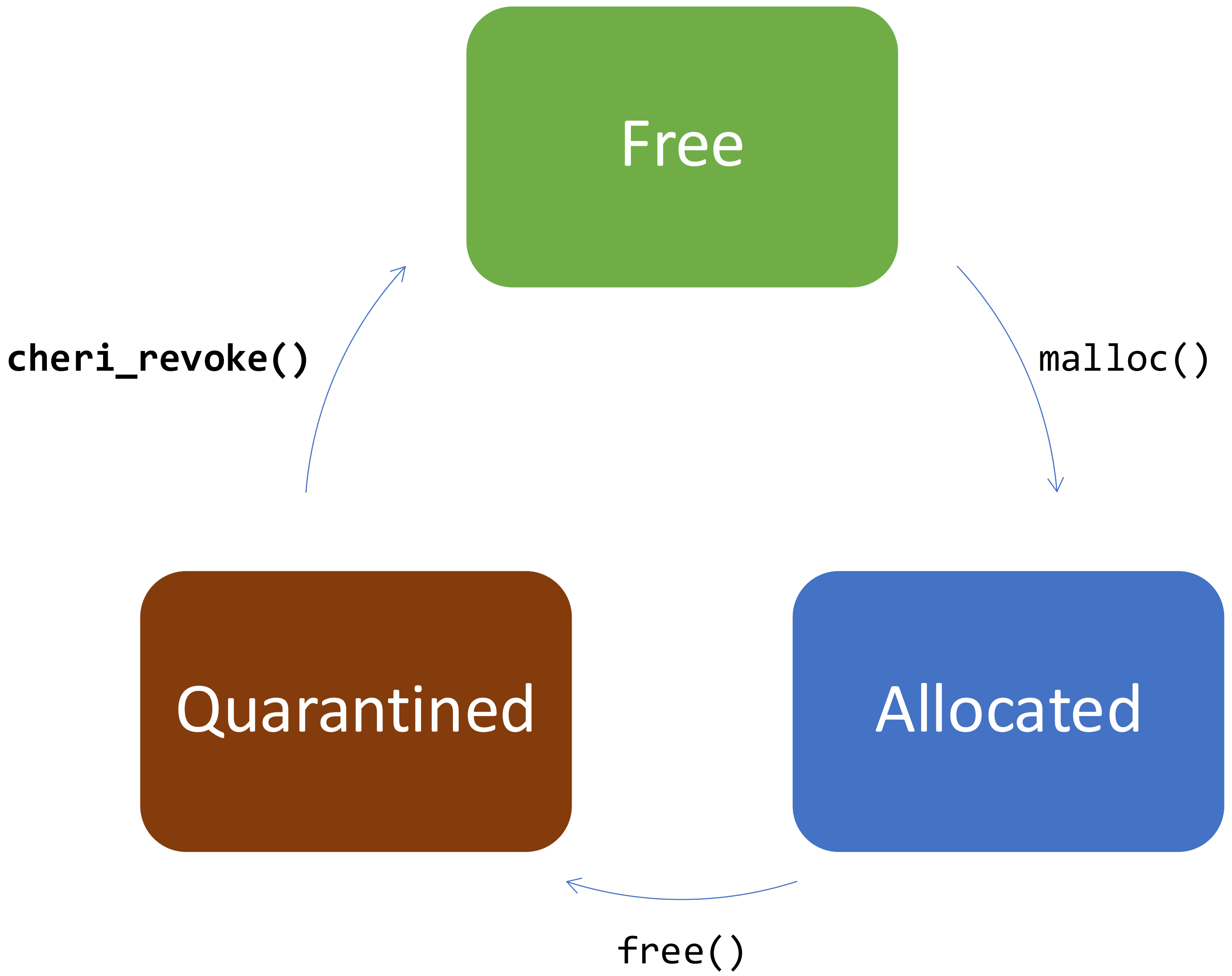
- CheriABI makes system calls take and return *capabilities* instead of integer addresses!
  - Kernel uses passed-in capabilities to *limit its own behavior*.
  - `read(fd, buf, len)` won't write beyond `buf`'s capability bounds, even if `len` says to!
  - Passes the *user's* `buf` to BSD's centralized `copyout()` facility.
    - Facility exists to deal with page faults.
    - Easily extended for CHERI faults; *no new bounds-check instructions*!

## CHERI Heap Temporal Safety

## Cornucopia: CHERI Heap Temporal Safety



## Cornucopia: CHERI Heap Temporal Safety



## Cornucopia Eliminates Heap Use After Reallocation

So now what?

```
char *p = malloc(1024); // returns capability to memory @ 0x15410DE0U
free(p);
char *q = malloc(1024); // definitely not 0x15410DE0U

strcpy(p, "oh no");     // Allowed for “a while”, writes to old p
// At some later point, “magically”, p becomes NULL
```

This works because *CHERI tags make it easy to scan for pointers*;

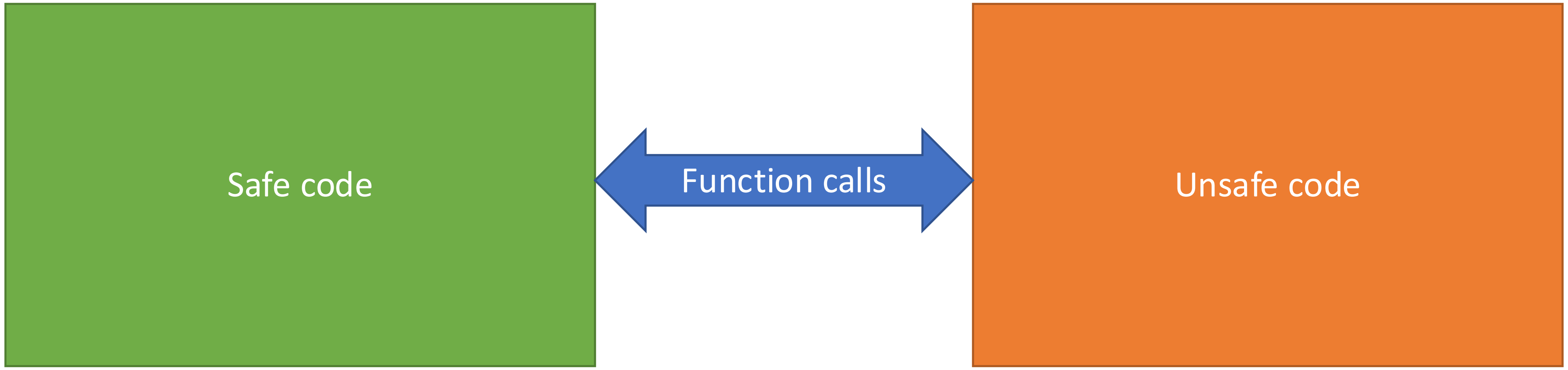
pointers to free memory can be *deleted*.

[Cornucopia: Temporal Safety for CHERI Heaps (2020)]

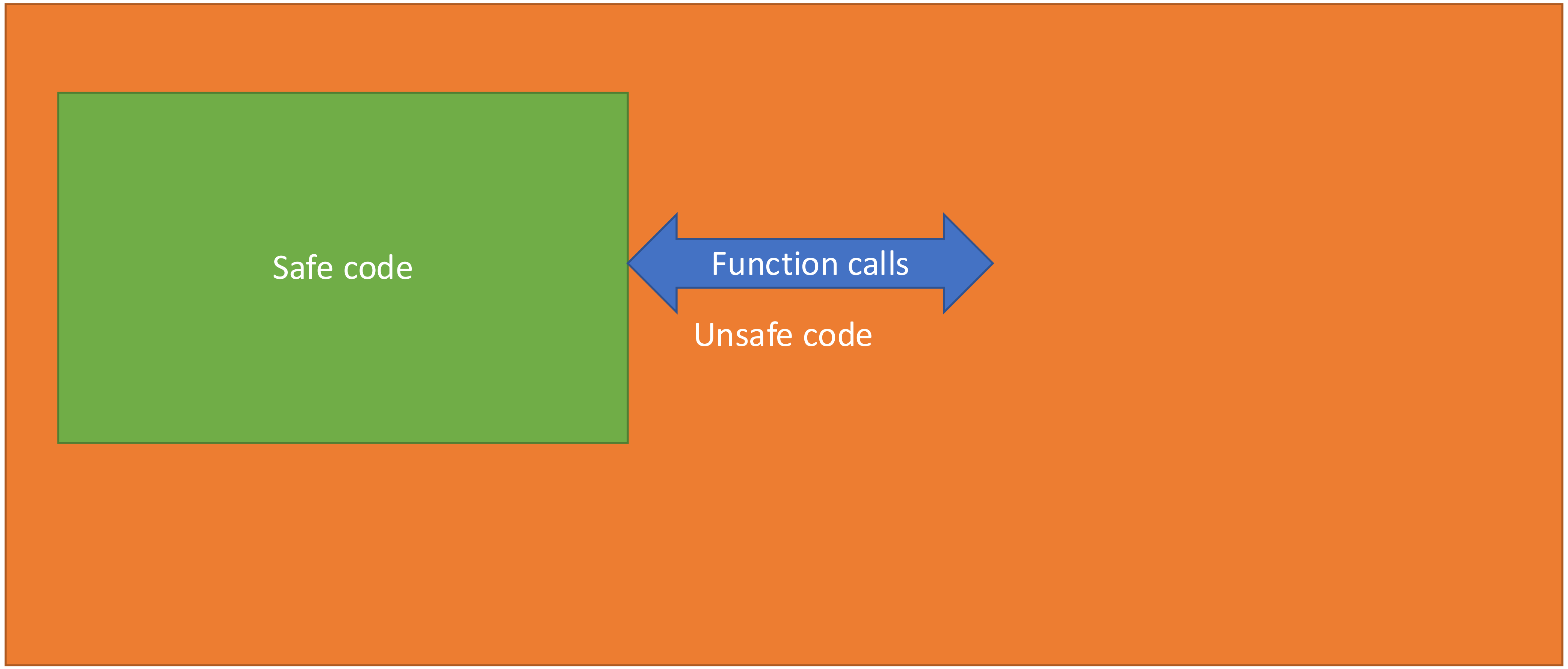
[Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety (2024)]

## Is New Architecture Competing With Safe Languages?

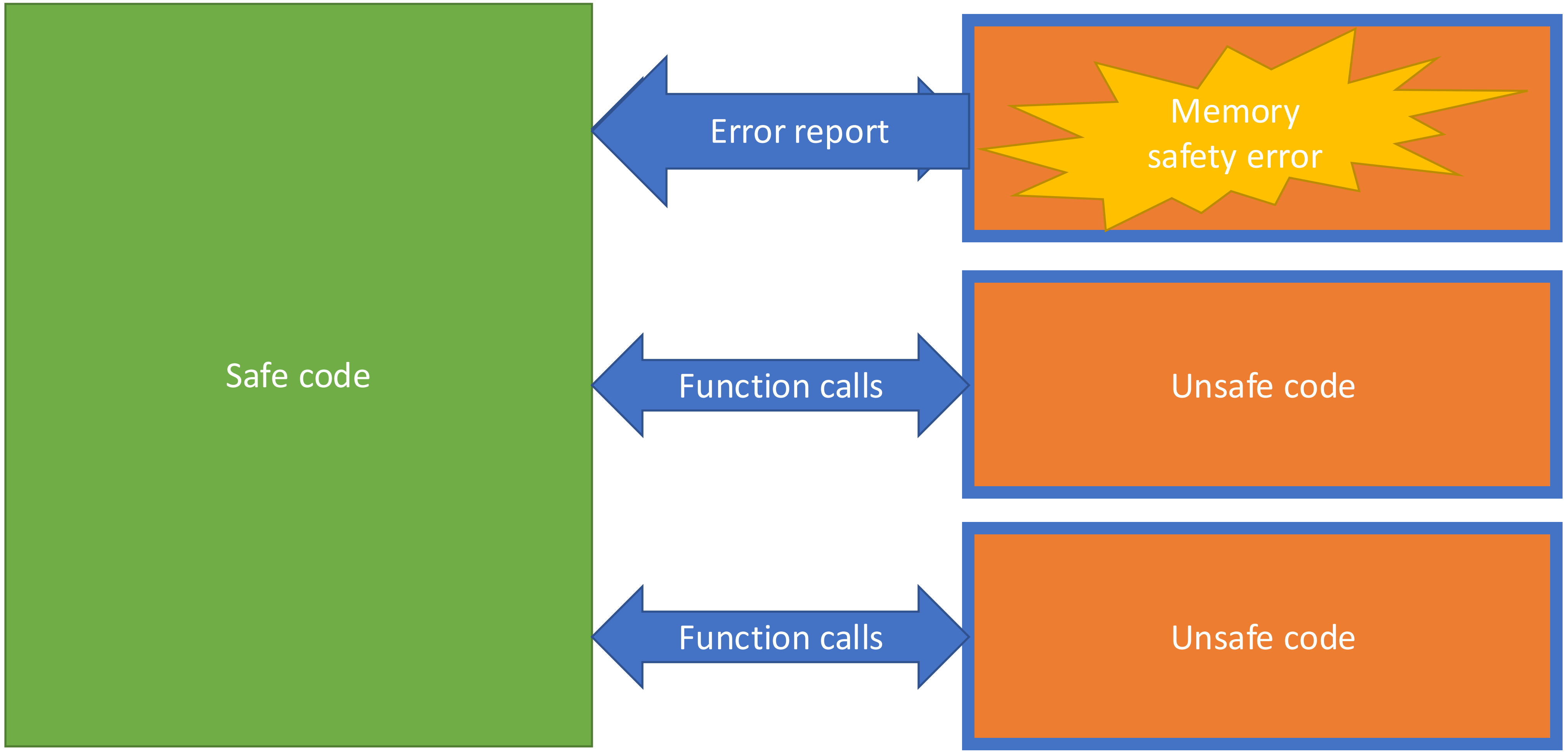
## A two-worlds abstraction?



## A two-worlds abstraction... leaks!



## A safe many-worlds abstraction



Chisnall et al. CHERI JNI: Sinking the Java security model into the C. (ASPLOS 2017)

Chisnall. I Don't Care About Memory Safety. (2023)

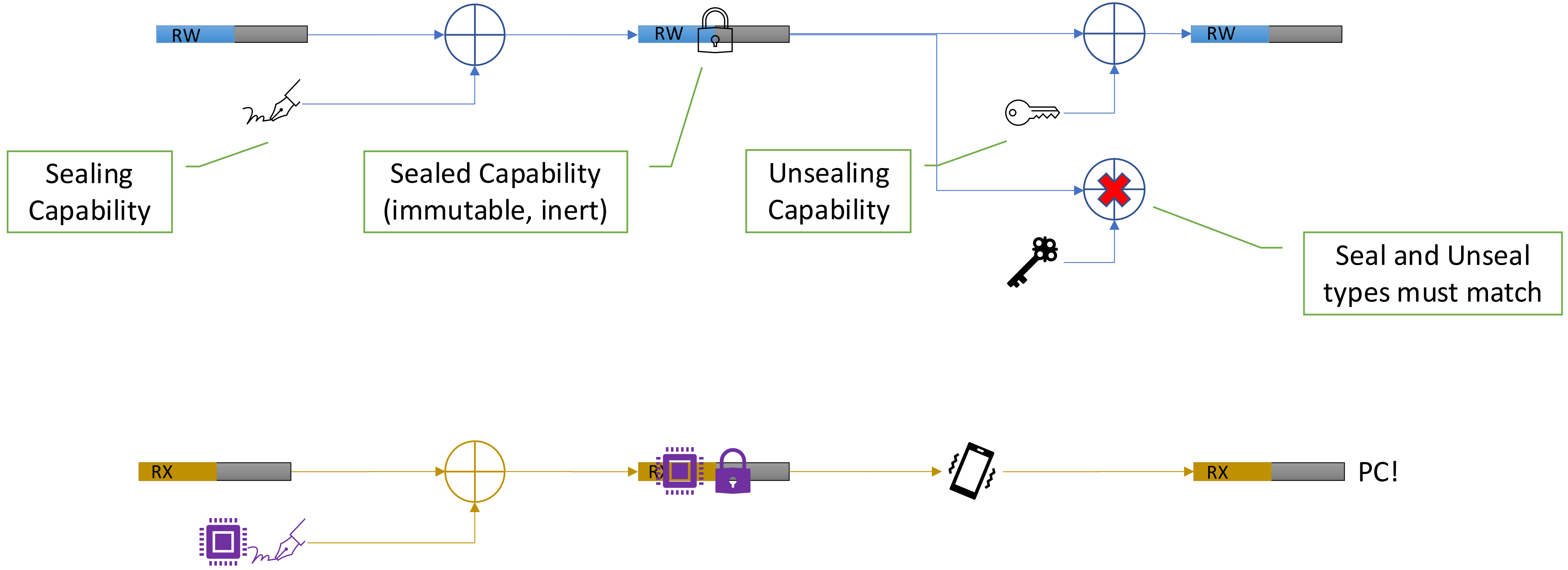
## Deterministic Memory Safety Enables Compartmentalization

- CHERI is not actually a memory safety technology, it is a *compartmentalization* technology
  - Memory safety is a necessary but not sufficient precondition
- Can *build* confined pieces of software with access to only particular resources
  - Without a (transitive) capability to a given resource, no way to access it! (Even if address known!)
- Simplest case is a CODEC (`xz`, `libpng`, …). If *all* we give some CODEC code is…

| Resource | Permissions |
|---|---|
| CODEC code (& constants) | Read, Execute |
| Input buffer(s) | Read-only |
| Output buffer(s) | Write-only |
| Ephemeral stack / scratch region | Read, Write |
| Return pointer | Execute only? |

  - … then even a fully compromised CODEC has very limited consequence on the broader program!
- Entering sandbox is easy; getting back out might be tricky?

