CHERIoT

David Chisnall

Everything in this talk is open source

The ISA specification:
https://github.com/microsoft/cheriot-sail

The reference core:
https://github.com/microsoft/cheriot-ibex

The embedded OS:
https://github.com/microsoft/cheriot-rtos

The compiler (cheriot branch):
https://github.com/CTSRD-CHERI/llvm-project/

IoT

The ‘S’ stands for security

Motivation – IoT and embedded

The IoT ecosystem:

Includes diverse codebases

Mostly unsafe C/C++

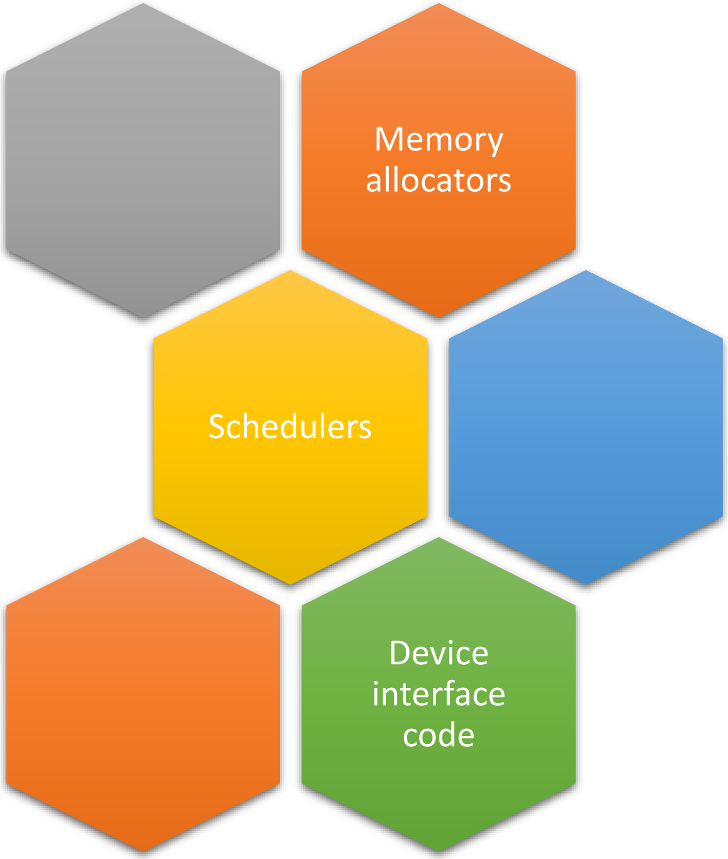
Mitigations are rare

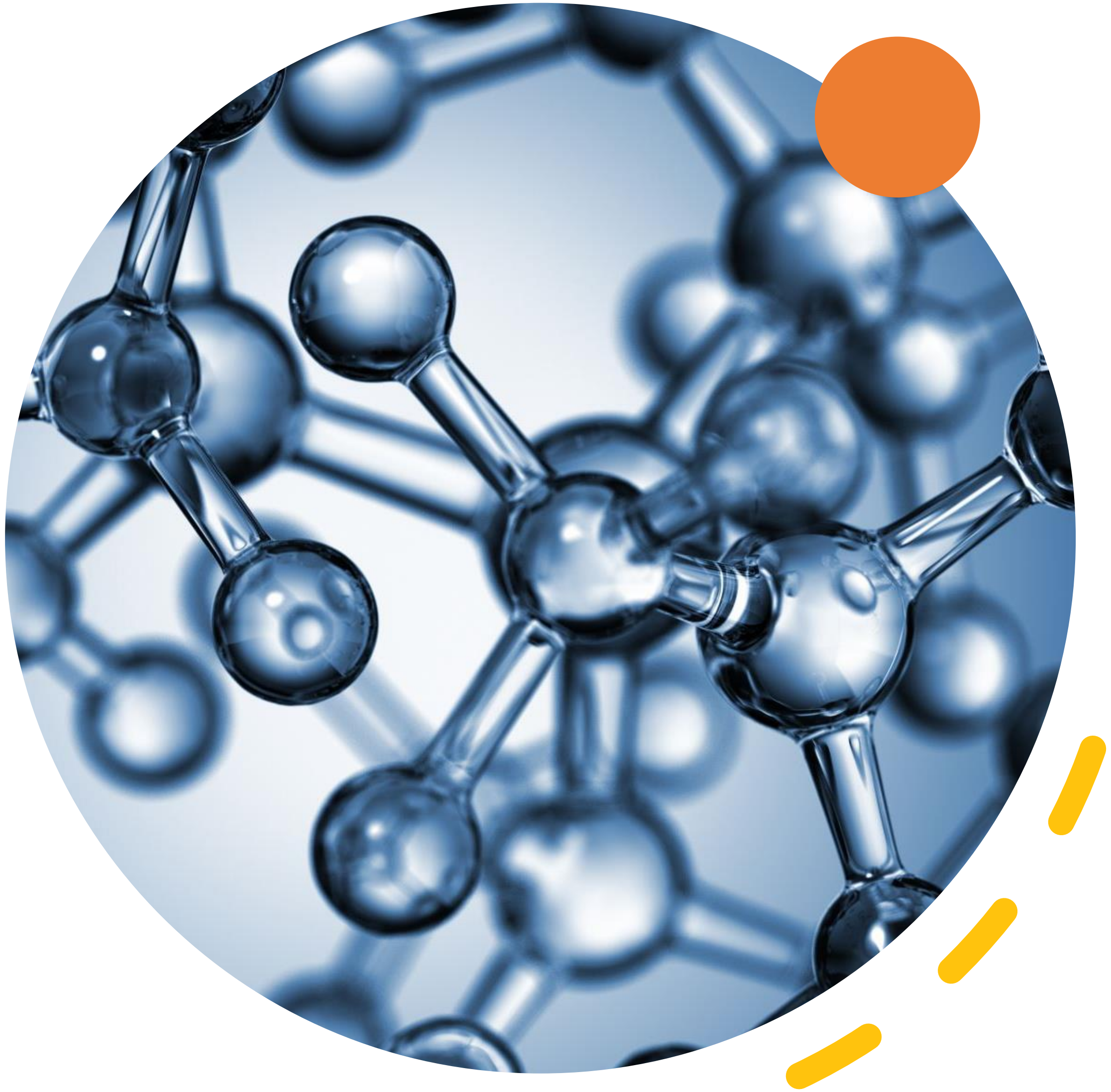
Rewriting has challenges:

