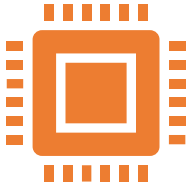
CHERIoT: Complete memory safety for embedded devices

Microsoft

Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, **Kunyan Liu**, **Robert Norton**, Simon W. Moore, Yucong Tao, Robert N. M. Watson, Hongyan Xia

Problem: microcontroller memory safety



Microcontrollers are everywhere

SoCs (AMD SP, RoT…), peripheral devices, IoT devices, vehicles, industrial control, robots…

Security critical, hostile environments

Firmware usually C / asm

Vulnerabilities are common e.g. CVE-2021-26354

Some firmware never updated…

Constrained HW, little or no security features

No MMU, small memory (100s kB)

SW defences rarely employed (overhead? compatibility?)

HW defences absent or incomplete

CHERIoT project goals

Strong memory safety for existing C / C++ (actually any language, even assembly code)

Scalable compartmentalization for defense in depth, fault isolation, safe 3rd party code integration

Lightweight switching between compartments

Efficient sharing between compartments

Low hardware cost

CHERI Collaborators

Origin: Cambridge & SRI c. 2010

Government: DARPA, UKRI, DSbD Program

Industry: arm (Morello SoC), Google, HP, Microsoft, Linaro…

Academia: KCL, Imperial, Kent, Queen's, Oxford, Edinburgh, KU Leuven, ETH Zurich…

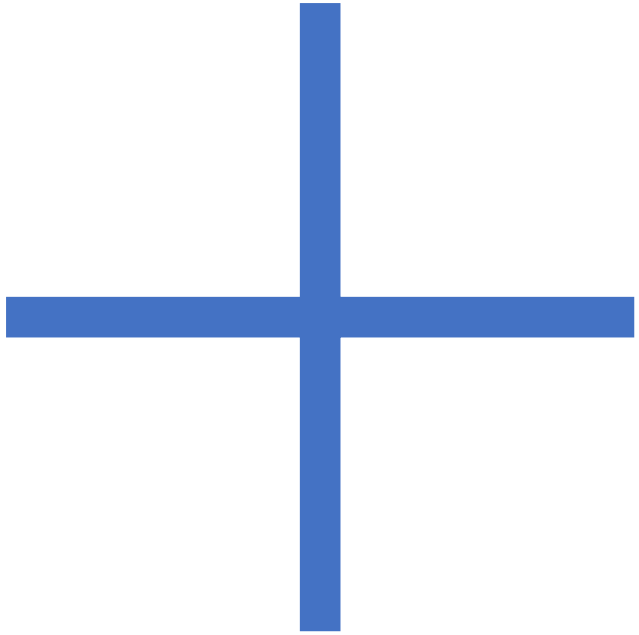
CHERI

chericpu.org

Starting point: CHERI on 64-bit systems

- Hardware knows about pointers (aka capabilities)
- Pointers carry bounds
- Pointers carry permissions
- Pointers can't be created from thin air (constrained manipulation only)
- All guarantees are deterministic
- No guarantees rely on secrets
- All checks are constant time