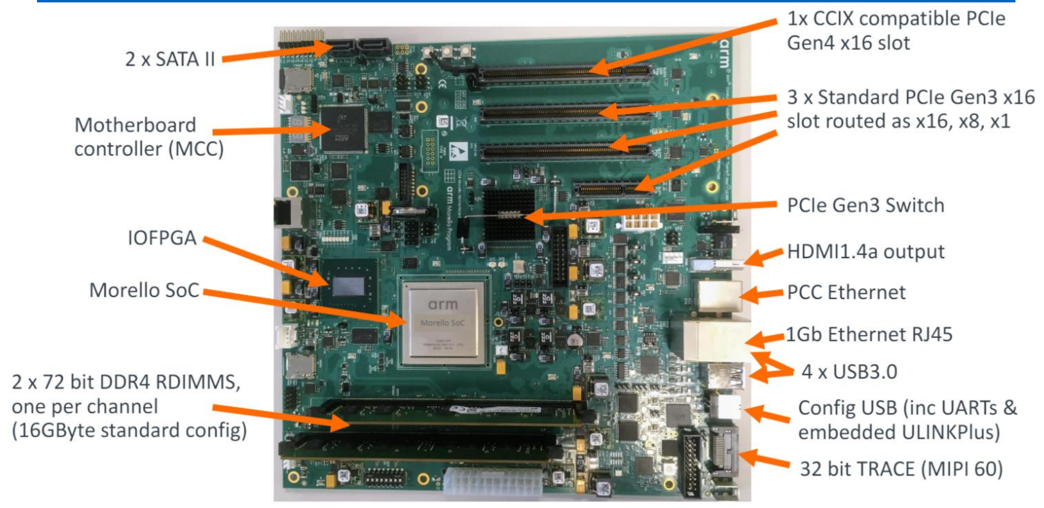
## Sealed and Sentry Capabilities



RW
RW
RW
Sealing Capability
Sealed Capability (immutable, inert)
Unsealing Capability
Seal and Unseal types must match
RX
RX
RX
PC!

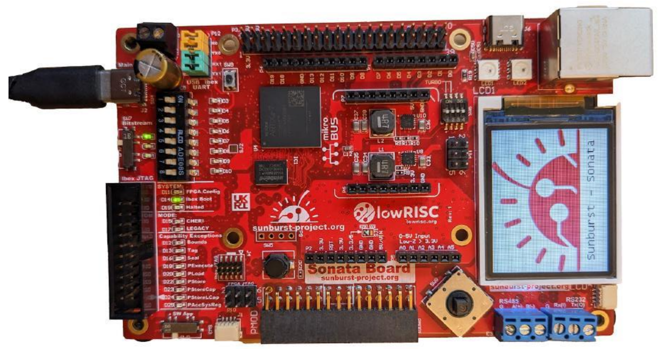
# CHERI Is Escaping The Lab And Heading For The Village

## "Morello" prototype SoC & board: 4-core, 2.5-GHz


2 x SATA II
Motherboard controller (MCC)
IOFPGA
Morello SoC
2 x 72 bit DDR4 RDIMMS, one per channel (16GByte standard config)
1x CCIX compatible PCIe Gen4 x16 slot
3 x Standard PCIe Gen3 x16 slot routed as x16, x8, x1
PCIe Gen3 Switch
HDMI1.4a output
PCC Ethernet
1Gb Ethernet RJ45
4 x USB3.0
Config USB (inc UARTs & embedded ULINKPlus)
32 bit TRACE (MIPI 60)

arm

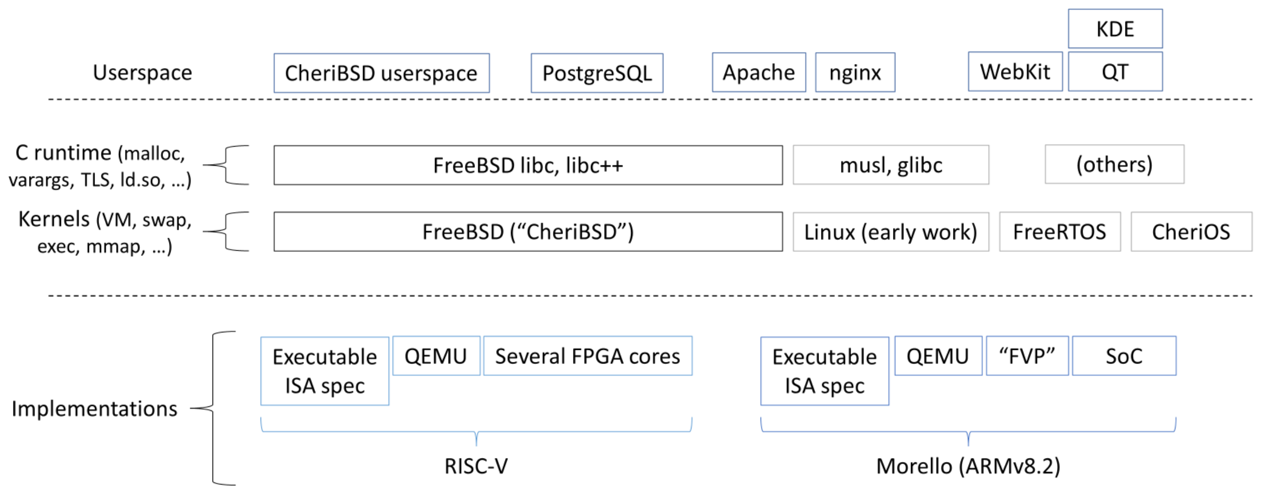
## CHERIoT (32-bit CHERI RISC-V)

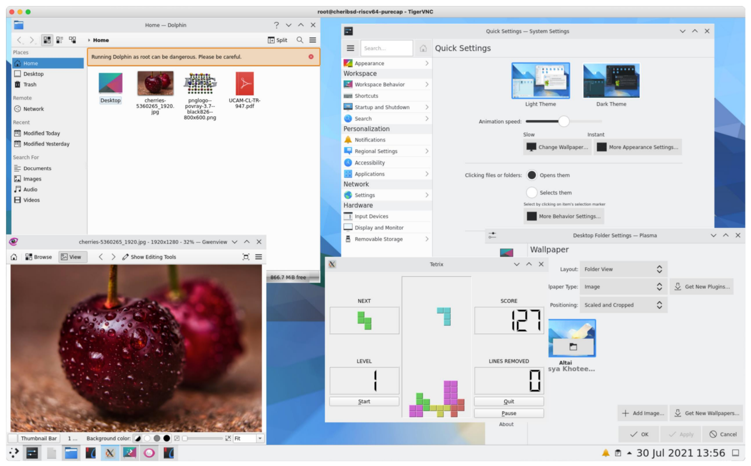


lowRISC Sonata board

SCI Semi ICENI

### CHERI Ecosystem At A Glance





Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem

## Architectural Metadata for Memory Safety

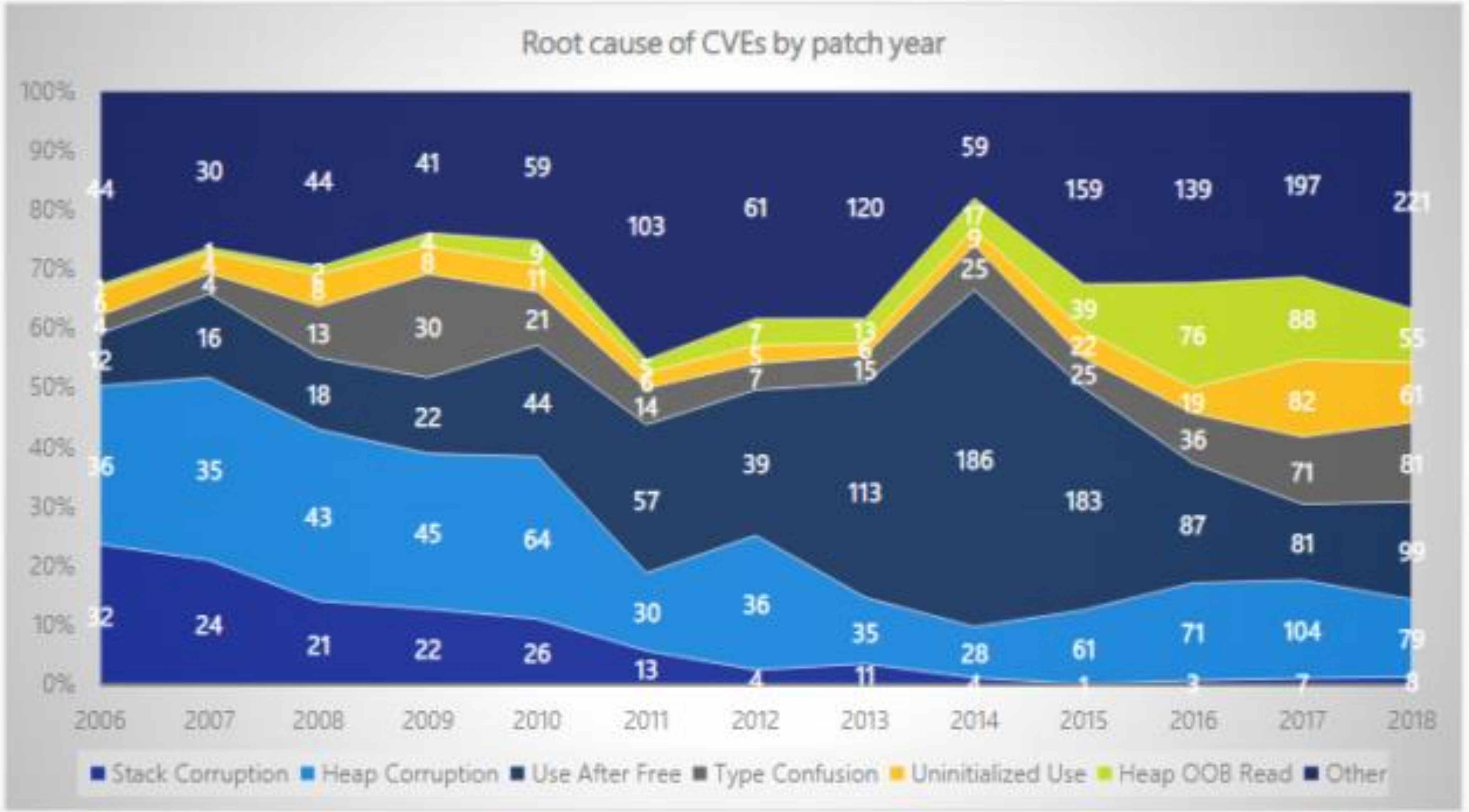
Architectural metadata: an idea whose time has finally come?

- New systems to better let the CPU understand *programmer intent*
  - PAC: “sign” pointers to convey authenticity and intent
  - MTE: “color” memory to convey information about object layout and lifetime
  - CHERI: replace pointers with “*capabilities*”, unforgeable tokens of authority
- Useful to debug and/or mitigate the cause of many long-standing classes of security vulnerabilities
- PAC has been around, MTE shipped last year, CHERI very soon through next few years!
- If you continue to be systems programmers, expect to see *more and different kinds of metadata*

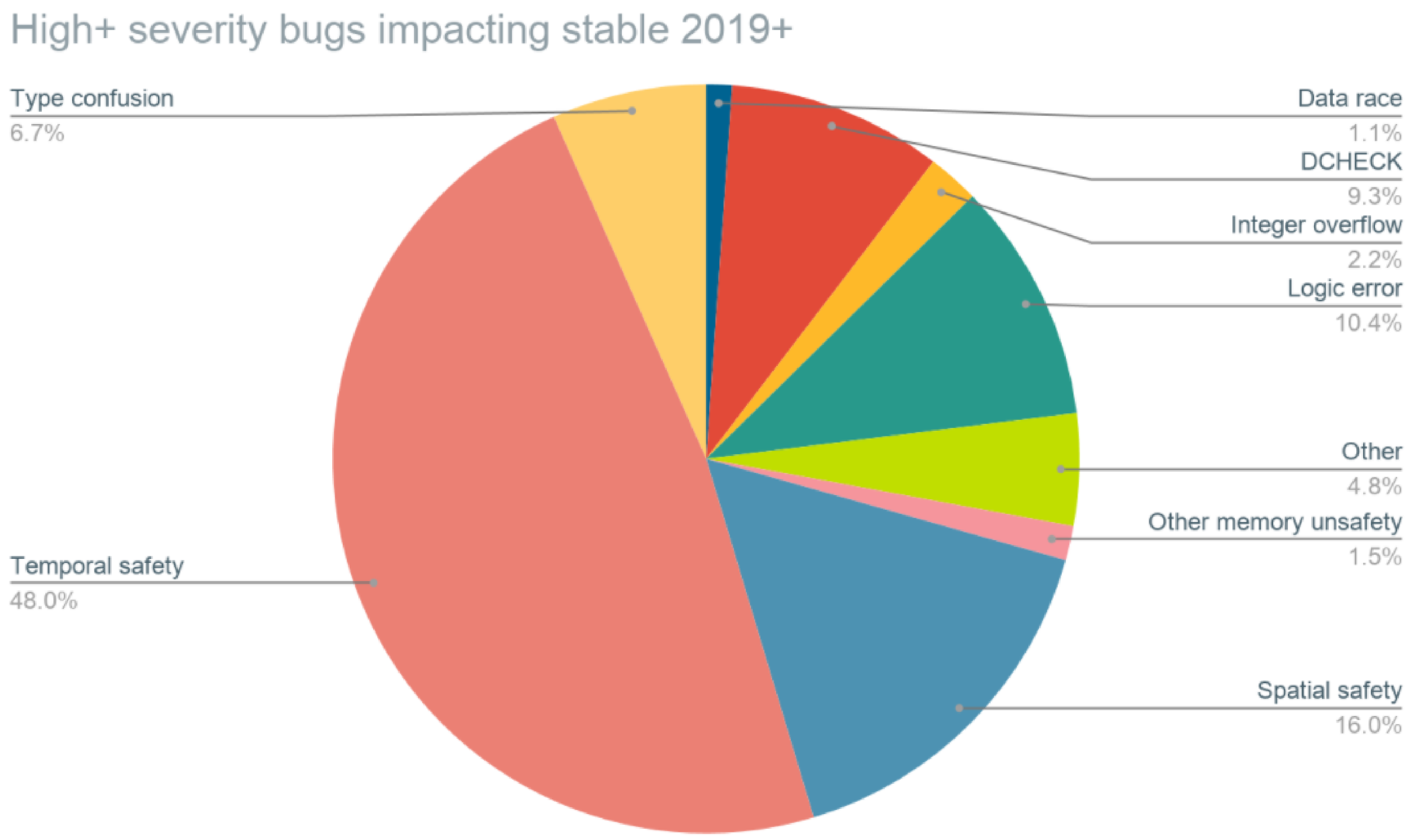
## Design Matrix!

| | PAC | MTE | CHERI |
|---|---|---|---|
| **Metadata Location** | In-pointer | In-pointer + out of band (4) | In-pointer + out of band (1) |
| **Pointer & address size** | Native; ~40 bits | Native; ~60 bits | 2x Native; Native |
| | | | |
| **Pointer Integrity** | Yes, but *cryptographic* | No | Yes, *deterministic* |
| **Adjacent overflow** | No | Yes<br>O(n) and flat | Yes, O(1) and can nest |
| **General spatial bounds** | | Stochastic | |
| **Heap obj. temporal safety** | No | UAF, yes; UAR, stochastic | UAF safe; UAR via sweeping |
| **Flow control** | Some: context word | No | Some: sealing & others |
| **Secrets?** | Yes ☹ | Yes ☹ | No ☺ |
| | | | |
| **Hardware mods required** | New instructions | New instructions, checks and traps, OOB colors, caches | Wider registers, new instructions, checks and traps, OOB tags, caches |
| **Software modes required** | Compiler (& recompile), small kernel changes | Heap allocator, compiler (& recompile), small kernel changes | Compiler (& recompile), kernel, libc, & small app changes |