Expensive

Talent shortage

Much embedded code is intrinsically unsafe



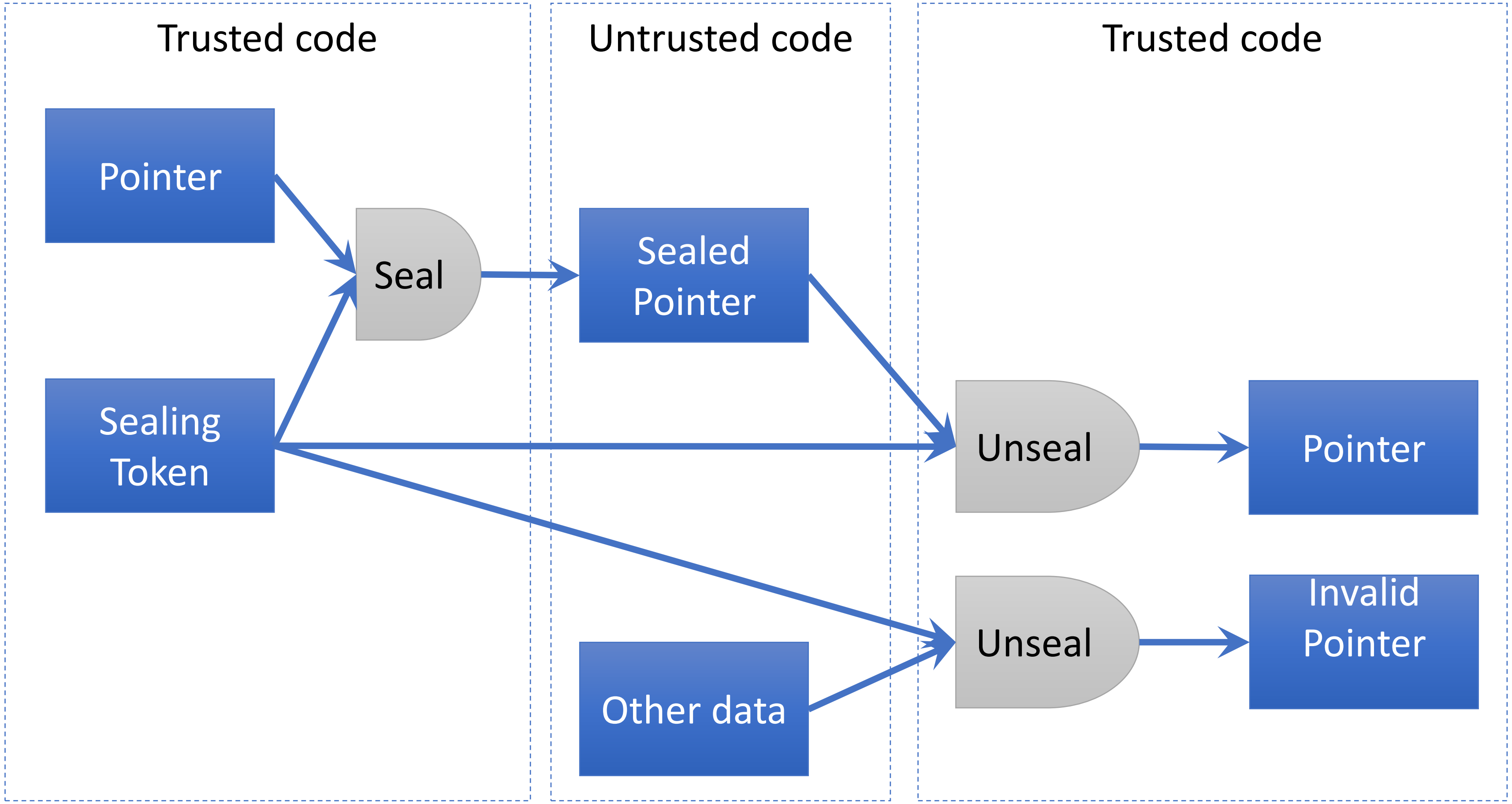


Starting point: CHERI on 64-bit systems

- Hardware knows about pointers
- Pointers can’t be created from thin air
- Pointers carry bounds
- Pointers carry permissions