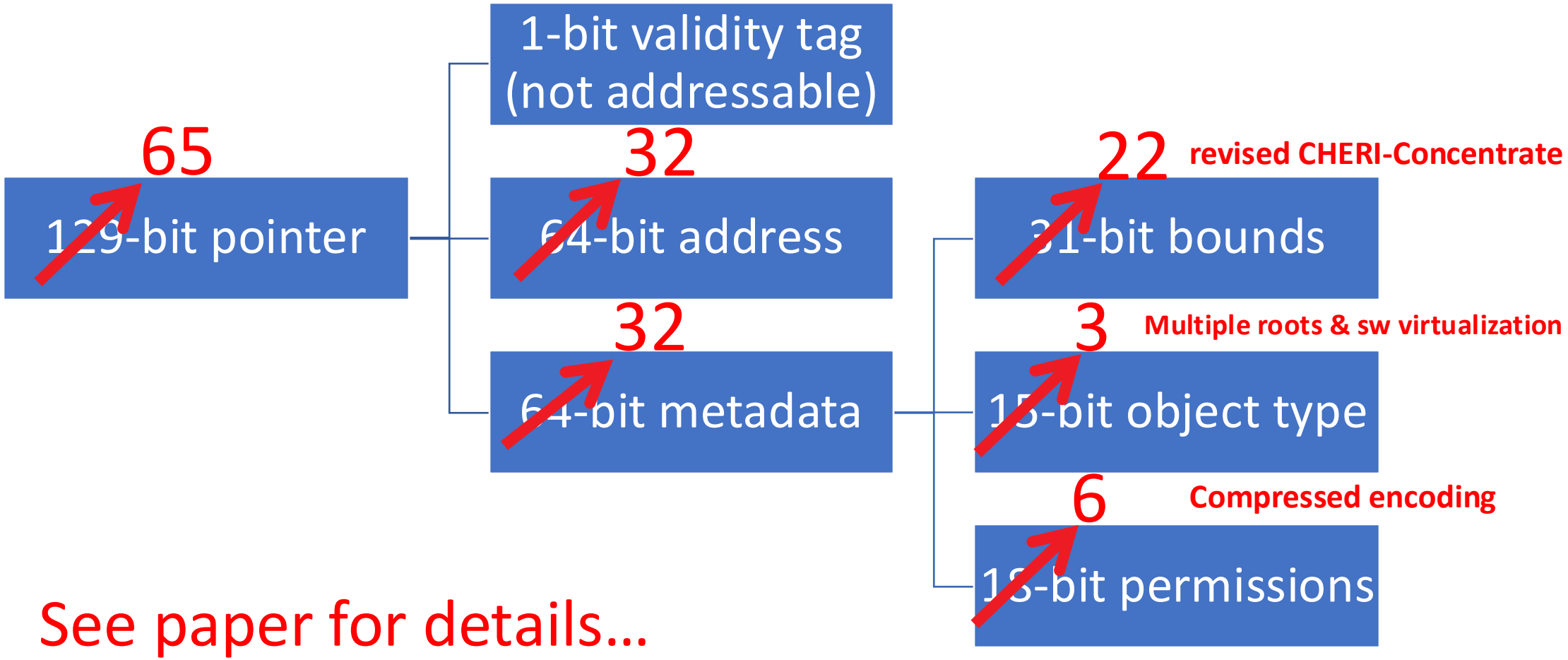
All memory access instructions require a valid pointer operand



Contributions in talk

1. Shrinking capabilities for RV32
2. Adding temporal safety for the heap
3. Building compartments using capabilities
4. Adding temporal safety for the stack