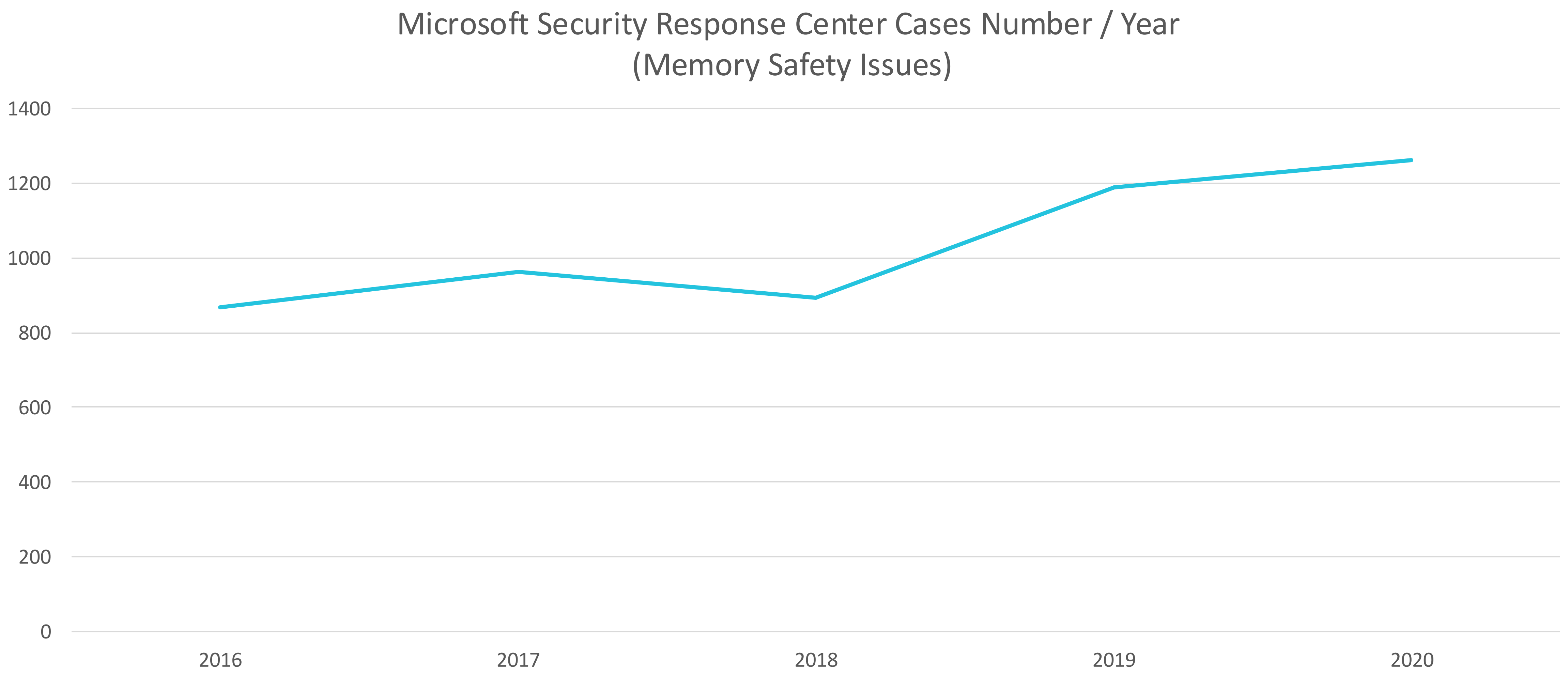
## CVEs and High Severity Bugs from (Lack of) Memory Safety



## CVEs and High Severity Bugs from (Lack of) Memory Safety

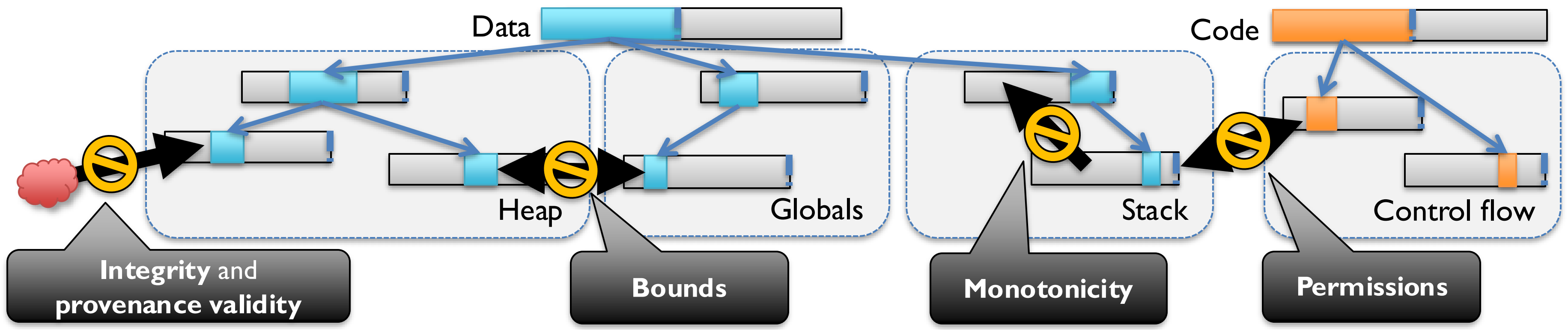


## CVEs and High Severity Bugs from (Lack of) Memory Safety



Amar et al. *Security Analysis of CHERI ISA*. (Blackhat 2021)

# CHERI enforces protection semantics for pointers



- **Integrity** and **provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
  - Valid pointers, once removed, cannot be reintroduced solely unless rederived from other valid pointers
  - E.g., Received network data cannot be interpreted as a code/data pointer – even previously leaked pointers
- **Bounds** prevent pointers from being manipulated to access the wrong object
  - Bounds can be minimized by software – e.g., stack allocator, heap allocator, linker
- **Monotonicity** prevents pointer privilege escalation – e.g., broadening bounds
- **Permissions** limit unintended use of pointers; e.g., W^X for pointers
- These primitives not only allow us to implement **strong spatial and temporal memory protection**, but also higher-level policies such as **scalable software compartmentalization**

## Misbehaving C Program, Now With CHERI but Without Narrowed Bounds?

```
void foo(char *buf) {
  buf[16] = 'A';
  buf[32] = 'A';
}

int main(void) {
  char pad[16], buf[16];

  foo(buf);
  return 0;
}
```

CHERI RISC-V, w/o csetbounds

```
0000000000001b00 <foo>:
   addi        a1, zero, 65
   csb         a1, 16(ca0)
   csb         a1, 32(ca0)
   cret

0000000000001b10 <main>:
   cincoffset  csp, csp, -48
   csc         cra, 32(csp)
   cmove       ca0, csp
   auipcc      cra, 0
   cjalr       -28(cra)
   mv          a0, zero
   clc         cra, 32(csp)
   cincoffset  csp, csp, 48
   cret
```

Whoops! Forgot to narrow bounds.

| Stack as of entry to foo() | | V |
|---|---|---|
| sp+32 | in's saved %cra | 0 |
| sp+16 | ad[0] … [15] | 0 |
| sp+0 | buf[0] … [15] | 0 |

ca0

gdb says:

```
Program received signal SIGPROT, CHERI protection violation
Capability tag fault caused by register cra
… in main ()
```

## CHERI: A New Foundation for Software Security?

CHERI is…

- 12+ year project from the University of Cambridge's Computer Laboratory

- radical, "new computer" approach: change *how pointers work*
  - A foundational shift akin to turning on virtual memory between P1 and P3; things will be *different*.

- not so radical after all?
  - CHERI composes well with modern microarchitectures
  - Maybe C/C++ (and FFI) can be made safe(r)

## CHERI Summary

- CHERI enriches CPUs to have tagged *capabilities* with architecturally-enforced *invariants*
  - Solves many *root causes* of long-standing security vulnerabilities
  - Promising *compartmentalization* designs
  - If nothing else, a good candidate for the 410 book-report!

- Looks quite real: FPGA RISC-V & Arm Morello SoC, LLVM, CheriBSD, Qt, KDE, …

- If you want to know more, please do get in touch:
  - http://www.cheri-cpu.org/ for (much) more reading material, Slack, e-mail lists, &c.
  - CHERI-related 412 projects!

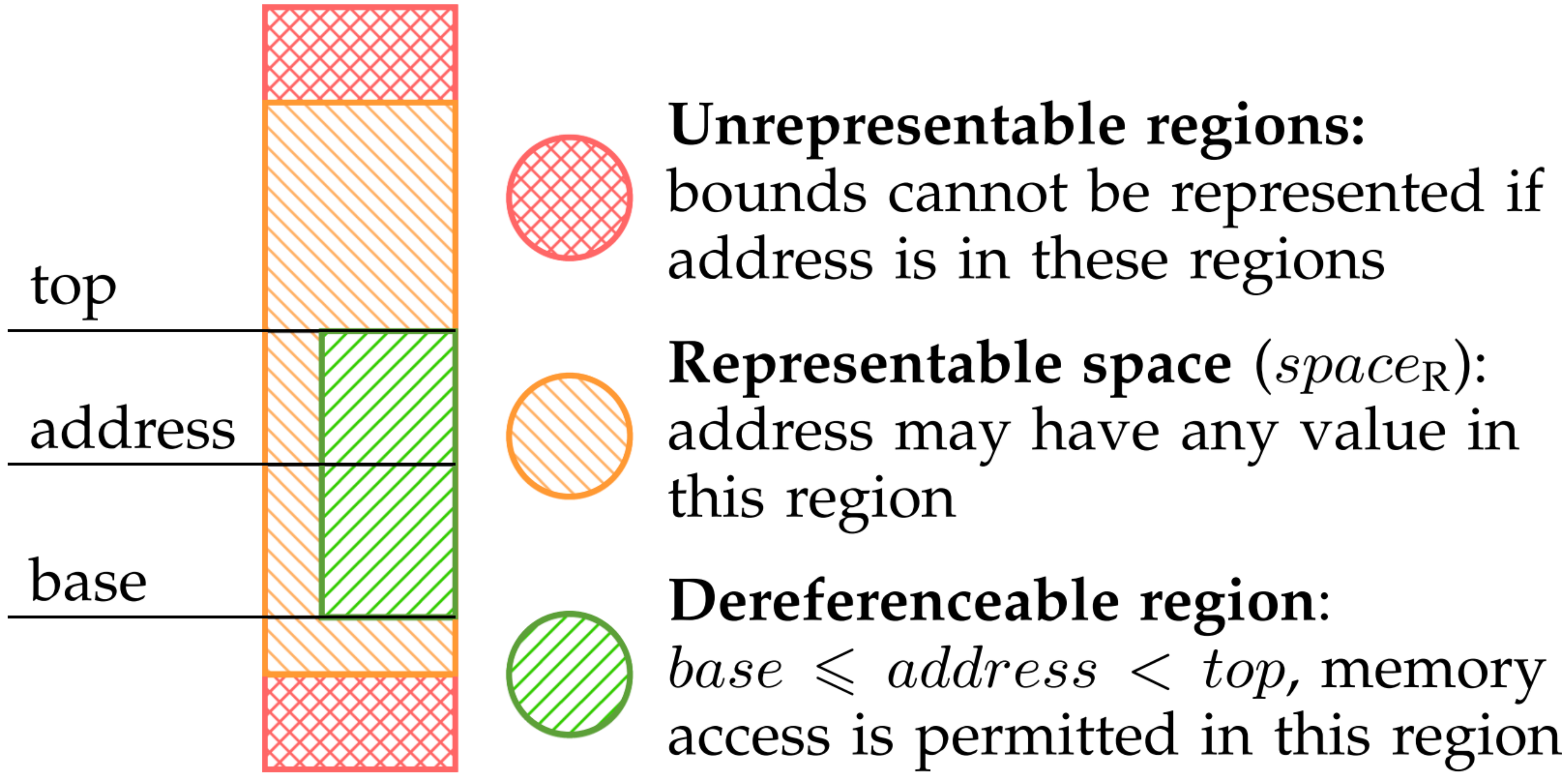
- Play along at home, too; almost everything is FLOSS:
  - https://github.com/CTSRD-CHERI/cheripedia/wiki/Getting-Started a how-to (from another former 410 TA!)
  - https://github.com/ctsrd-cheri/cheribuild one-stop-shop build system
  - https://github.com/CTSRD-CHERI/cheri-exercises hands-on introductory exercises

## Book Report Fodder

CHERI:

- Watson et al. *Introduction to CHERI*. (Tech report, 2019).
- Joly et al. *Security analysis of CHERI ISA*. (2020).
- Microsoft Security Response Center. *What's the smallest variety of CHERI?* (2022)
- Chisnall et al. *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine*.
- Davis et al. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*. (extended report).
- Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*.
- Joannou et al. *Efficient Tagged Memory*.
- Esswood. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.
- Watson et al. Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA).
- Capabilities Limited. Assessing the Viability of an Open Source CHERI Desktop Software Ecosystem.
- CHERI Instruction-Set Architecture (Version 9).
- Henry M. Levy. *Capability-Based Computer Systems*.

## CHERI Concentrate Representability



🟥 **Unrepresentable regions:** bounds cannot be represented if address is in these regions

🟧 **Representable space** ($space_R$): address may have any value in this region

🟩 **Dereferenceable region**: $base \leqslant address < top$, memory access is permitted in this region

Fig. 8. Memory regions implied by a CC encoding.

Woodruff et al. *CHERI Concentrate: Practical Compressed Capabilities.* (2019)