All memory access instructions require a valid pointer operand

Sealing gives unforgeable opaque tokens



CHERIoT shrinks metadata to 32 bits

Bounds

- No guaranteed out-of-bounds range

Sealing

- Only 3 bits of sealing type
- Separate code and data sealing spaces

Permissions

- 12 permissions in 6 bits

And we add things

Transitive permissions

- Permit-load-mutable, deep immutability
- Permit-load-global, deep no-capture

Interrupt control via function pointers

- Jumping to these enables / disables interrupts

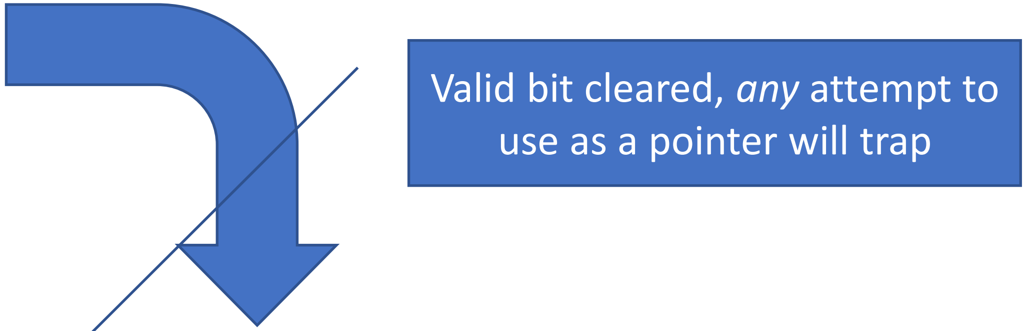
Temporal safety via a hardware revocation bitmap

- 1 bit per 8 bytes in a separate SRAM bank

Hardware load barrier adds temporal safety

- Load pointer computes the base address
- Looks up the corresponding revocation bit
- Invalidates the pointer if the memory is freed

```
void *x = malloc(42);
// Print the allocated value:
Debug::log("Allocated: {}", x);
free(x);
// Print the dangling pointer
Debug::log("Use after free: {}", x);
```



Valid bit cleared, *any* attempt to use as a pointer will trap

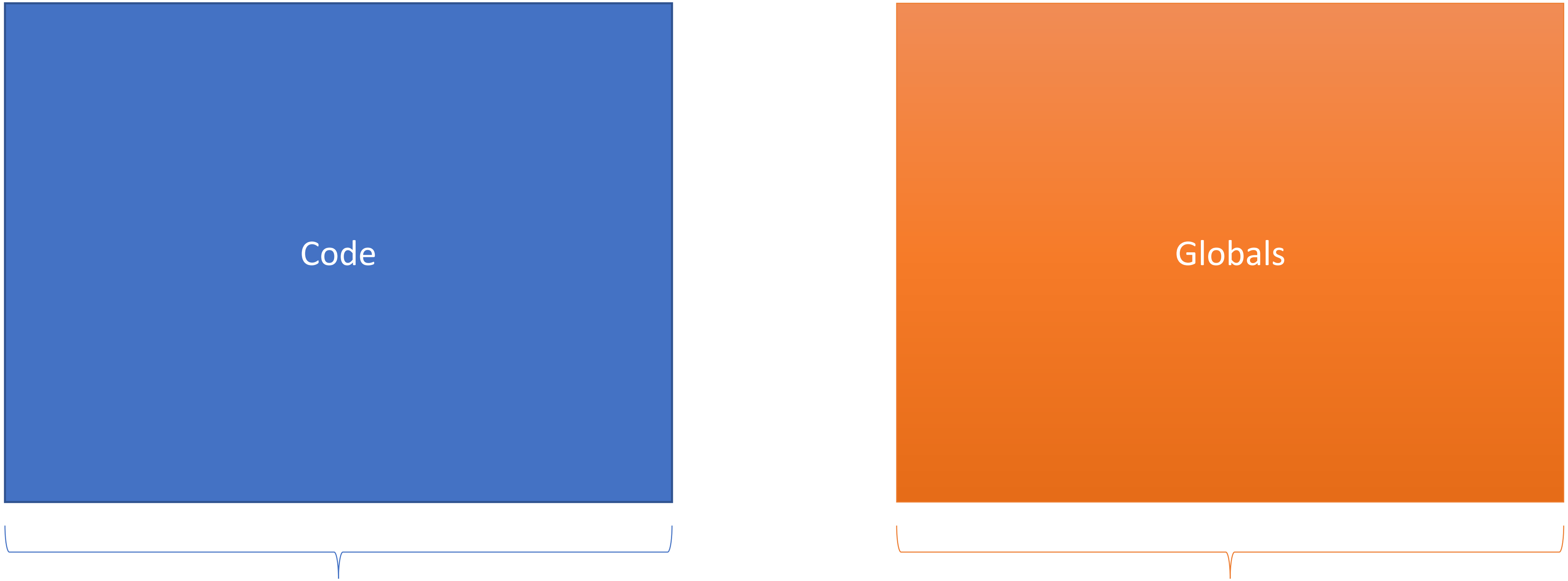
```
Allocating compartment: Allocated: 0x80005900 (v:1 0x80005900-0x80005930 l:0x30 o:0x0 p: G RWcgm- -- ---)
Allocating compartment: Use after free: 0x80005900 (v:0 0x80005900-0x80005930 l:0x30 o:0x0 p: G RWcgm- -- ---)
```

Baseline security guarantees

- No pointer injection
- No bounds violations
- No use after free

The system can assume these for building higher-level abstractions.