For 32-bits we have to shrink things



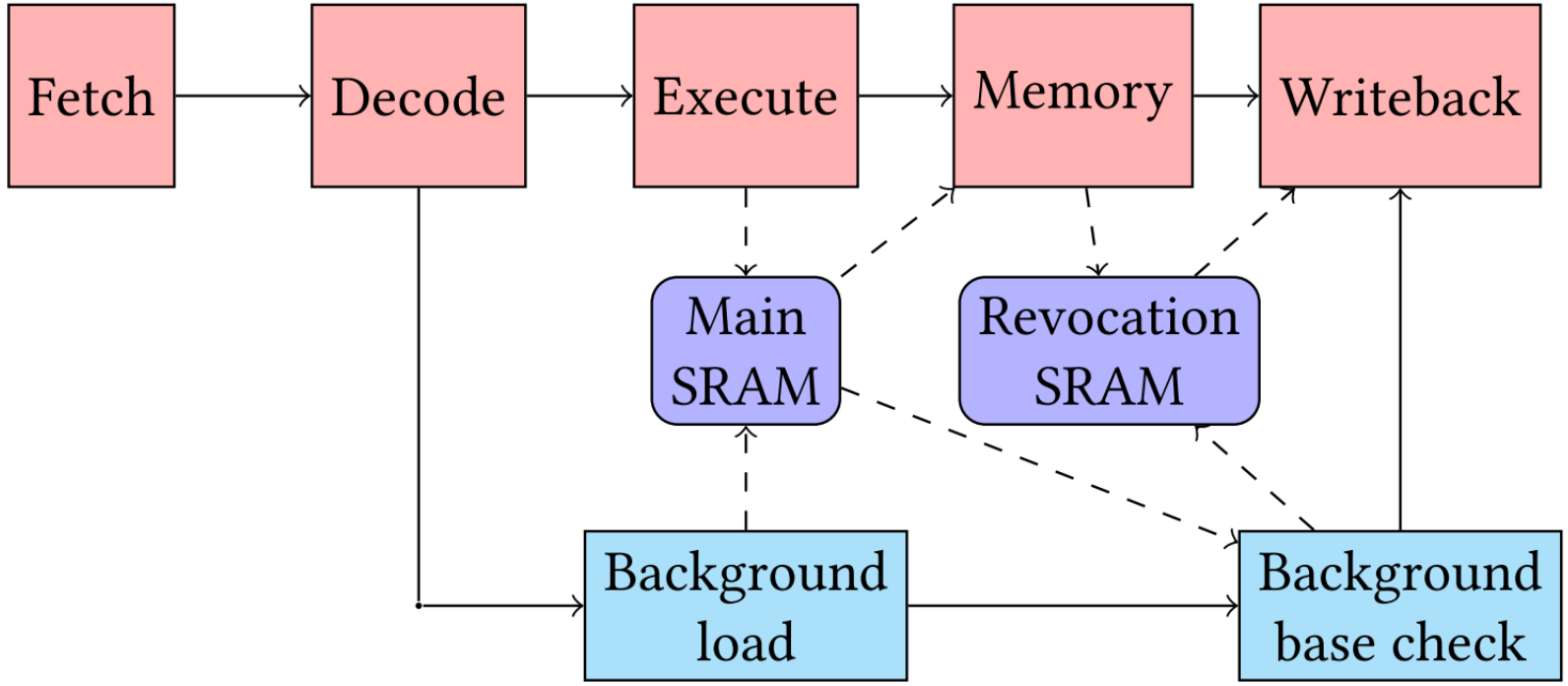
See paper for details…

+1 spare!

Adding temporal safety for heap

Add one-bit ‘is freed’ marker (revocation bit) per 8 bytes of heap memory

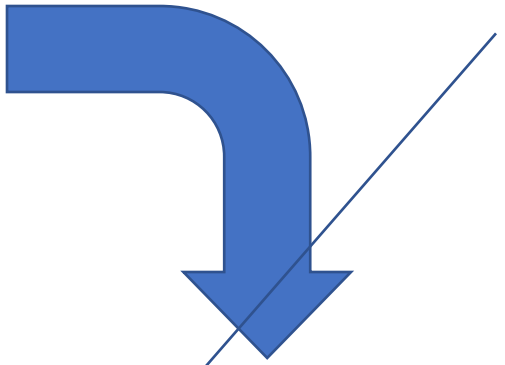
Load filter: check on pointer load and invalidate if freed



Deterministic use-after-free protection in C/C++

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

Hidden detail:
Allocator is compartmentalised so return from free reloads x .