## CHERI Concentrate Representability

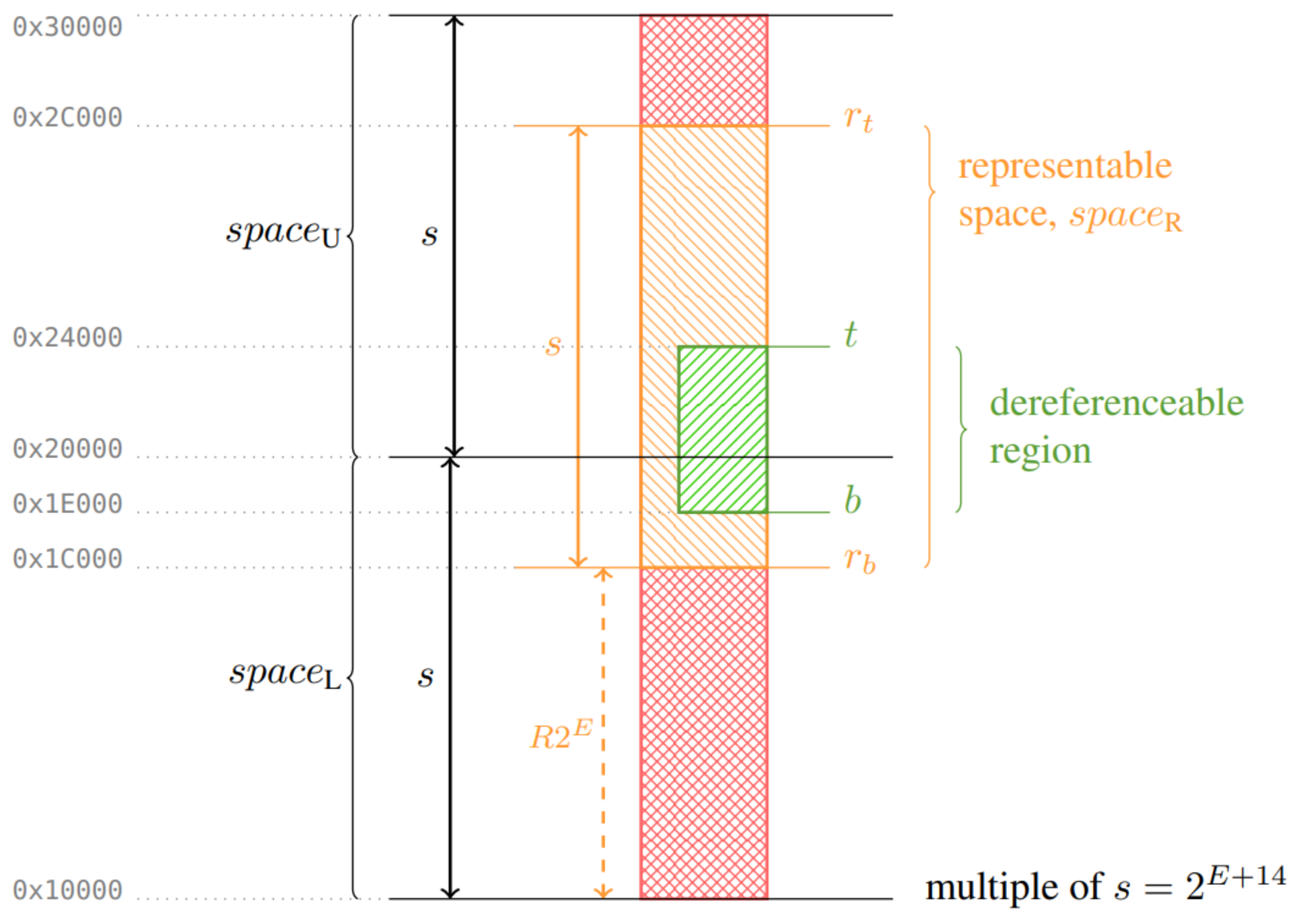
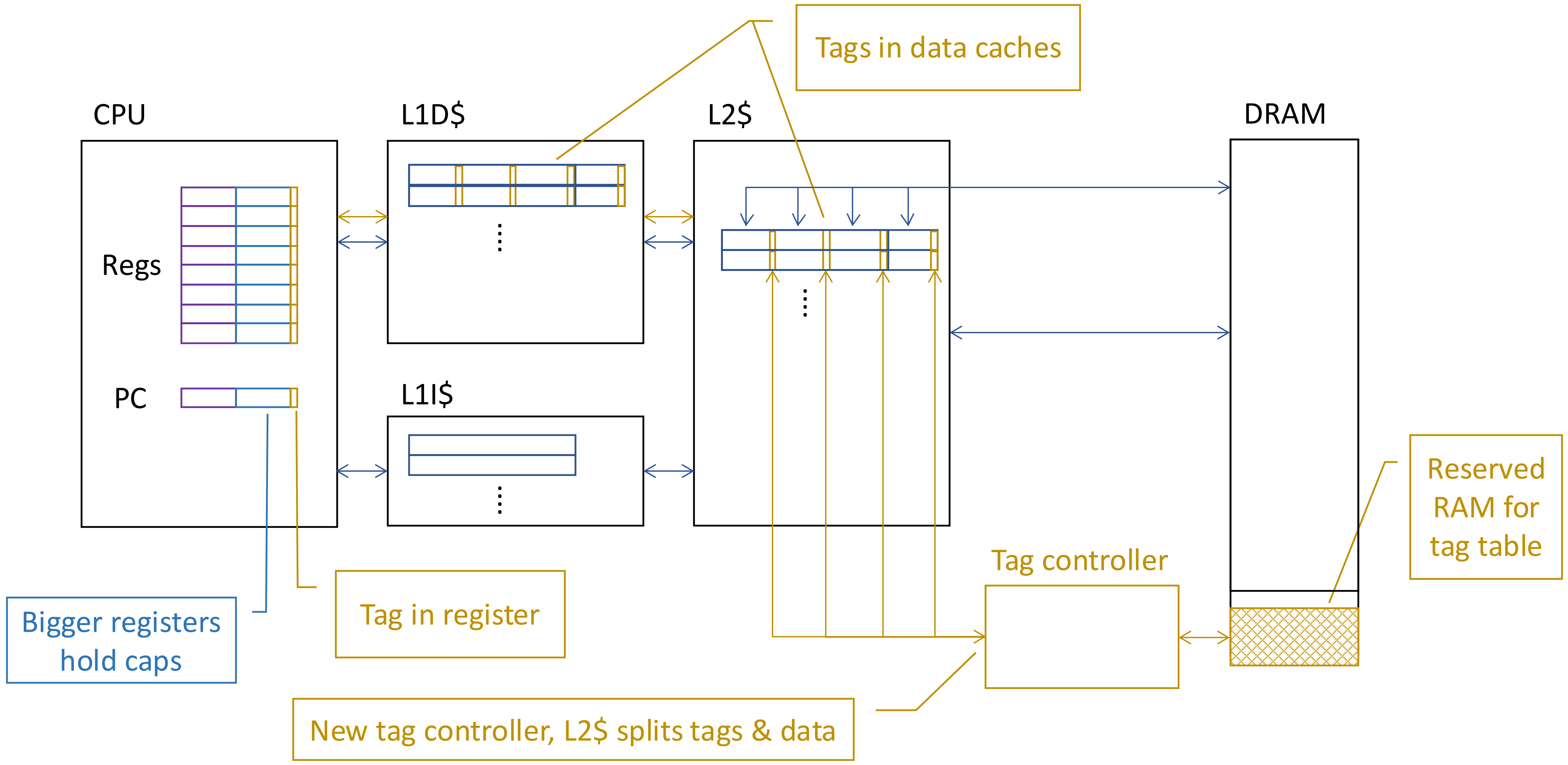


Figure 3.2: Graphical representation of memory regions encoded by CHERI Concentrate. The example addresses on the left are for a `0x6000`-byte object located at `0x1E000`; the representable region extends `0x2000` below the object's base and `0x8000` above the object's limit.

CHERI Instruction-Set Architecture (Version 9).

## CHERI Tags in Cores and Caches


Tags in data caches
CPU
L1D$
L2$
DRAM
Regs
PC
L1I$
Tag controller
Reserved RAM for tag table
Bigger registers hold caps
Tag in register
New tag controller, L2$ splits tags & data

Joannou et al. *Efficient Tagged Memory*. (ICCD 2017)

# CheriABI: Spatially Safe UNIX Processes

## Discussion: `read()` and capability bounds

```
read(fd, lower, sizeof(lower) + sizeof(upper))
```

**RISC-V Baseline**

```
Write OK
lower=0x80922400 upper=0x80922410
Read 0x20 OK; lower[0]=0x10 upper[0]=0x20
```

Kernel overwrite!

**CHERI-RISC-V**

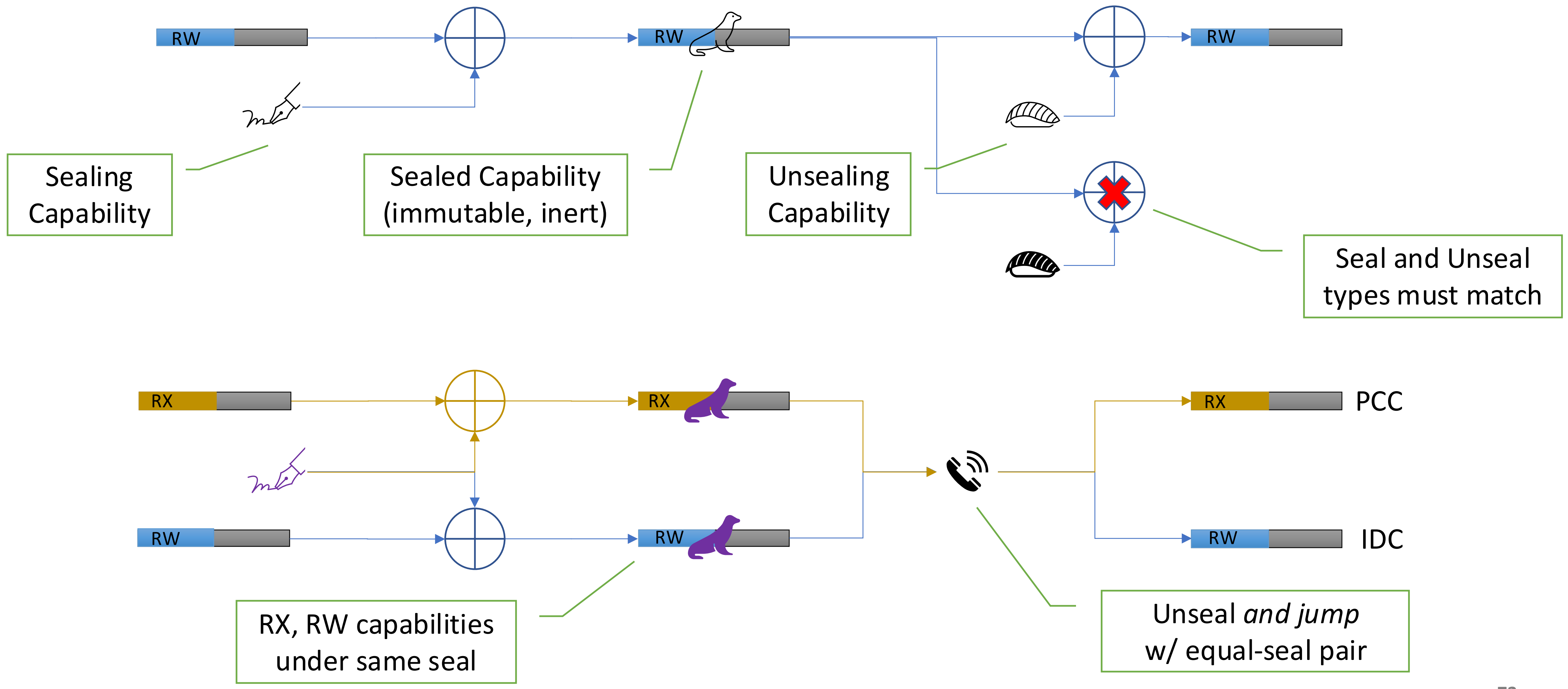
```
Write OK
lower=0x3fffdfff28 upper=0x3fffdfff38
Bad read (Bad address); lower[0]=0x10 upper[0]=0x0
```

Kernel return –EFAULT;
Does not write OOB

Fault detected during copy-out

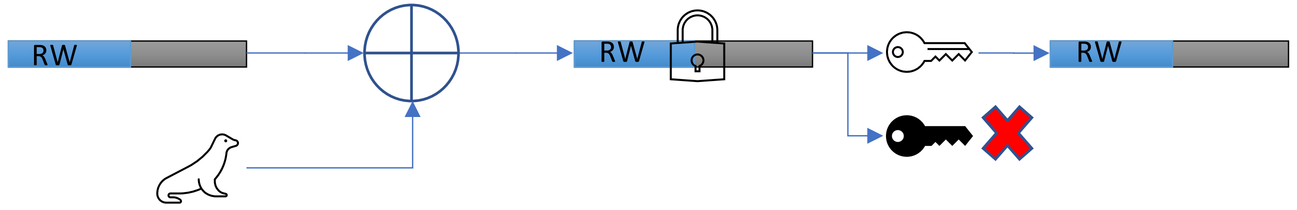
CheriABI system calls take capabilities, and *voluntarily act with implied restricted authority!*

## Sealed and Sealing Capabilities

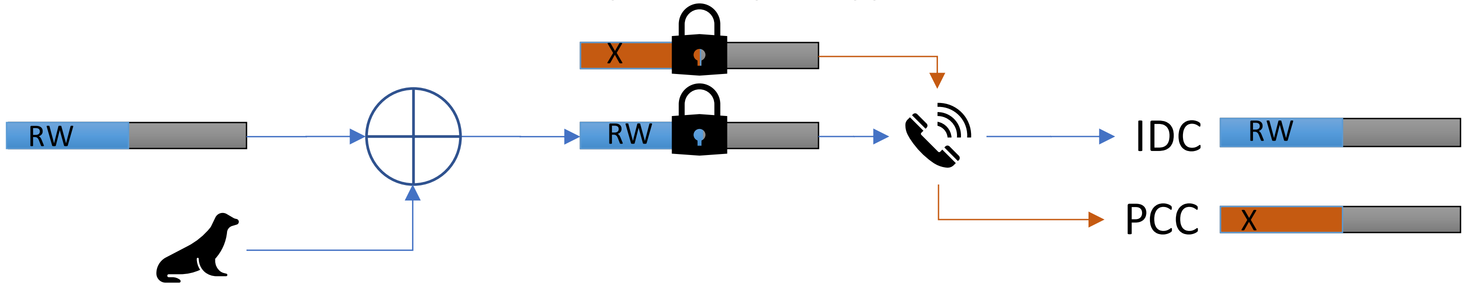

RW
RW
RW
Sealing Capability
Sealed Capability (immutable, inert)
Unsealing Capability
Seal and Unseal types must match
RX
RX
RX
PCC
RW
RW
RW
IDC
RX, RW capabilities under same seal
Unseal *and jump* w/ equal-seal pair

## Advanced Topics
## Sealed and Sentry Capabilities

- Sealing and Explicit Unsealing:
  - Sealed capabilities' authority cannot be exercised until unsealed
  - Seals come in multiple types; must have appropriate *type*-capability to seal and/or unseal
  - Intended uses include RTTI checks and for inter-compartment references



- Can unseal by `CInvoke`: sealed code and data caps of equal type; code becomes PCC, data IDC:



- CHERI also defines some flavors of "sentry" ("sealed entry") capabilities which unseal in jumps:
  - Single capability, becomes PCC when unsealed – useful for function entry, return addresses
  - Pointer to PCC, becomes IDC when unsealed, PCC loaded from target – "pointer to intrusive vtable"
  - Pointer to pair, PCC and IDC loaded – "proxy for method and instance"

## CheriBSD Code Changes

| Area | Files total | Files modified | % files | LoC total | LoC changed | % LoC |
|---|---|---|---|---|---|---|
| Kernel | 11,861 | 896 | 7.6 | 6,095k | 6,961 | **0.18** |
| • Core | 7,867 | 705 | 9.0 | 3,195k | 5,787 | **0.18** |
| • Drivers | 3,994 | 191 | 4.8 | 2,900k | 1,174 | **0.04** |
| Userspace | 16,968 | 649 | 3.8 | 5,393k | 2,149 | **0.04** |
| • Runtimes (excl. libc++) | 1,493 | 233 | 15.6 | 207k | 989 | **0.48** |
| • libc++ | 227 | 17 | 7.5 | 114k | 133 | **0.12** |
| • Programs and libraries | 15,475 | 416 | 2.7 | 5,186k | 1,160 | **0.02** |

**Notes:**

- Numbers from cloc counting modified files and lines for identifiable C, C++, and assembly files
- Kernel includes changes to be a hybrid program and most changes to be a pure-capability program
  - Also includes most of support for CHERI-MIPS, CHERI-RISC-V, Morello
  - Count includes partial support for 32 and 64-bit FreeBSD and Linux binaries.
  - 67 files and 25k LoC added to core in addition to modifications
  - Most generated code excluded, some existing code could likely be generated

## Clang/LLVM/LLD Code Changes

| Area | Files total | Files modified | % Files | LoC total | LoC changed | % LoC |
|---|---|---|---|---|---|---|
| LLVM | 4220 | 44 | 1.0 | 1656k | 217 | **0.013** |
| Clang* | 1593 | 30 | 1.9 | 911k | 190 | **0.021** |
| LLD | 249 | 5 | 2.0 | 67.8k | 30 | **0.044** |
| Total | 6062 | 79 | 1.3 | 2365k | 432 | **0.018** |

**Notes:**

- Changes predominantly (u)intptr_t vs size_t/ptrdiff_t confusion, static_asserts about struct sizes/layouts no longer true with 128-bit pointers, and a few instances of using uint64_t for pointers
- Able to compile and link a pure-capability C hello world natively on CHERI-RISC-V
- (*) One outstanding known issue in the frontend prevents compiling a C++ hello world
  - Implementation and header files in question only total an additional 193 lines, or 0.021%, as a worst-case upper bound
- Just over half the Clang changes (99 LoC) are for its bytecode-based C++ constexpr interpreter