Compartments are code and data



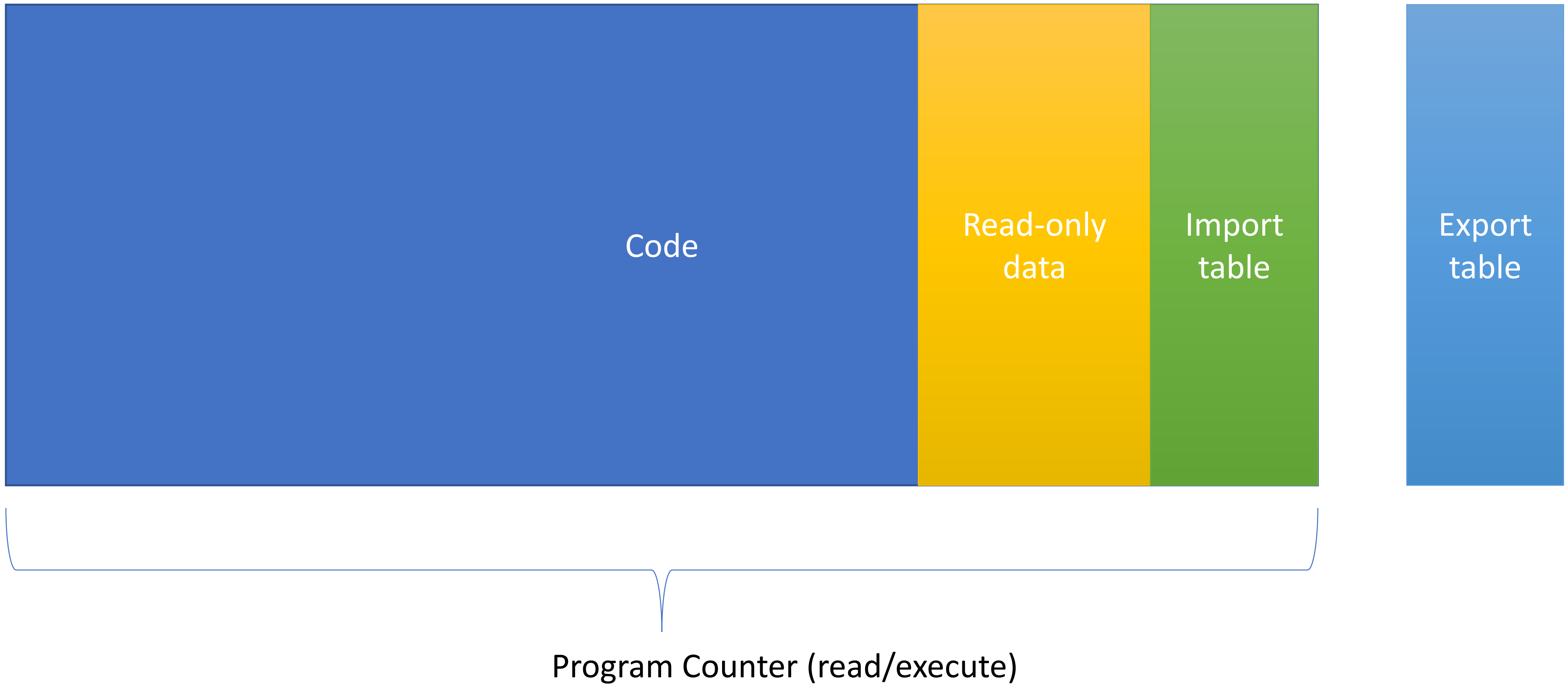
Code

Globals

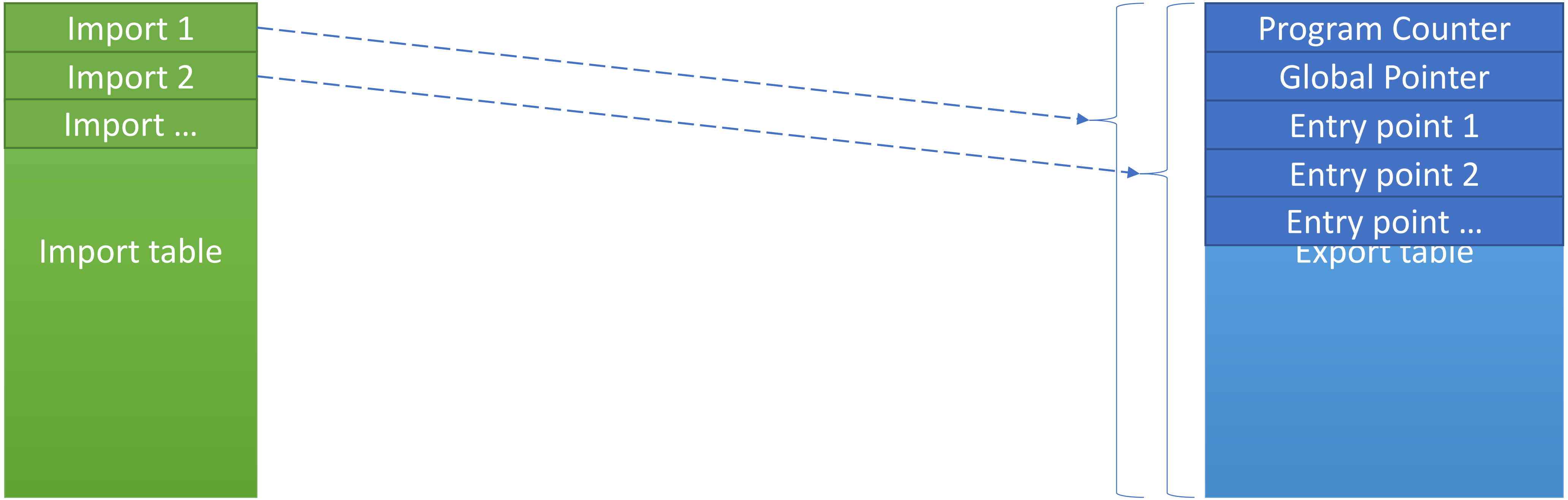
Program Counter (read/execute)