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

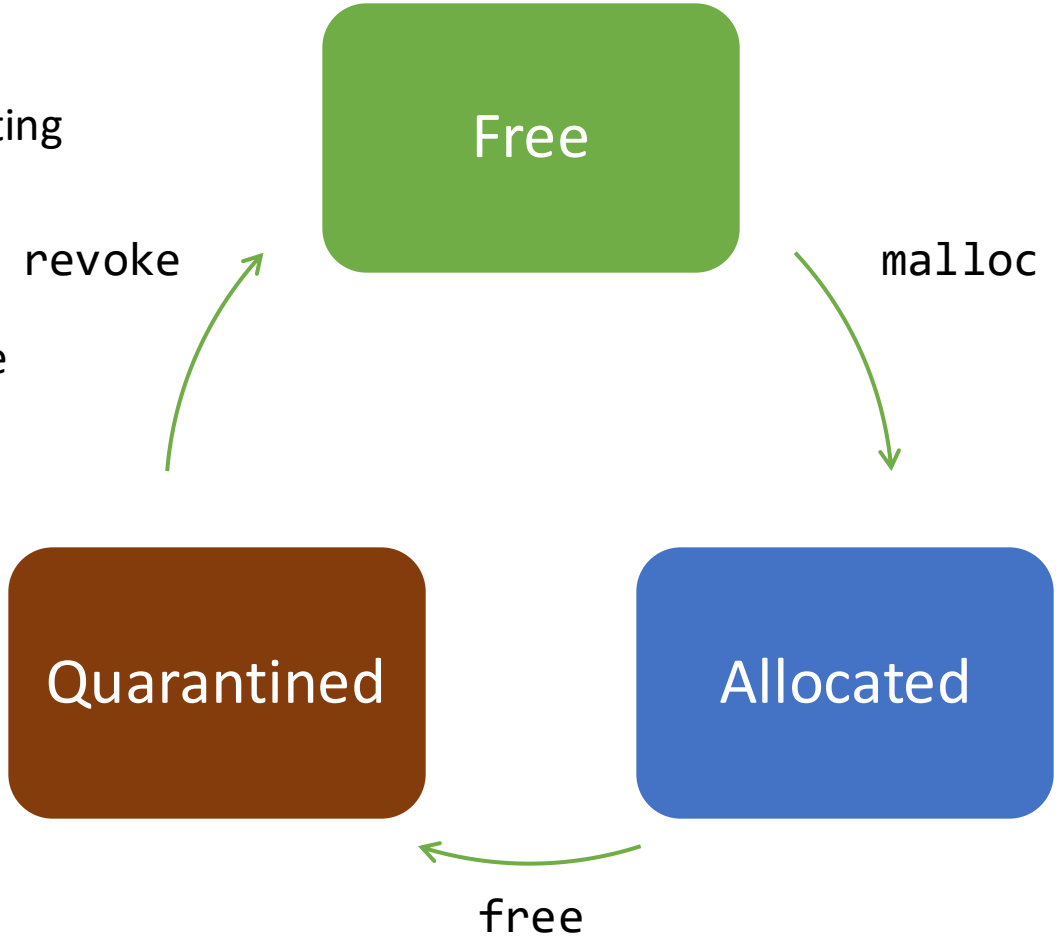
```
Allocating compartment: Allocated: 0x80005900 (v:1 0x80005900-0x80005930 l:0x30 o:0x0 p: G RWcgm- -- ---)
Allocating compartment: Use after free: 0x80005900 (v:0 0x80005900-0x80005930 l:0x30 o:0x0 p: G RWcgm- -- ---)
```

What about reusing memory?

Need to clear revocation bits before reuse, but have dangling pointers

1. sweep all memory invalidating dangling pointers
2. clear revocation bits
3. add to free list