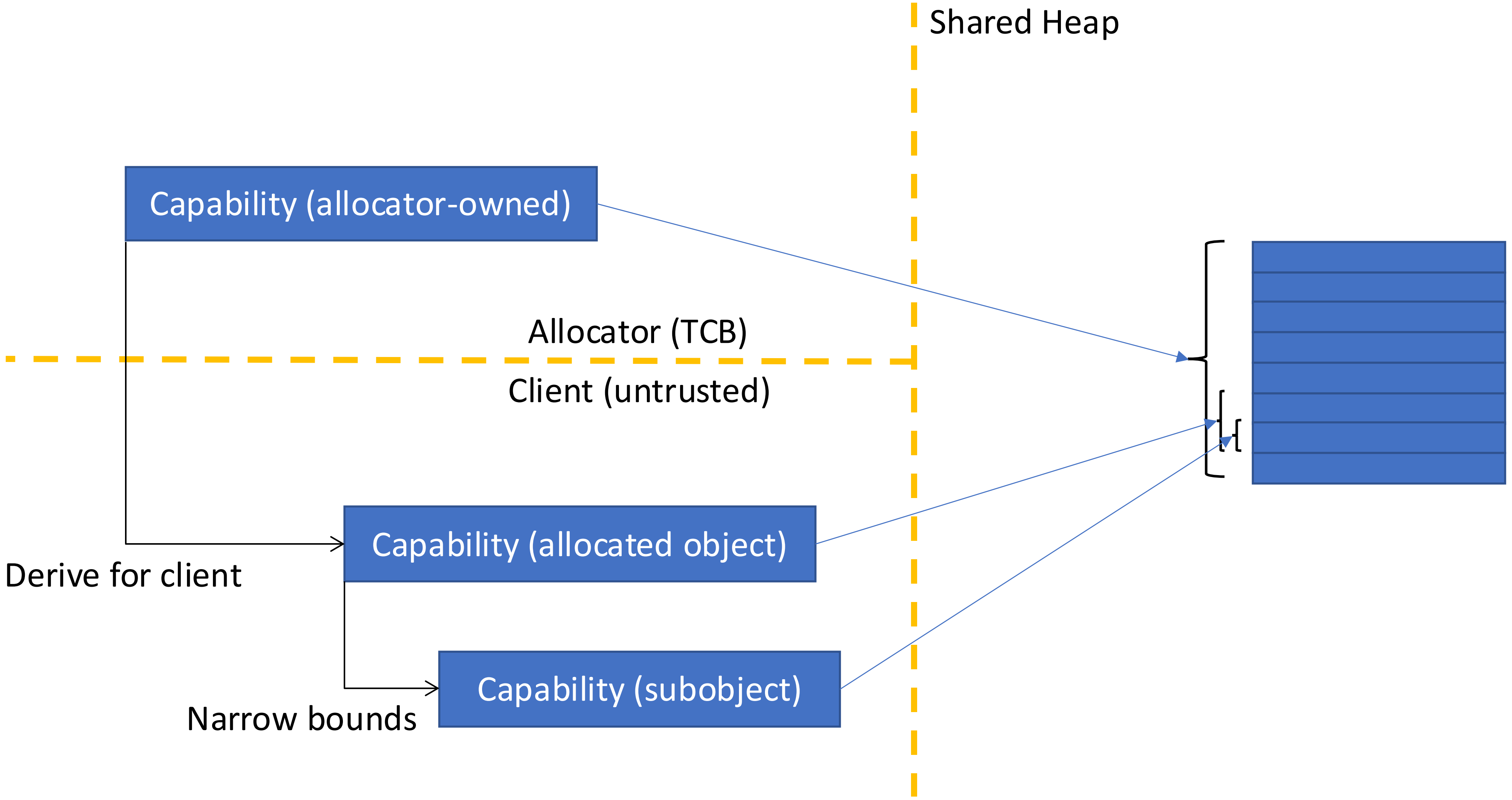
## WebKit - JSC Code Changes

| Area | Files total | Files modified | % Files | LoC total | LoC changed | % LoC |
|---|---|---|---|---|---|---|
| JSC-C | 3368 | 148 | 4.4 | 550k | 2217 | **0.40** |
| JSC-JIT | 3368 | 339 | 10.1 | 550k | 7581 | **1.38** |

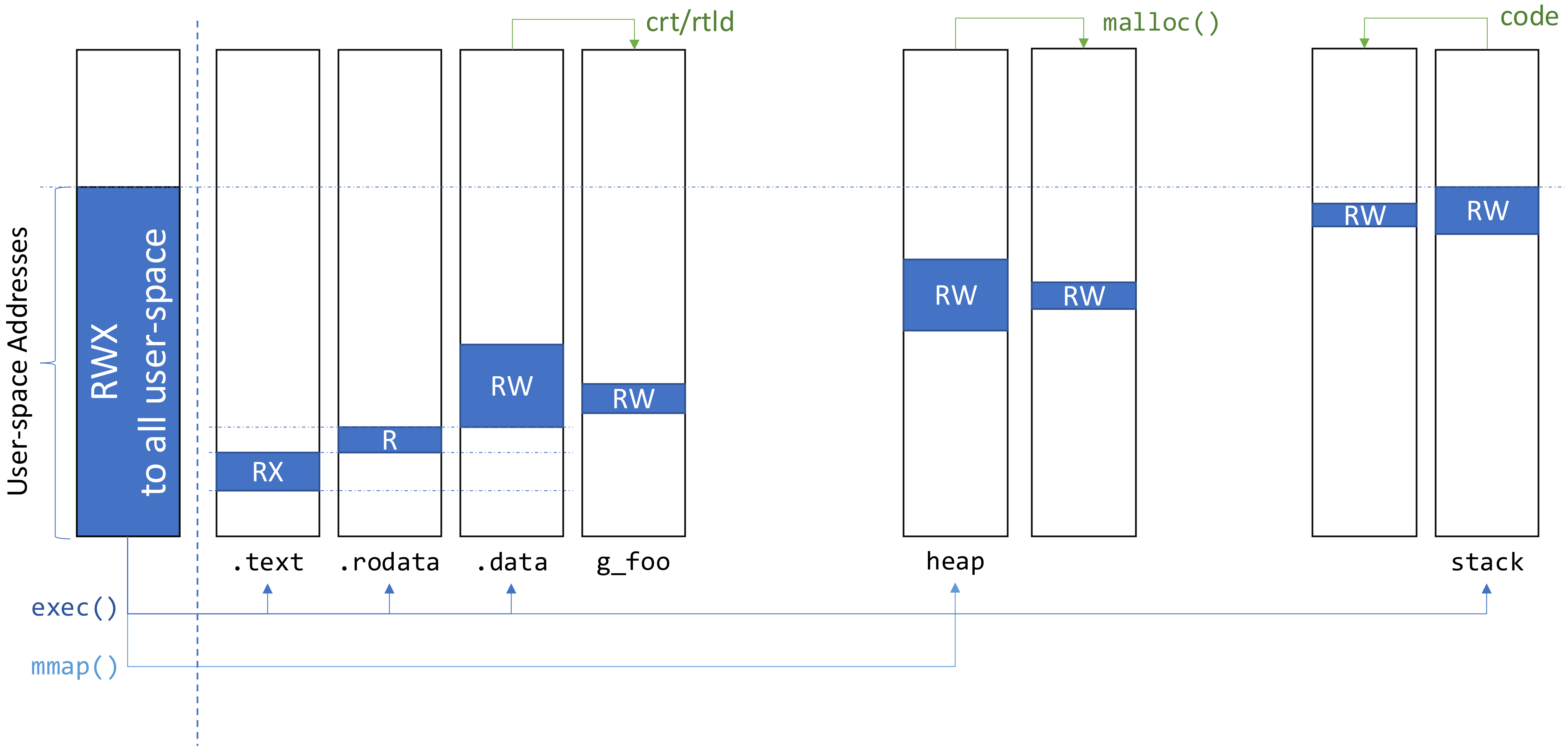
**Notes:**

- JSC-C is a port of the C-language JavaScriptCore interpreter backend
- JSC-JIT includes support for a meta-assembly language interpreter and JIT compiler
- Runs SunSpider JavaScript benchmarks to completion
- Language runtimes represent worst-case in compatibility for CHERI
  - Porting assembly interpreter and JIT compiler requires targeting new encodings
- Changes reported here did not target diff minimization
  - Prioritized debugging and multiple configurations (including integer offsets into bounded JS heap) for performance and security evaluation
  - Some changes may not be required with modern CHERI compiler

## Heap Allocator & Spatial Safety (Montonicity)



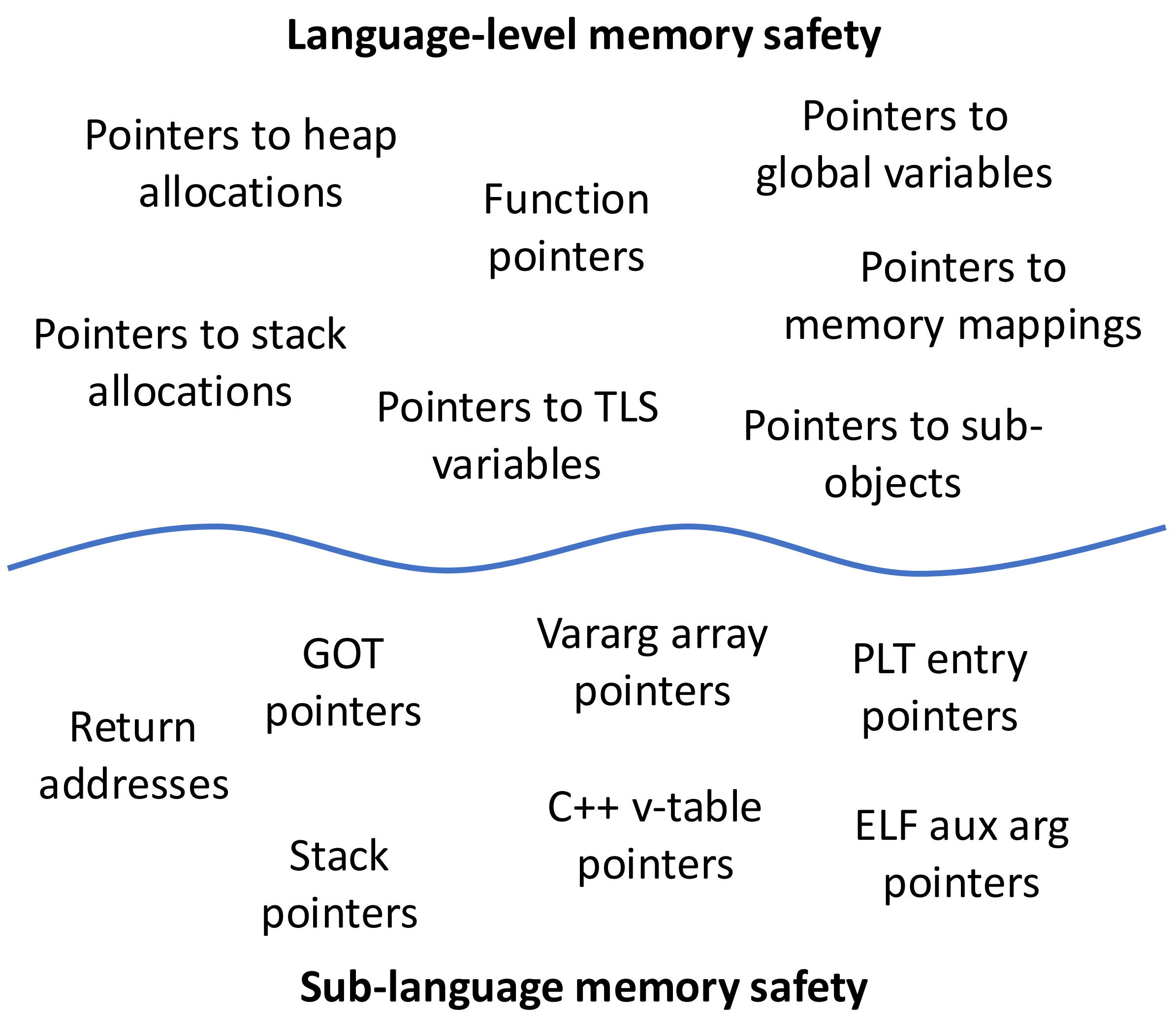
## Advanced Topics
## CheriABI (2/2)



## CheriABI

CHERI Memory Capabilities Meet *NIX

## Compiling C to CHERI

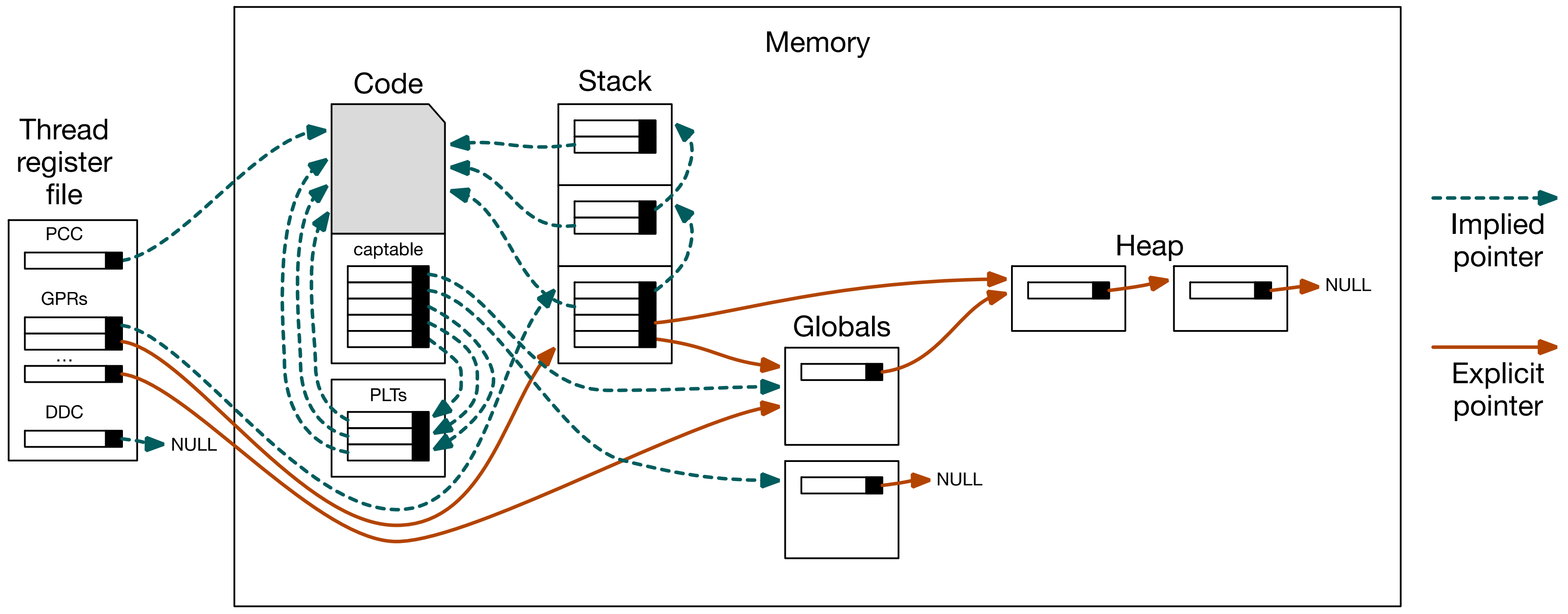

Language-level memory safety
Pointers to heap allocations
Function pointers
Pointers to global variables
Pointers to memory mappings
Pointers to stack allocations
Pointers to TLS variables
Pointers to sub-objects
Return addresses
GOT pointers
Vararg array pointers
PLT entry pointers
Stack pointers
C++ v-table pointers
ELF aux arg pointers
Sub-language memory safety

- CHERI capabilities used for both
  - **Language-level** pointers visible in source program
  - **Implementation** pointers *implicit* in source
- *Compiler* generates code to
  - bound address-taken stack allocs & sub-objects
  - build caps for vararg arrays
- *Loader* builds capabilities to globals, PLT, GOT
  - Derived from kernel-provided roots
  - Bounds applied during reloc processing
- Small changes to C semantics!
  - `intptr_t`, `vaddr_t`
  - `memmove()` preserves tags
  - Pointers have single provenance
  - Integer ↔ pointer casts require some care

See CHERI C/C++ Programming Guide.