Global Pointer (read/write/global)

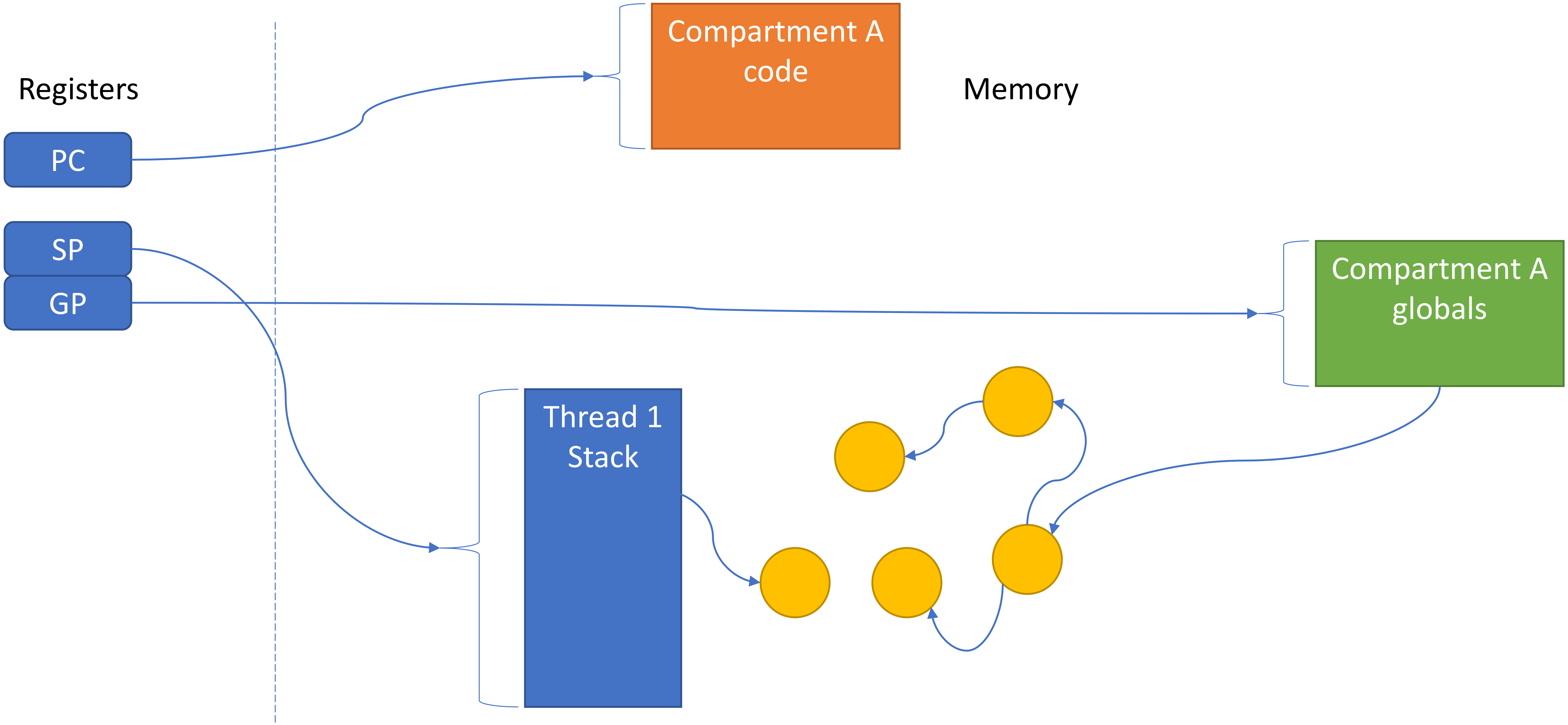
Compartments are code and data and exports