- NOT synchronous: can make concurrent in background

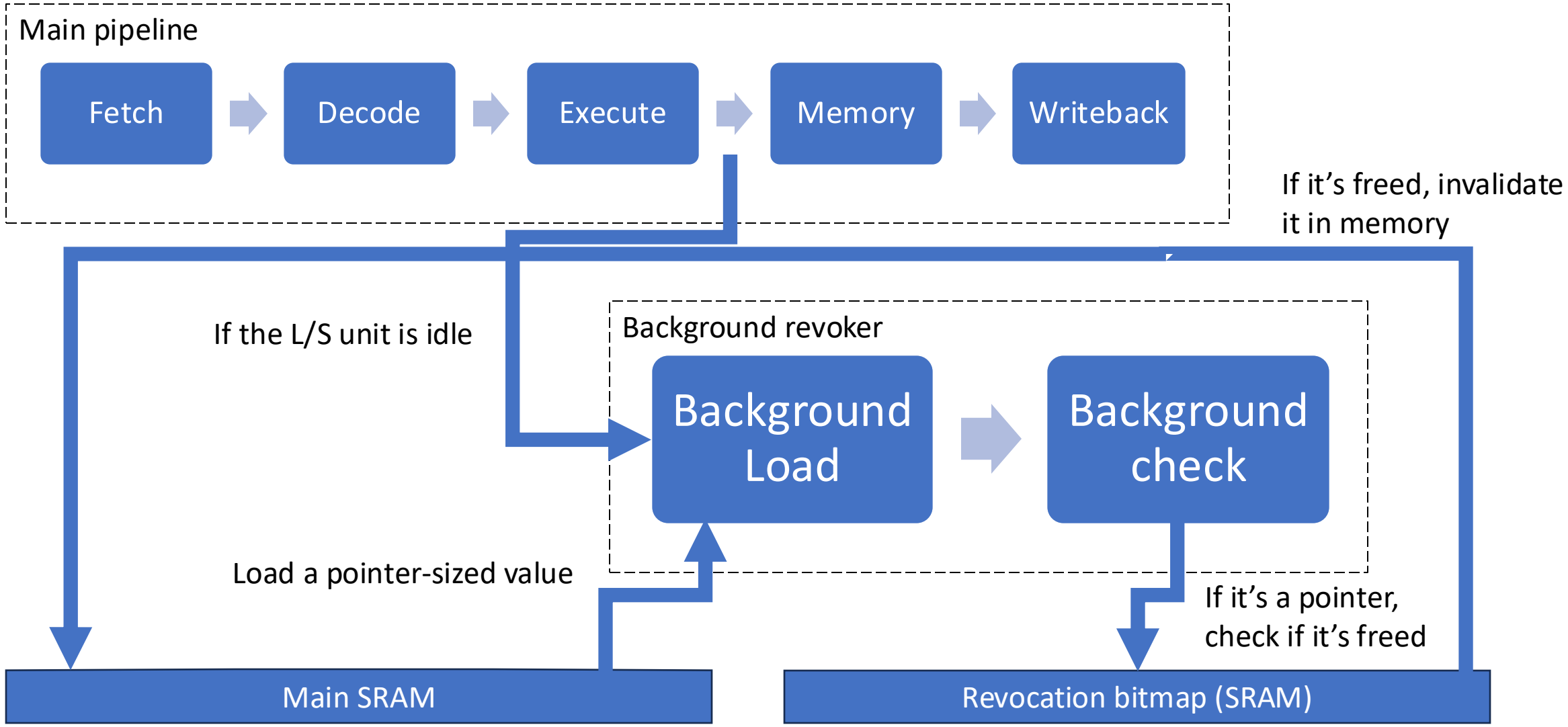


1. allocate from free list
2. set pointer bounds
3. return pointer

accumulate quarantine until heuristic triggered
(e.g. % of heap quarantined)

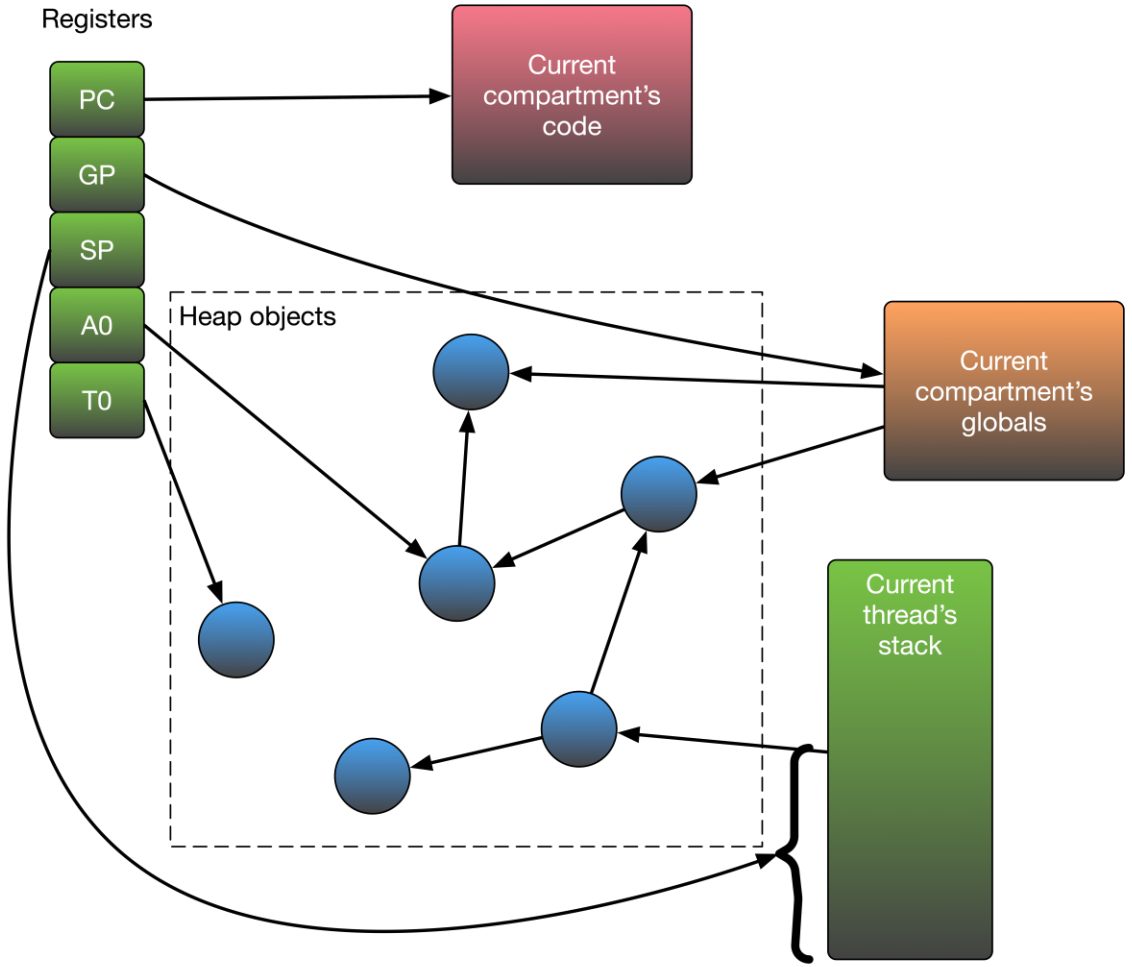
1. set revocation bits 2. add to quarantine 3. can no longer load pointers to it