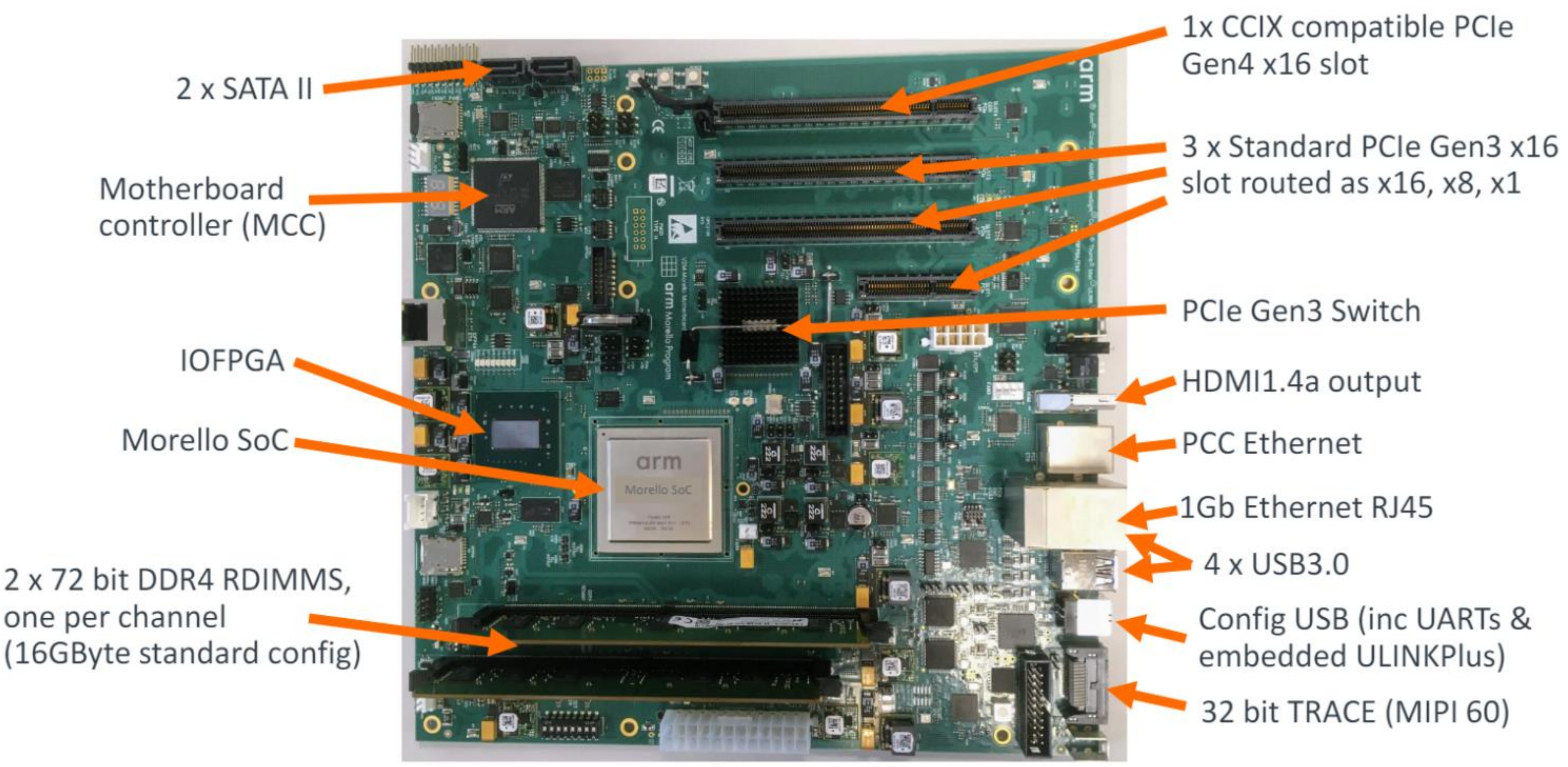
## CheriABI: Spatially Safe *NIX Processes

- Capabilities now implement *all* pointers in a process
- More faithfully captures *program intent* as “objects with links between them”

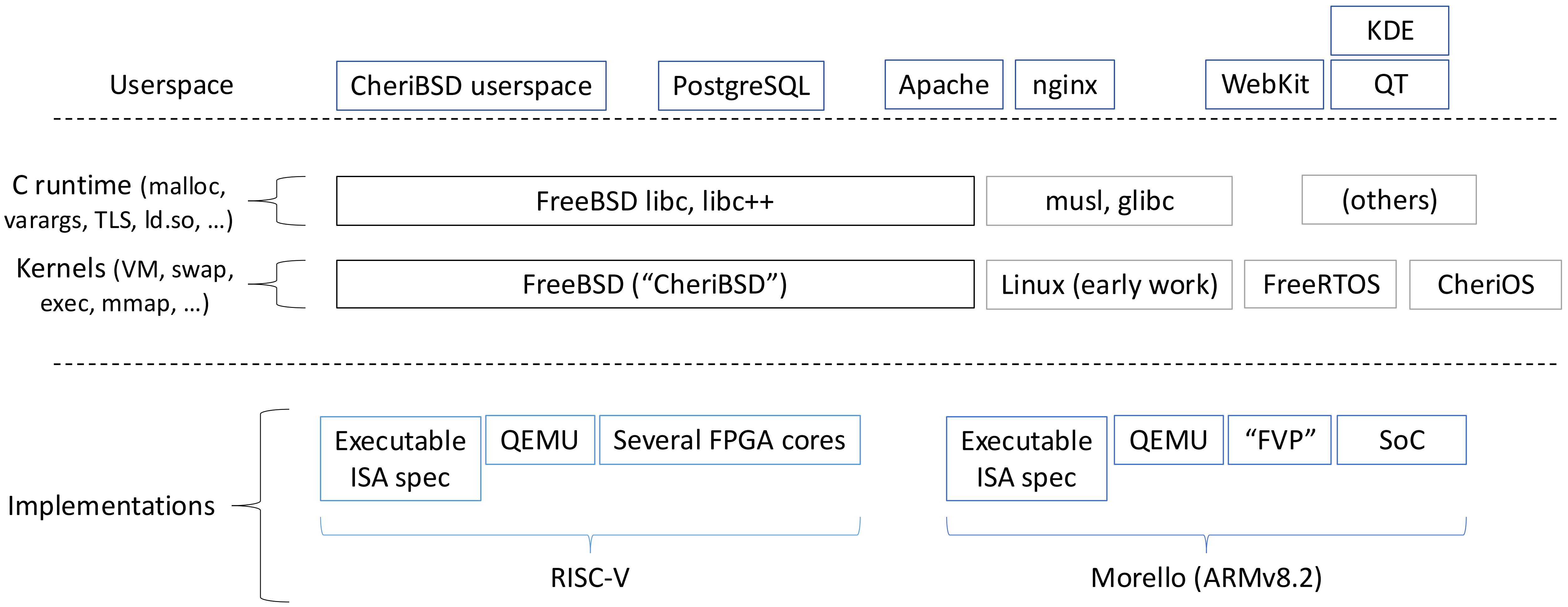


## Morello: An experimental ARMv8 with CHERI

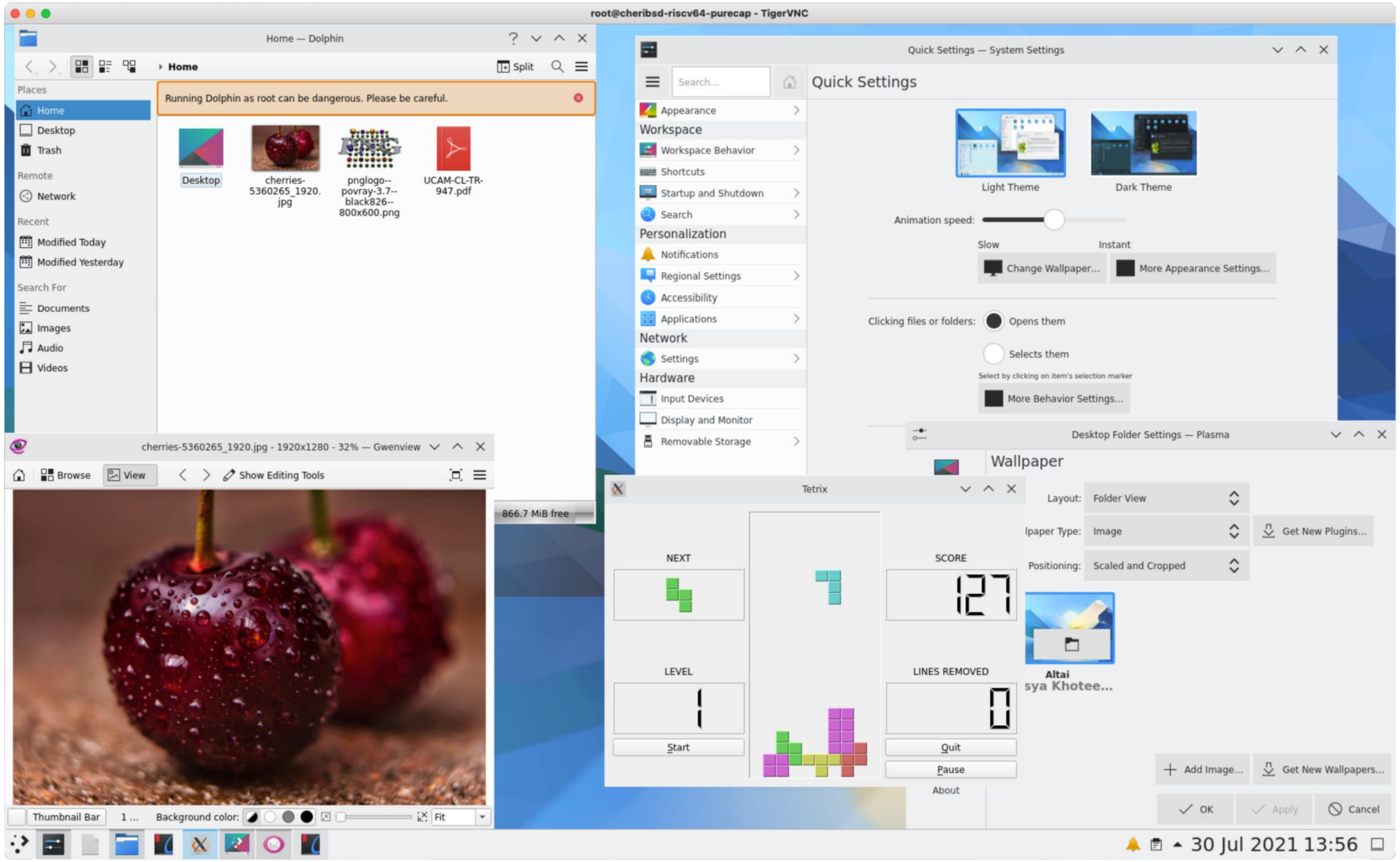
"Morello" prototype SoC & board: 4-core, 2.5-GHz Armv8.2-A w/ CHERI extensions


2 x SATA II
Motherboard controller (MCC)
IOFPGA
Morello SoC
2 x 72 bit DDR4 RDIMMS, one per channel (16GByte standard config)
1x CCIX compatible PCIe Gen4 x16 slot
3 x Standard PCIe Gen3 x16 slot routed as x16, x8, x1
PCIe Gen3 Switch
HDMI1.4a output
PCC Ethernet
1Gb Ethernet RJ45
4 x USB3.0
Config USB (inc UARTs & embedded ULINKPlus)
32 bit TRACE (MIPI 60)
arm
Morello SoC

## CHERI Ecosystem At A Glance



## KDE on CHERI-RISC-V over VNC



Running Dolphin as root can be dangerous. Please be careful.

Animation speed:

Clicking files or folders: [✓ Opens them] Selects them

Layout: Folder View

Wallpaper Type: Image

Positioning: Scaled and Cropped

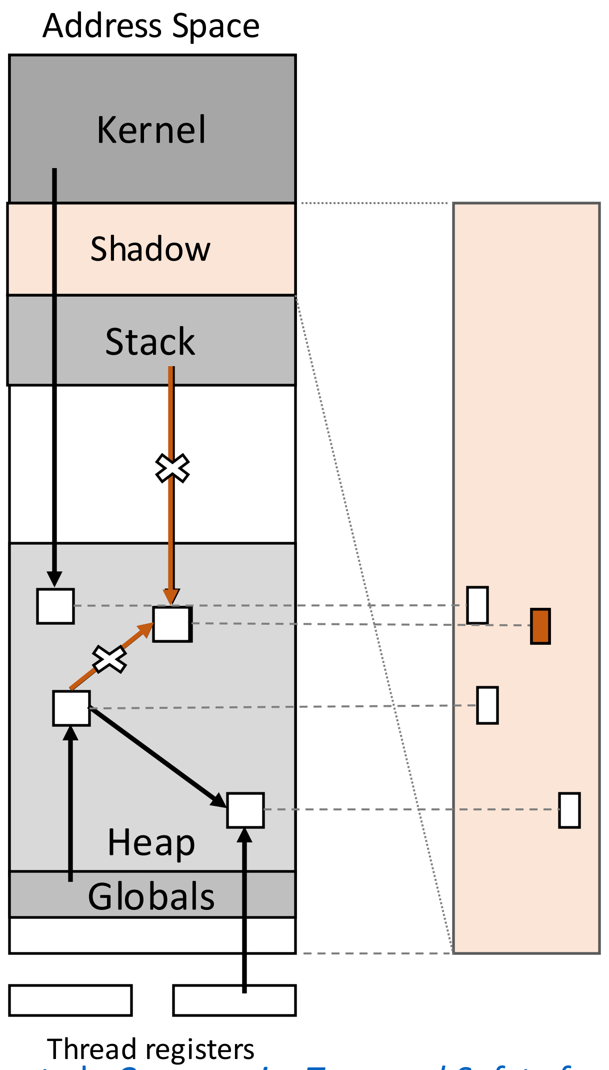
SCORE 127

LEVEL 1

LINES REMOVED 0

# Cornucopia: CHERI Heap Temporal Safety

## Quarantine & Batched Revocation



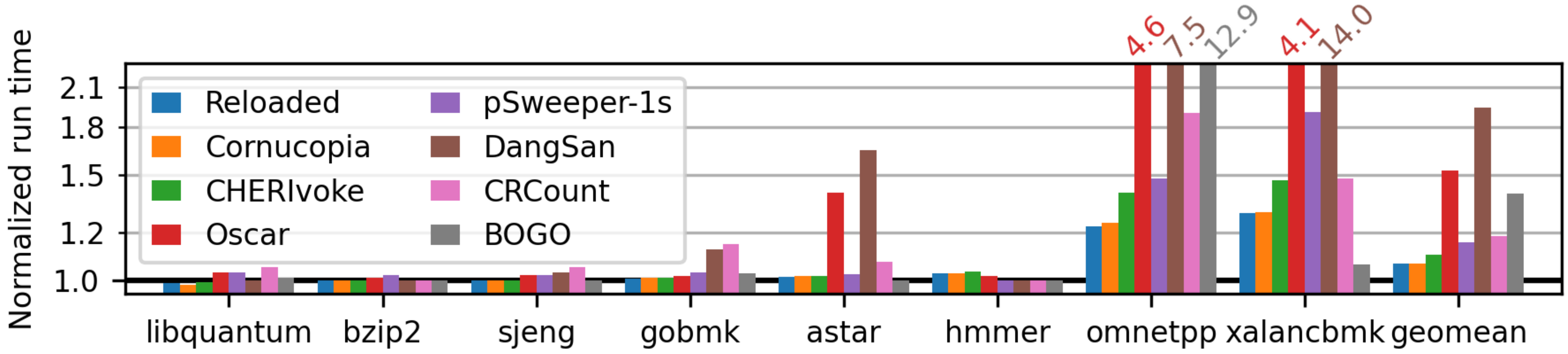
- Kernel offers revocation *service* to user programs
  - Exposes *revocation bitmap*, encodes live/free state of memory.
- On free, allocator…
  - holds address space in *quarantine*
  - *marks* corresponding bits of object
- When quarantine fills, allocator invokes revoker service
  - Deletes all capabilities whose targets have marked revocation bits
- After revocation, safe to reuse address space
  - Allocator *clears* shadow, enqueues address space to free lists

Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

## Sweeping Revocation Performance

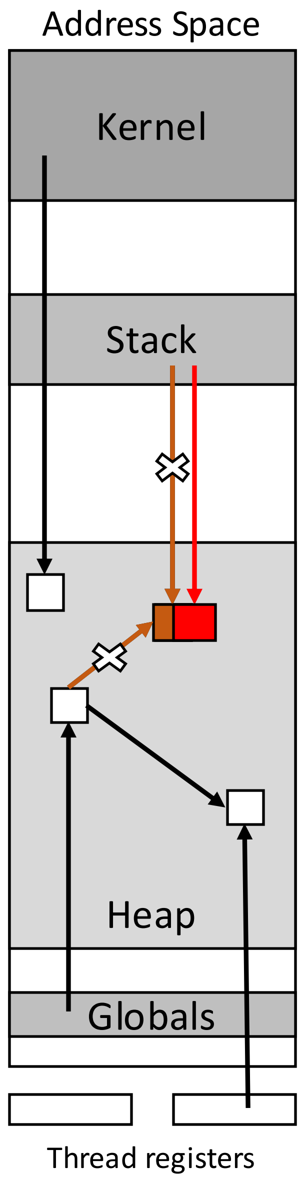
Isn’t checking every capability in the address space horrifically expensive?