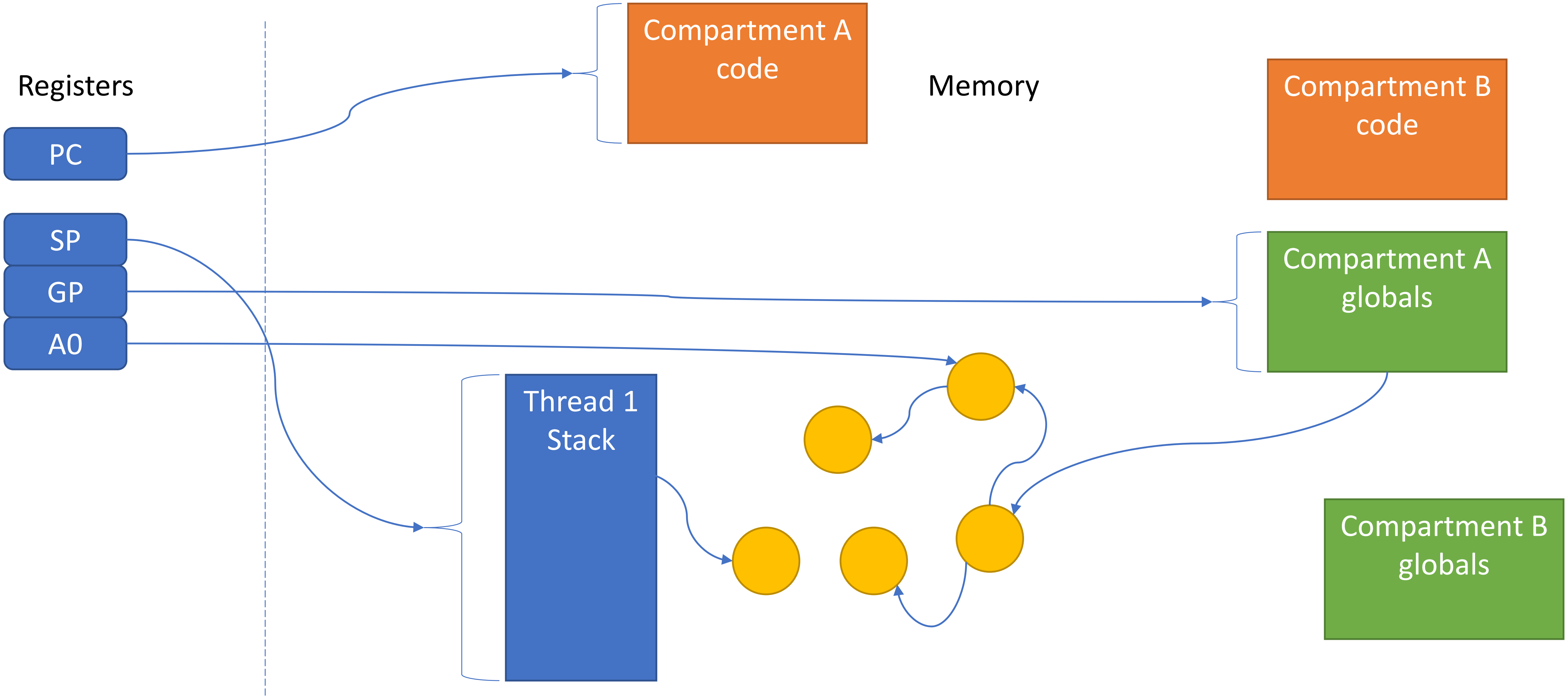
Compartments are code and data and exports



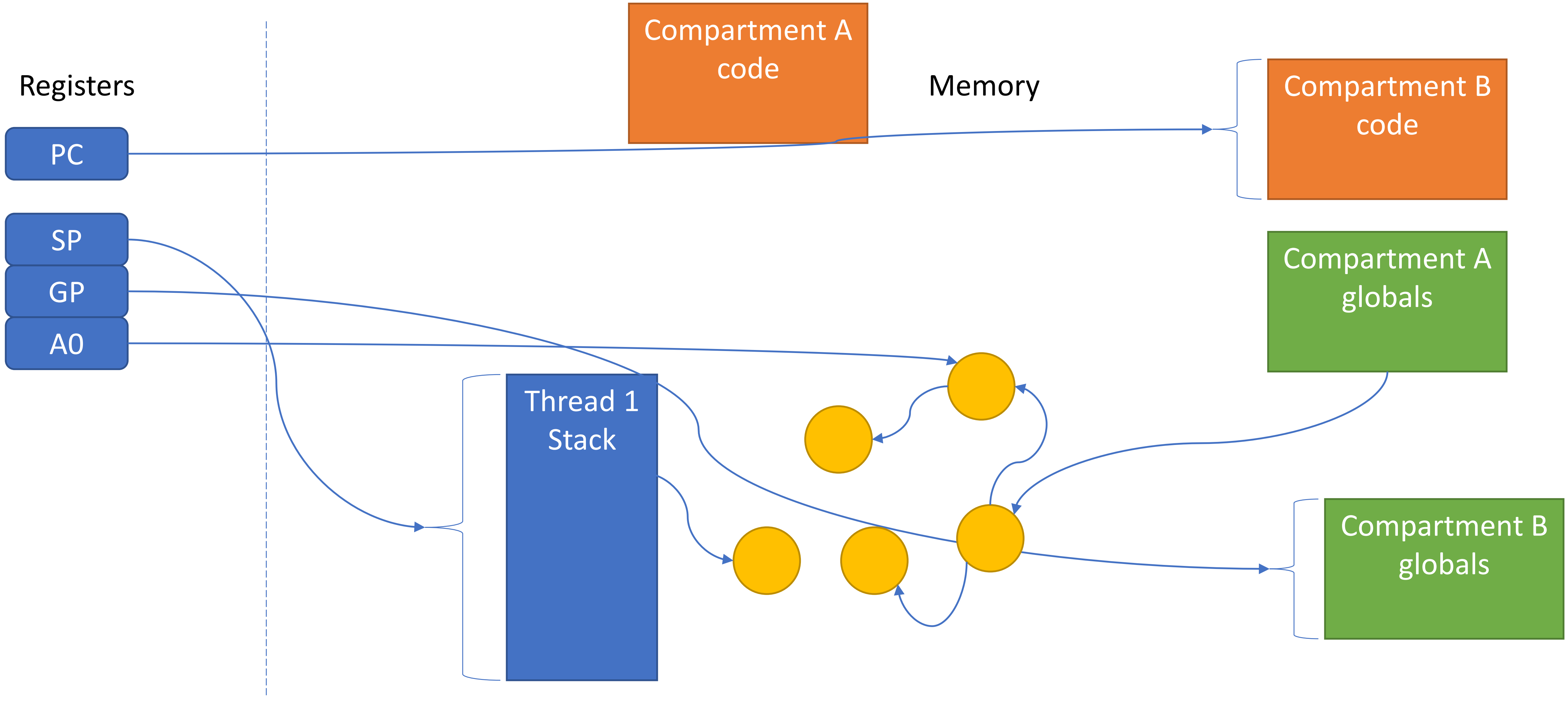
From unforgeable pointers to compartments