Hardware revocation



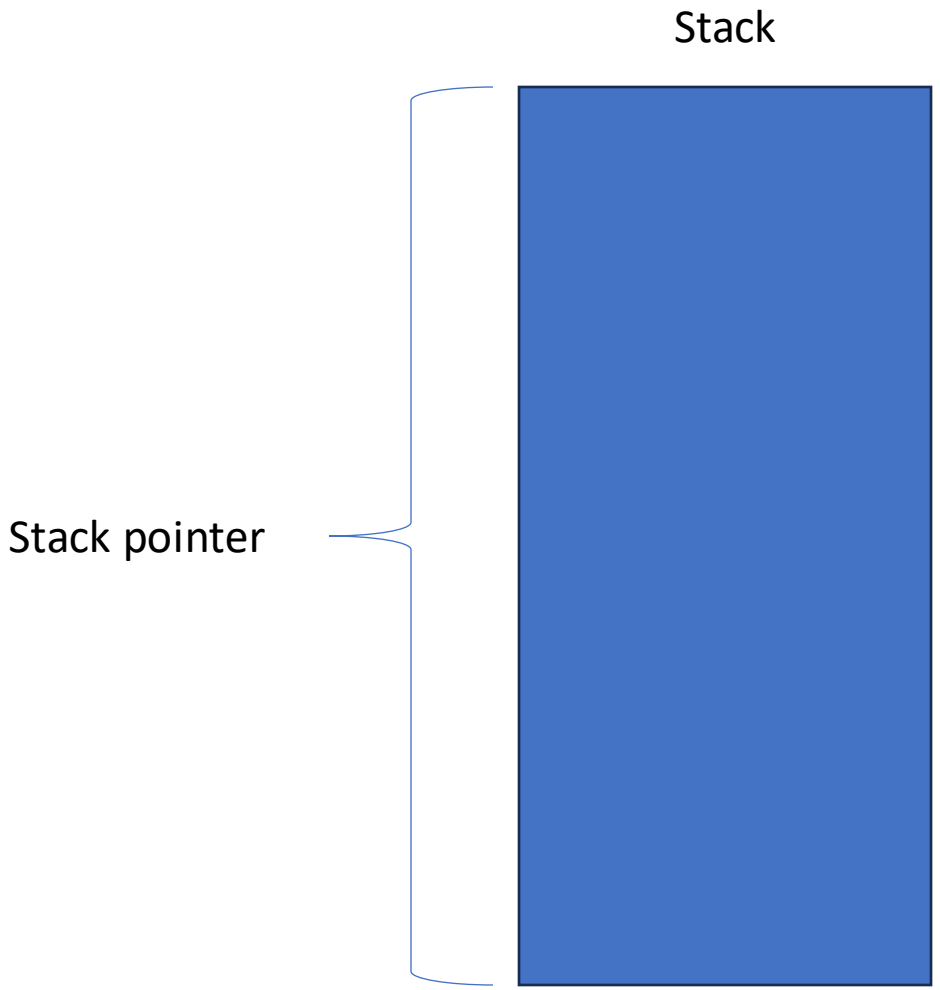
From unforgeable pointers to compartments