- “Cornucopia Reloaded”, SPEC CPU2006 INT, revoke target <33% heap in quarantine, wall-clock overheads on Morello: **<10% geomean, <30% worst case!**



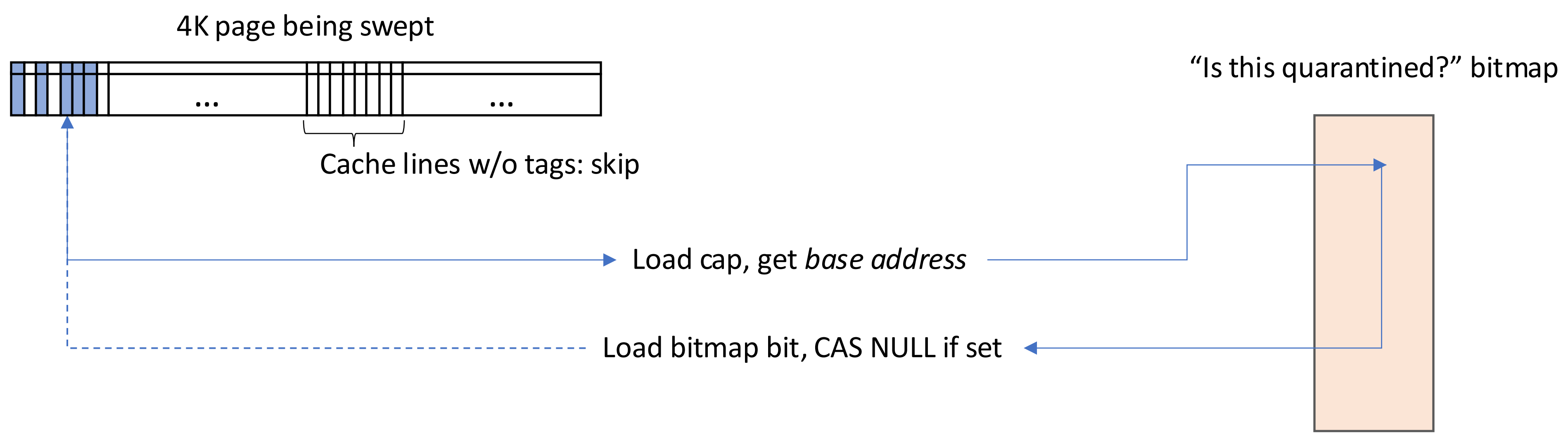
- Key insight: CHERI validity bits *precisely* identify all potential references to memory.
  - Don’t have to guess, and we are justified in *erasing* pointers to quarantine.

## Cornucopia: Heap Temporal Safety Atop CHERI
## Address Space Quarantine, Revocation



- Focused on *heap* temporal safety
  - More complex lifetimes than stack objects, resists static approaches
- Heap pointers end up in globals, stacks, registers, kernel heap, …
- Risk: retain references to `free()` object, overlap new allocation
- Eliminate "use-after-reallocation" by *revoking* dead references
  - UAF still possible, but accesses old object
- Hold address space in *quarantine* to amortize sweep cost
  - Quarantine state held *out of band*
- "Dual" of garbage collection: (lazily) enforce `free()`

## Sweeping Revocation Implementation


4K page being swept
…
…
Cache lines w/o tags: skip
"Is this quarantined?" bitmap
Load cap, get *base address*
Load bitmap bit, CAS NULL if set

## Architectural Acceleration for Revocation

### CHERI Tags identify capabilities

- Don't have to guess; revoker justified in erasing!

### Capability-Dirty PTE Flags

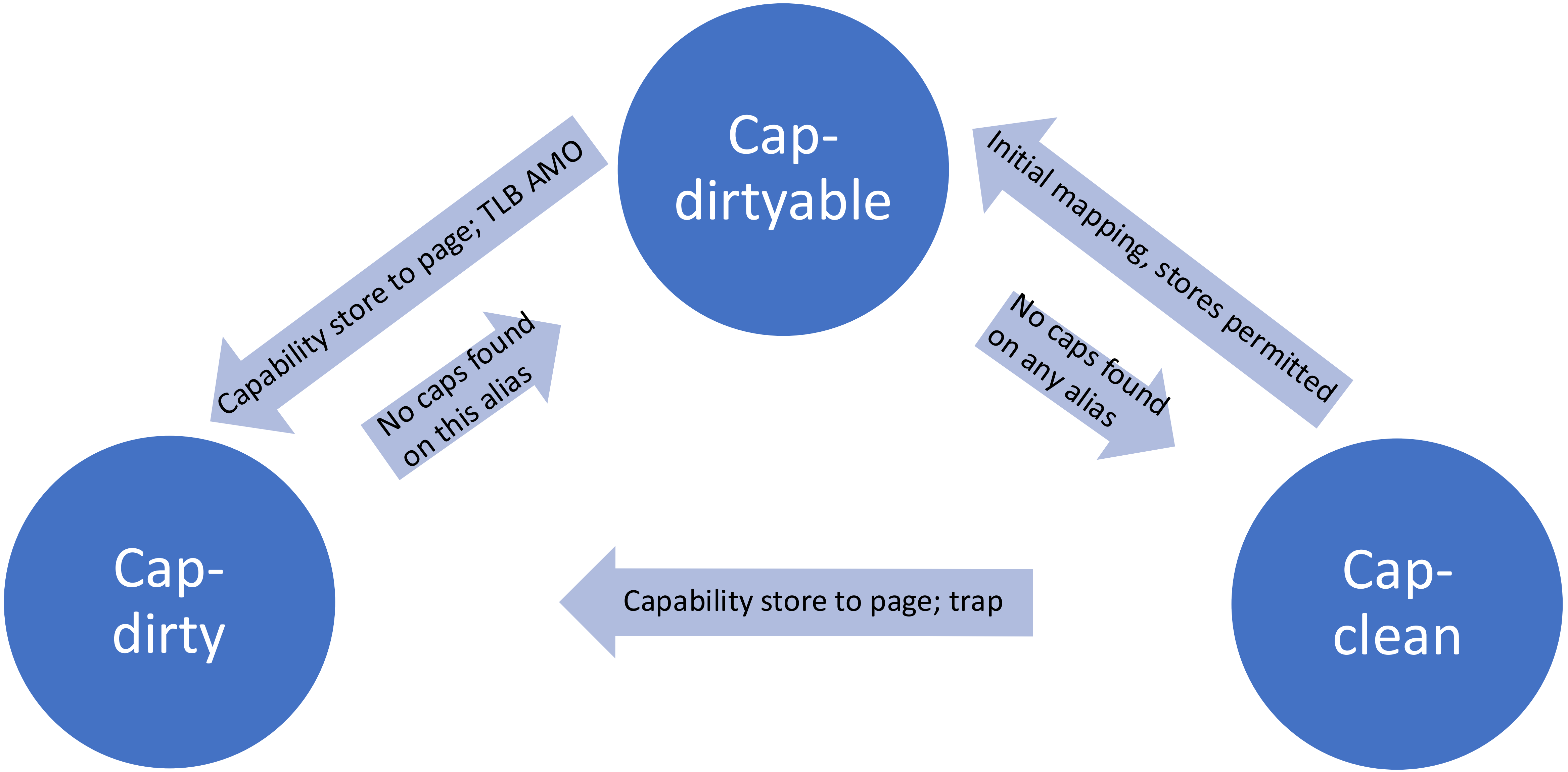
- Set by PTW; skip sweep of pages w/o capabilities

### Capability-Load Trap PTE Flags

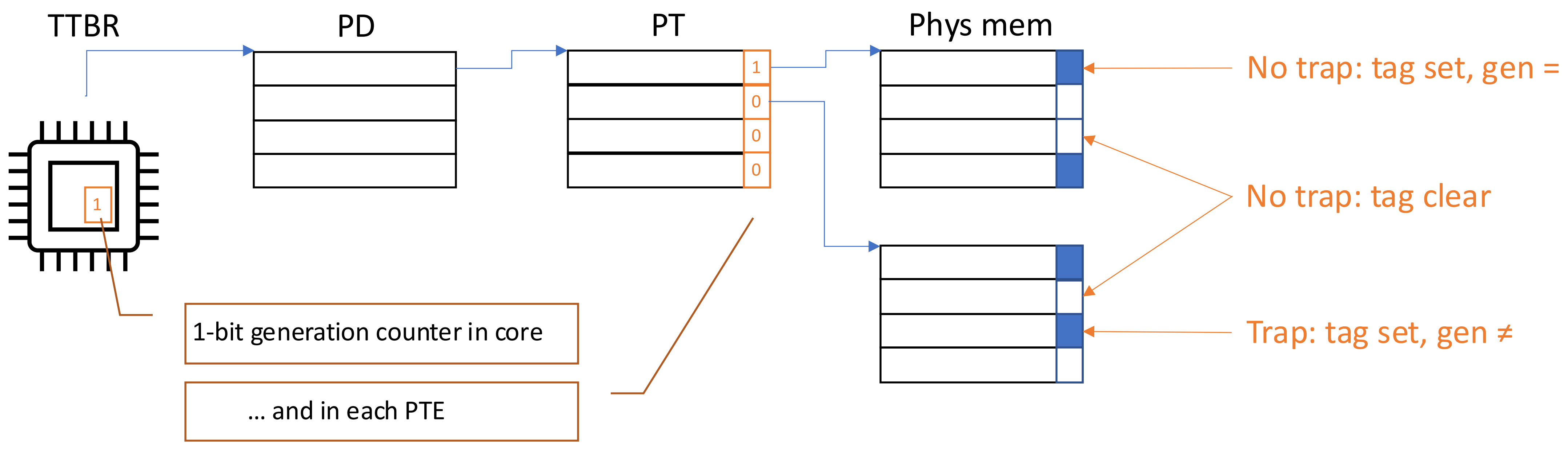
- Cause CPU to trap; revoker scans (WIP)

Filardo et al. *Cornucopia: Temporal Safety for CHERI Heaps*. (Oakland 2020)

## Cornucopia Architecture
## Per-Page "Capability-Dirty" Tracking

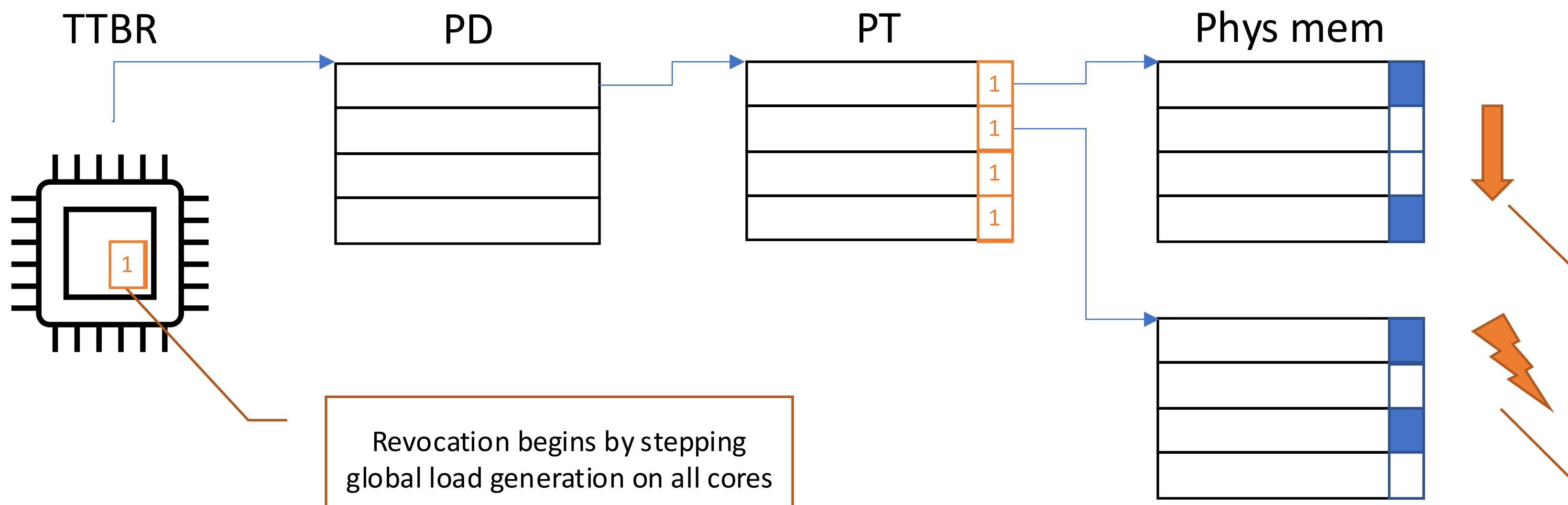

Cap-dirtyable
Cap-dirty
Cap-clean
Capability store to page; TLB AMO
No caps found on this alias
Initial mapping, stores permitted
No caps found on any alias
Capability store to page; trap

## Cornucopia Architecture
## Per-Page Capability Load Generations



Loads trap if (loaded CHERI tag set) and (core gen ≠ source page PTE gen)

## Cornucopia Architecture
## Revoking With Capability Load Generations

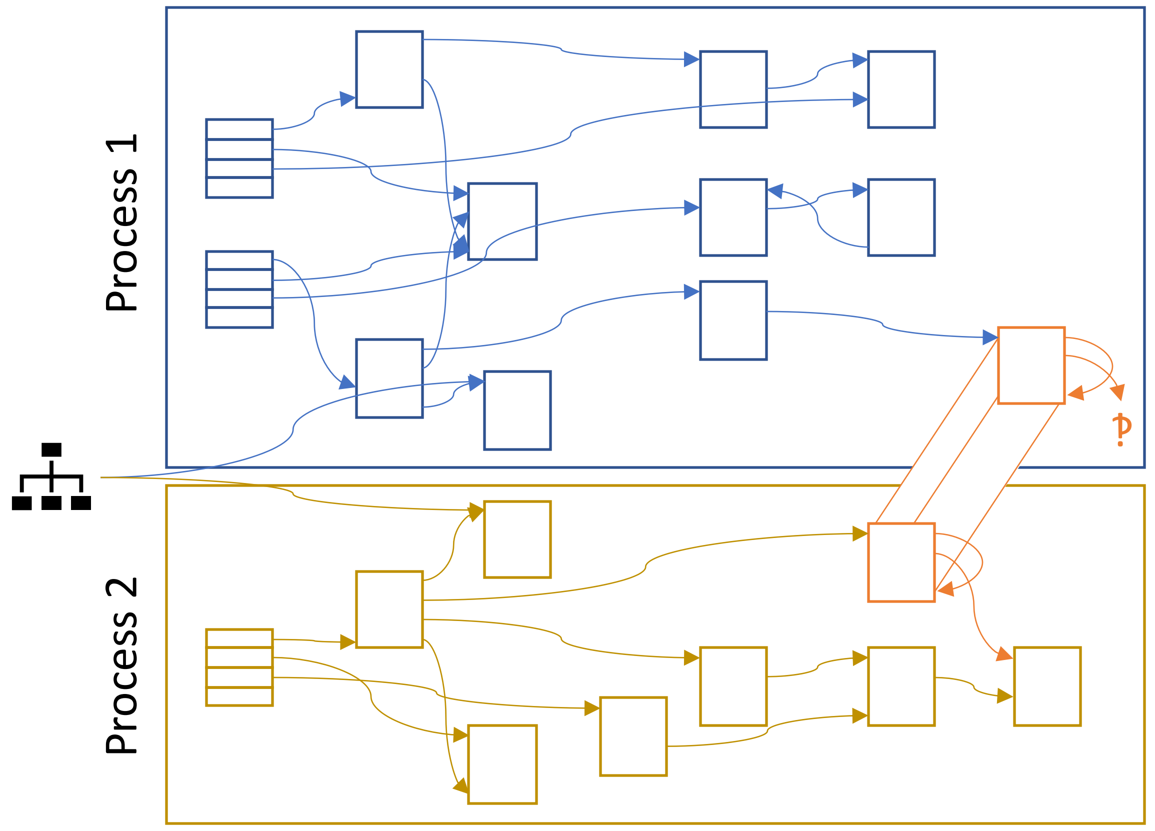


Revocation begins by stepping global load generation on all cores

Background scan visits all pages w/ caps, updates PTE generation

As loads cause traps, sweep per page and update PTE generation

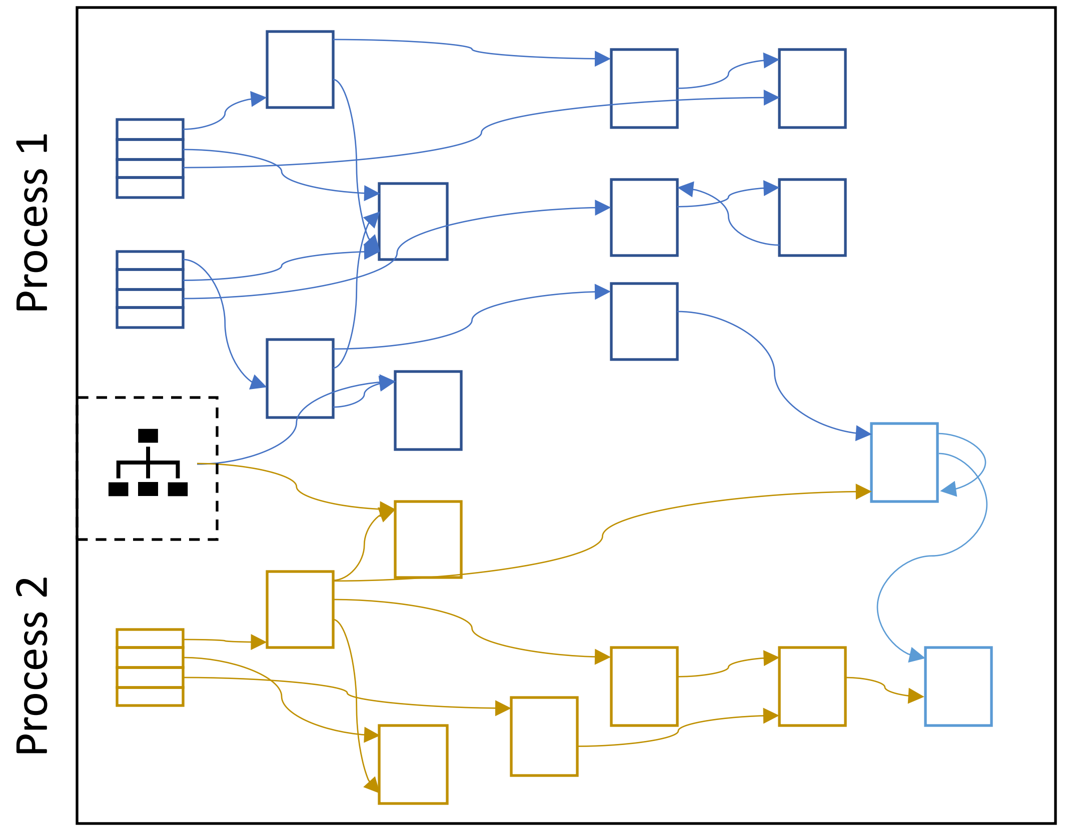
# Research: Colocation: Multiple Processes In One Address Space!