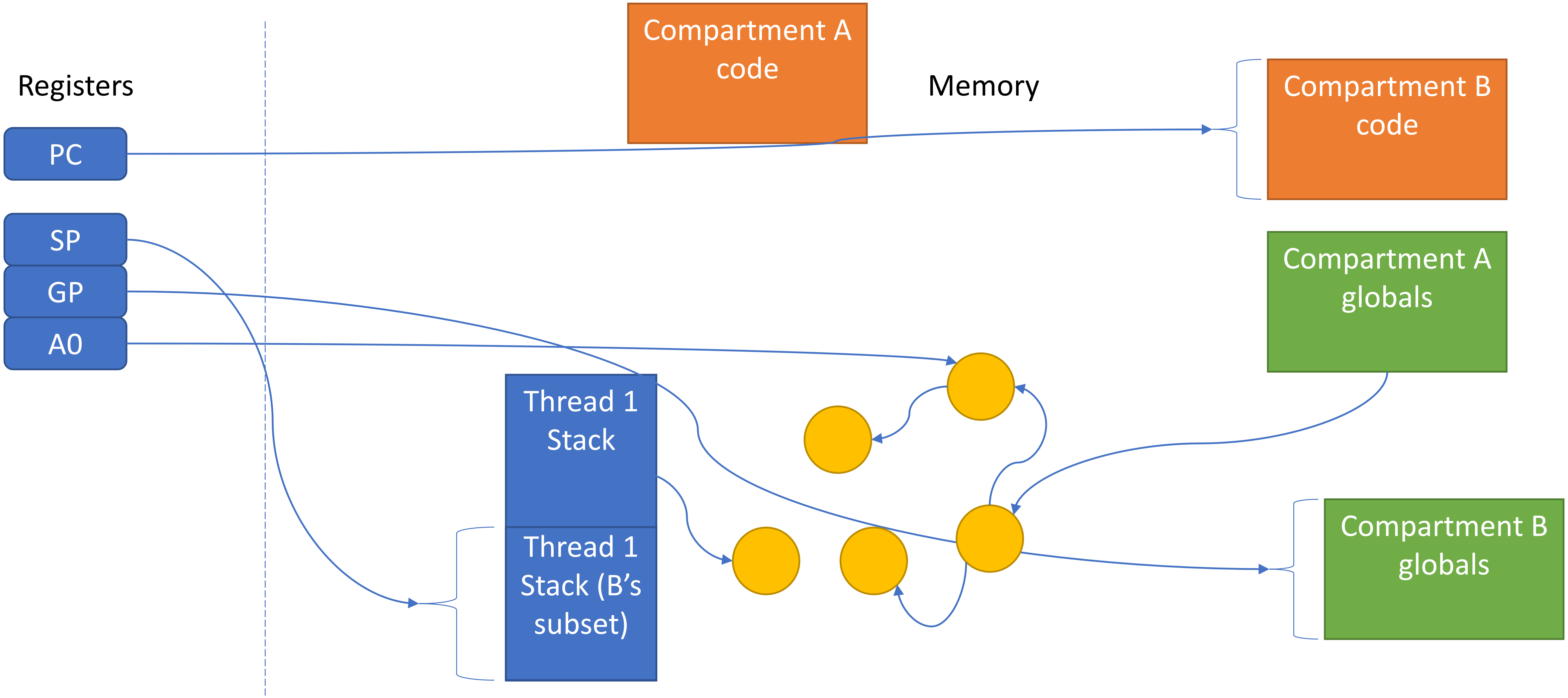
From unforgeable pointers to compartments



From unforgeable pointers to compartments



From unforgeable pointers to compartments



Security guarantees across compartments

No sharing except via explicit pointer passing

Pointers from the caller may prevent modification or capture

Trusted (privilege-separated) components

Loader

- Has full access to all memory
- Not needed if flash can store tags

Switcher

- Can see state from multiple threads and compartments
- Has access to a reserved register
- Around 300 instructions