- Reacheable memory is defined by the capabilities in registers file
- We define isolated compartments using program counter and global pointer
- Call between them using a trusted switcher
- Looks like a function call
- Use same stack…



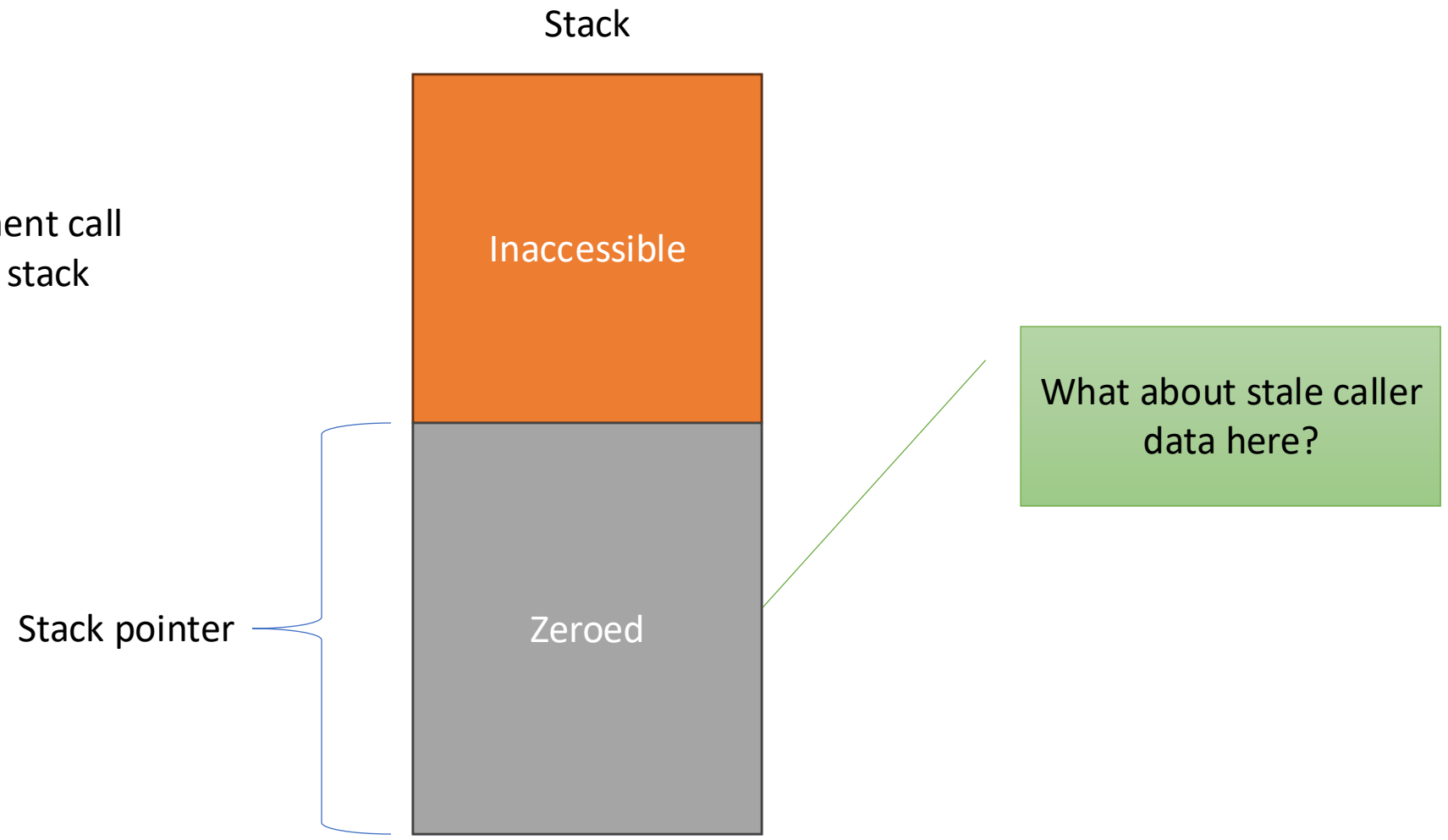
Stack sharing across compartments

Initial compartment has access to entire stack.