MMU-based isolation & selective sharing

- Programs in separate address spaces
- IPC by context switch
  - Data *copy* by *kernel* (write/read on pipe)
    - Both time and space costs!
  - TLB switching also costs!
    - Flush (time, power) or ASIDs (area, power)
- Selectively *shared pages*
  - Pointers *to* shared memory: fine
  - Pointers *in* shared memory: … carefully
  - Pointers *from* shared memory: WTF‽

## Research: Colocation: Multiple Processes In One Address Space



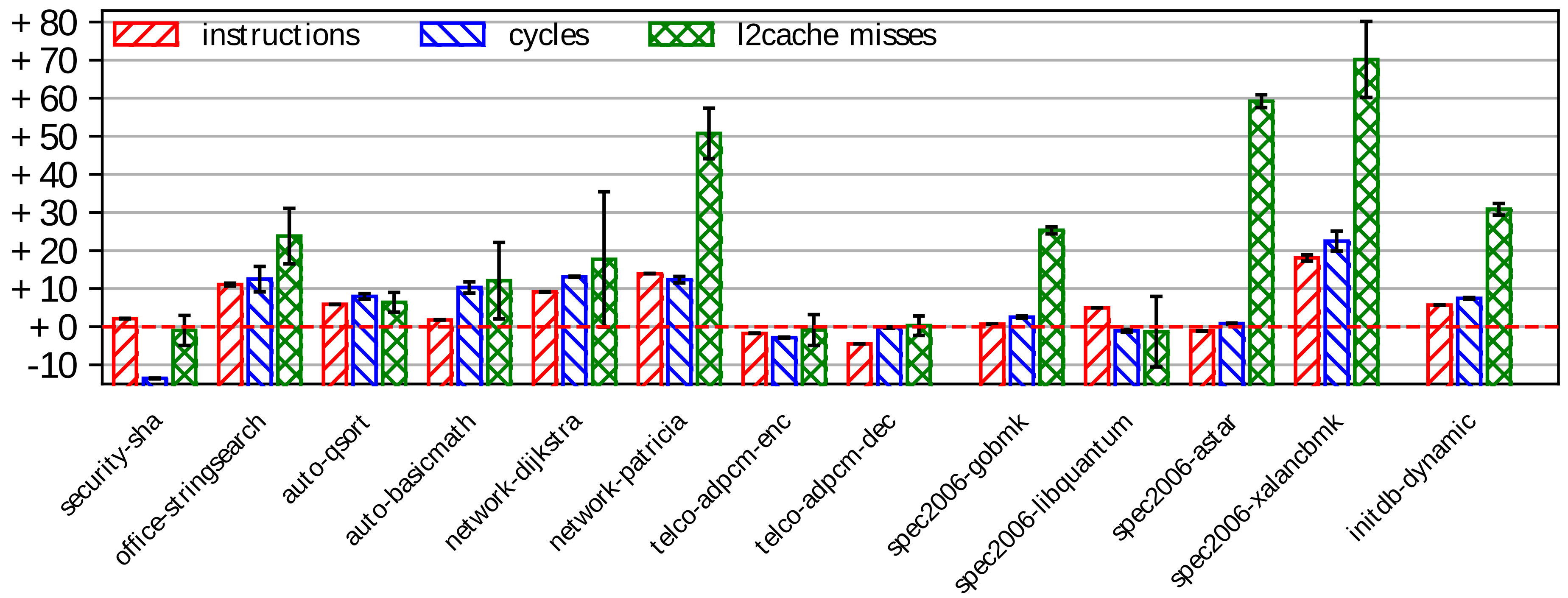
- *Colocated Processes*
  - Many programs in *one* address space
  - Isolation maintained with CHERI
- IPC by *function call* (eliding some details)
  - Can copy on call through "trusted switcher"
- Really fast sharing: pass capability across IPC
  - No *misinterpretation* risk from shared pointers

## Performance Overhead?

Clearly some costs to the story.

- Processor pipeline complexity, new cache “stuff”
  - Still RISC; not X86 levels of complexity.
- Space overheads: tag memory overheads (1/128th of DRAM space)
  - You probably won’t notice the 1% change
- Pointers double in size! Do we need all computers to have 2x as much DRAM??
  - Data still just data! Cute cat videos still mostly just (adorable) bytes.
  - Workload dependent. May be able to *relax* the truly expensive, pointer-heavy cases in interesting ways.
- Fit half as many pointers in each cache line?! Double cache sizes? Line sizes? Bus *frequencies*?
  - Not double, but certainly increase some thing(s) for workloads that need it.

## Performance Overhead Measurements



- As of ASPLOS'19, on CHERI-MIPS CPU in FPGA:
  - 0 - ~10% cycle overheads (= wall clock, here) in most cases
  - Many L2 cache misses for pointer-heavy workloads from increased pointer size
- Detailed report on Morello performance also available; *ample nuance in big, prototype chip*!

  "**1.8% to 3.0%** is our current best estimate of the [geomean] overhead … for a future optimized design"

## CHERI Source Compatibility

| Codebase kind | LoC Changes for CHERI |
|---|---|
| CheriBSD Kernel | 0.2% |
| Low-level runtime libraries | < 0.5% |
| JSC JIT | 1-2% |
| QT, KDE libraries | < 0.1% |
| CLI applications, libraries | ≈ 0.02% |
| QT, KDE applications | < 0.05% |

*DSbD Consortium Update.* (2021/05)

Capabilities Limited. *Assessing the Viability of an Open Source CHERI Desktop Software Ecosystem.* (2021)

## CHERIoT: Scaling CHERI Down to 32-bit Microcontrollers

CHERI scales down to microprocessor environments

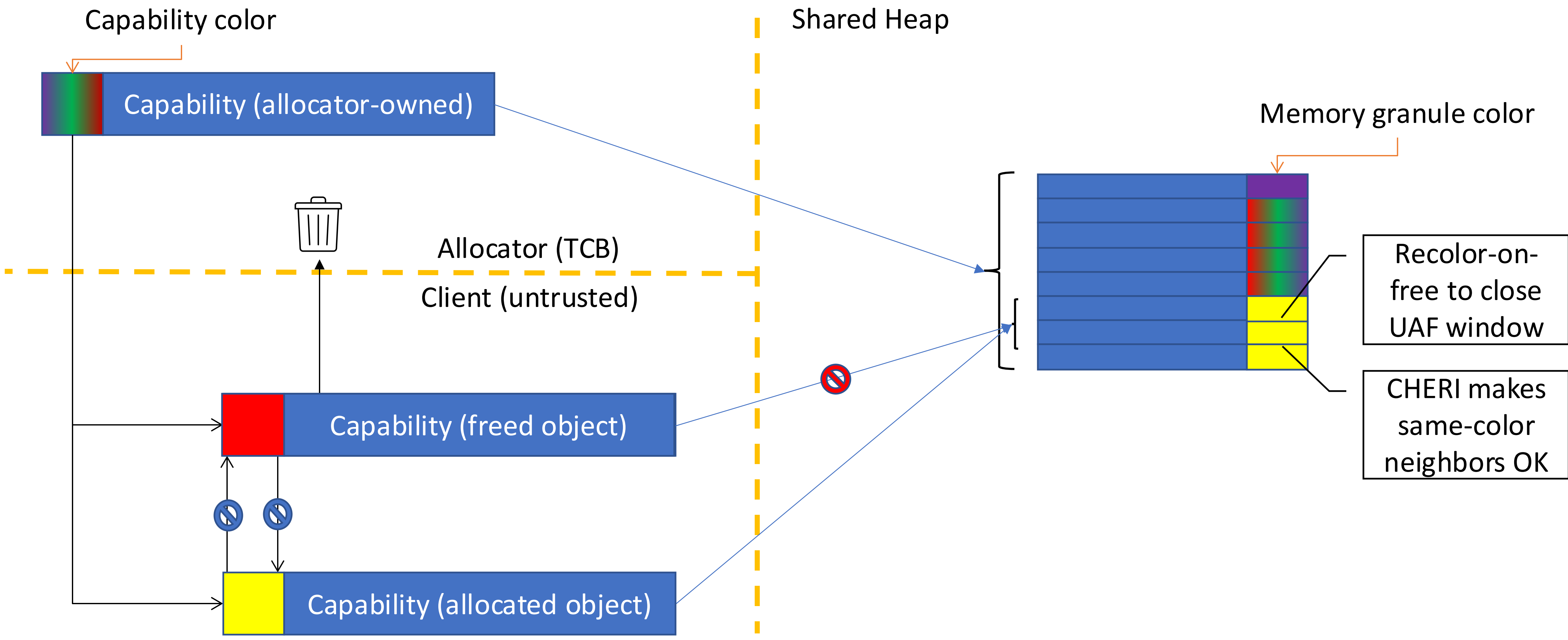
- 32-bit addresses, so 64-bit capabilities

- Designed for *compartmentalized* software: mutually distrusting components, secure communications
  - RTOS more “microkernel” than “kernel”, only essentially trusted component after boot is ~400 instructions.

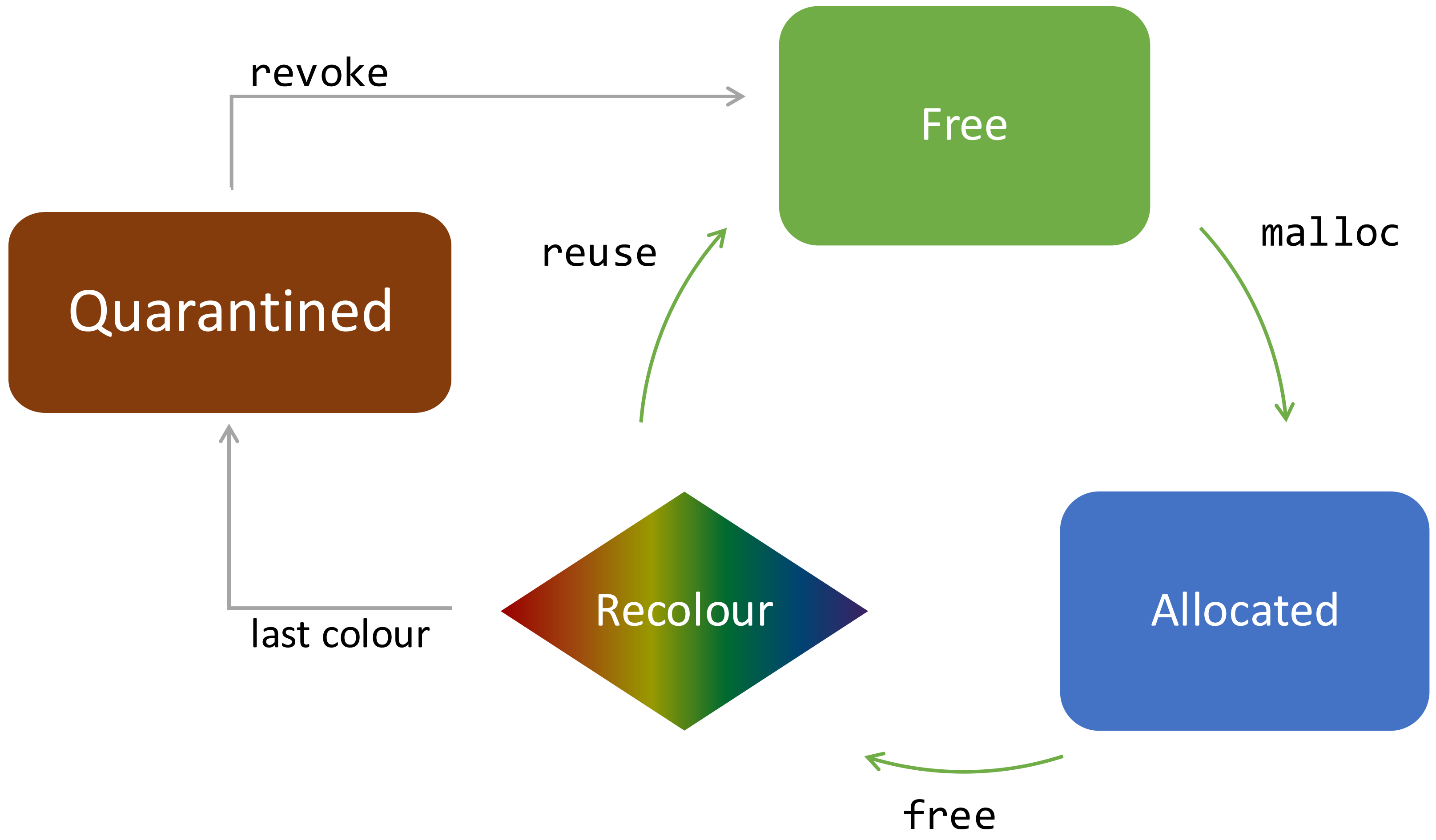
- Takes advantage of small memories:
  - Special permissions for stack capabilities, stack zeroed on cross-compartment call
  - Heap temporal safety baked into the architecture

- Fully open-source research project originally from Microsoft (now https://www.cheriot.org)
  - Formal spec, compiler, emulator, Verilog implementation, RTOS, compartmentalized JS interpreter, …
  - Tape out perhaps as early as next year(!)

# Heap Allocator Use Case


Capability color
Shared Heap
Capability (allocator-owned)
Memory granule color
Allocator (TCB)
Client (untrusted)
Recolor-on-free to close UAF window
CHERI makes same-color neighbors OK
Capability (freed object)
Capability (allocated object)

## Future work: CHERI+MTE Heap Temporal Safety



## Safe Languages?

### C/C++ on old computers

- Spatial and temporal errors lead to arbitrary code execution

### C/C++ on new computers

- Spatial errors fail-stop (and maybe heap temporal errors, too!)

### Ada / Java / C# / TypeScript / ML / Haskell / Rust / …

- Array index errors throw exceptions; other spatial errors impossible*
- Temporal errors impossible*

## Rewrite Everything to be Safe?

- There’s a lot of C, some of it very expensive to have made, and some of it very fast.
- TCB code is *intrinsically unsafe* (sit below safe language abstraction)
  - Memory managers, garbage collector, context switcher, …
- Different safe language runtimes likely view each other as *unsafe*!
- Rewrite *parts* of programs?

## CHERI + (Unsafe) Rust

- Recently, Rust community has been fretting about semantics of unsafe Rust.
  - Compiler transformations threatening correctness
- Recent proposal to use CHERI-like “strict provenance” semantics!
  - No integer-to-pointer casts, trivially “NPVI” semantics
  - Distinguish `usize` from `uaddr` from `uptr`?
  - Integers must be *recombined* with pointers: address from integer but *provenance* from pointer
- Unsafe strict provenance Rust code should be less unsafe on CHERI!