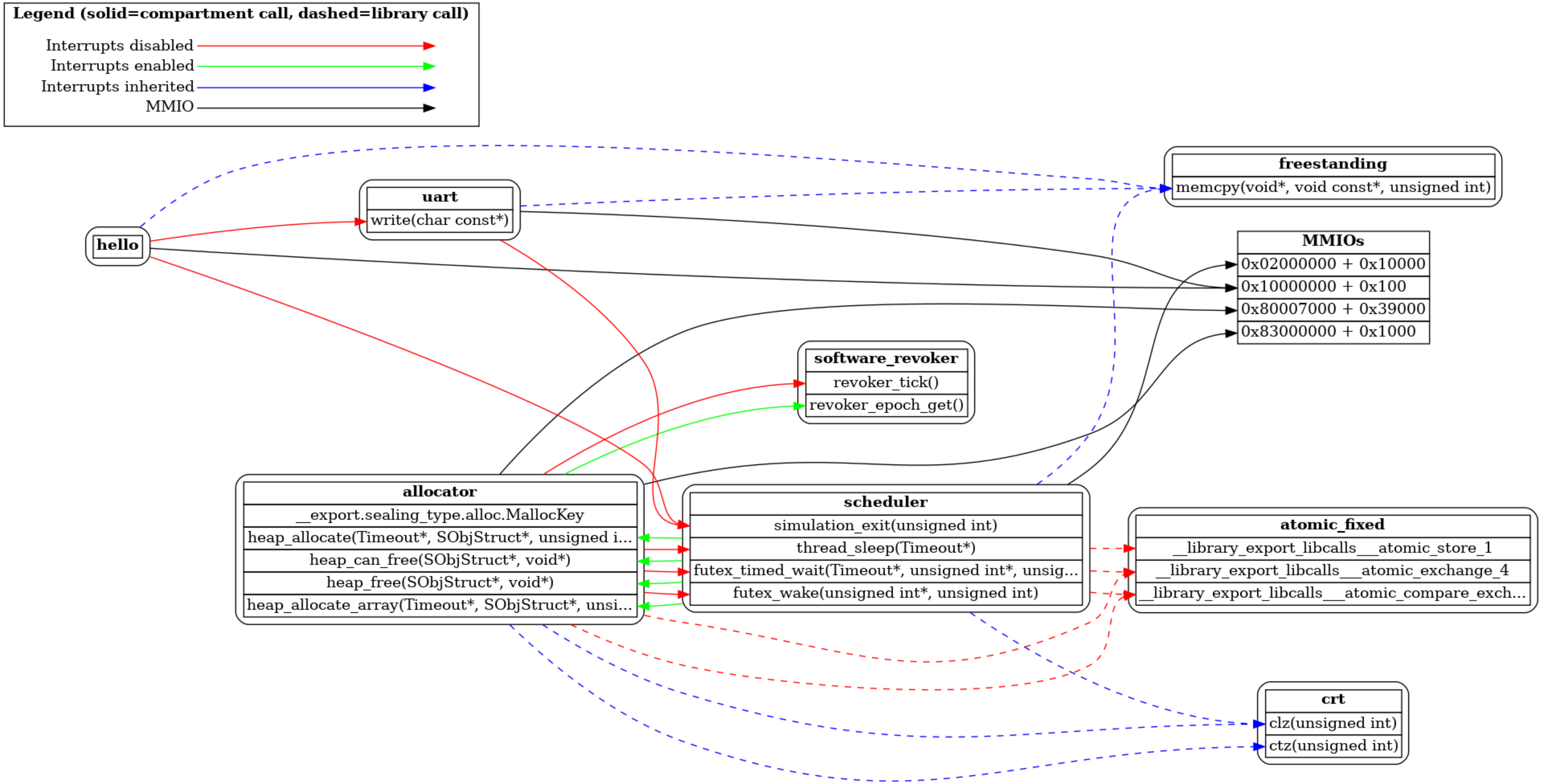
Scheduler

- Trusted for availability
- No access to suspended thread state (registers or stack)

Memory allocator (optional)

- Sets bounds / revocation state on allocations

What can we statically audit?



Legend (solid=compartment call, dashed=library call)

Interrupts disabled

Interrupts enabled

Interrupts inherited

MMIO

hello

uart

write(char const*)

freestanding

memcpy(void*, void const*, unsigned int)

MMIOs

0x02000000 + 0x10000

0x10000000 + 0x100

0x80007000 + 0x39000

0x83000000 + 0x1000

software_revoker

revoker_tick()

revoker_epoch_get()

allocator

__export.sealing_type.alloc.MallocKey

heap_allocate(Timeout*, SObjStruct*, unsigned i...

heap_can_free(SObjStruct*, void*)

heap_free(SObjStruct*, void*)

heap_allocate_array(Timeout*, SObjStruct*, unsi...

scheduler

simulation_exit(unsigned int)

thread_sleep(Timeout*)

futex_timed_wait(Timeout*, unsigned int*, unsig...

futex_wake(unsigned int*, unsigned int)

atomic_fixed

__library_export_libcalls___atomic_store_1

__library_export_libcalls___atomic_exchange_4

__library_export_libcalls___atomic_compare_exch...

crt

clz(unsigned int)

ctz(unsigned int)

Summary

- Fine-grained memory safety guarantees for C/C++
- Lighweight compartments
- Safe bounded cross-compartment sharing
- Strong attestation over compartment structure

Any more questions, please ask in the GitHub Microsoft/CHERIoT-RTOS Discussions!
https://github.com/microsoft/cheriot-rtos/discussions/categories/q-a

Backup

Most codebases require very few changes

Microvium embedded JavaScript interpreter

- No changes

TPM reference stack

- No changes for memory safety
- Small changes (<10LoC) for RISC-V
- One line changed to run in a compartment

FreeRTOS network stack

- No changes for memory safety
- Annotations for cross-compartment calls
- Explicit sealing and unsealing
- Small changes (~100 LoC) to run without disabling interrupts for mutual exclusion

mBedTLS

- No changes for memory safety
- Small changes for compartmentalisation

Add compartmentalization to C/C++

```
// Declaration adds an attribute to indicate
// the compartment containing the implementation
void __attribute__((cheri_compartment(“kv_store_sdk")))
publish(char *key, uint8_t *buffer, size_t size);

// Call site looks like normal C.
// Compiled to a direct call in compartments build with
// -cheri-compartment=kv_store_sdk
// Compiled to a cross-domain call in all other cases.
uint8_t buffer[BUFFER_SIZE];
publish("key_id", buffer, sizeof(buffer));
```