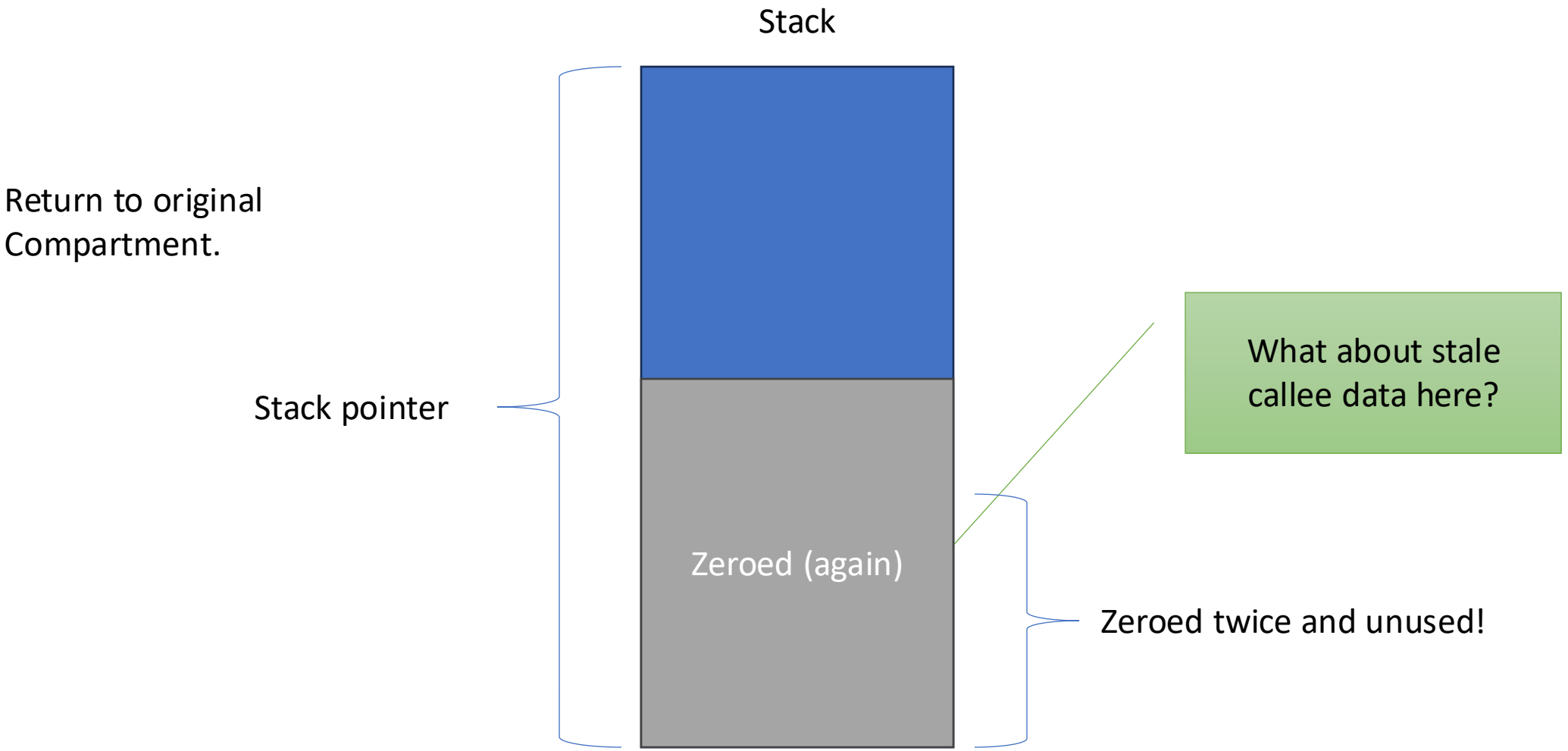


Stack sharing across compartments

During compartment call
Switcher restricts stack



Stack sharing across compartments



Stack high-water mark

- Every store into the stack (below the current mark) moves the mark downwards
- Tracks the most stack memory used
- Only used stack memory needs to be zeroed

Results

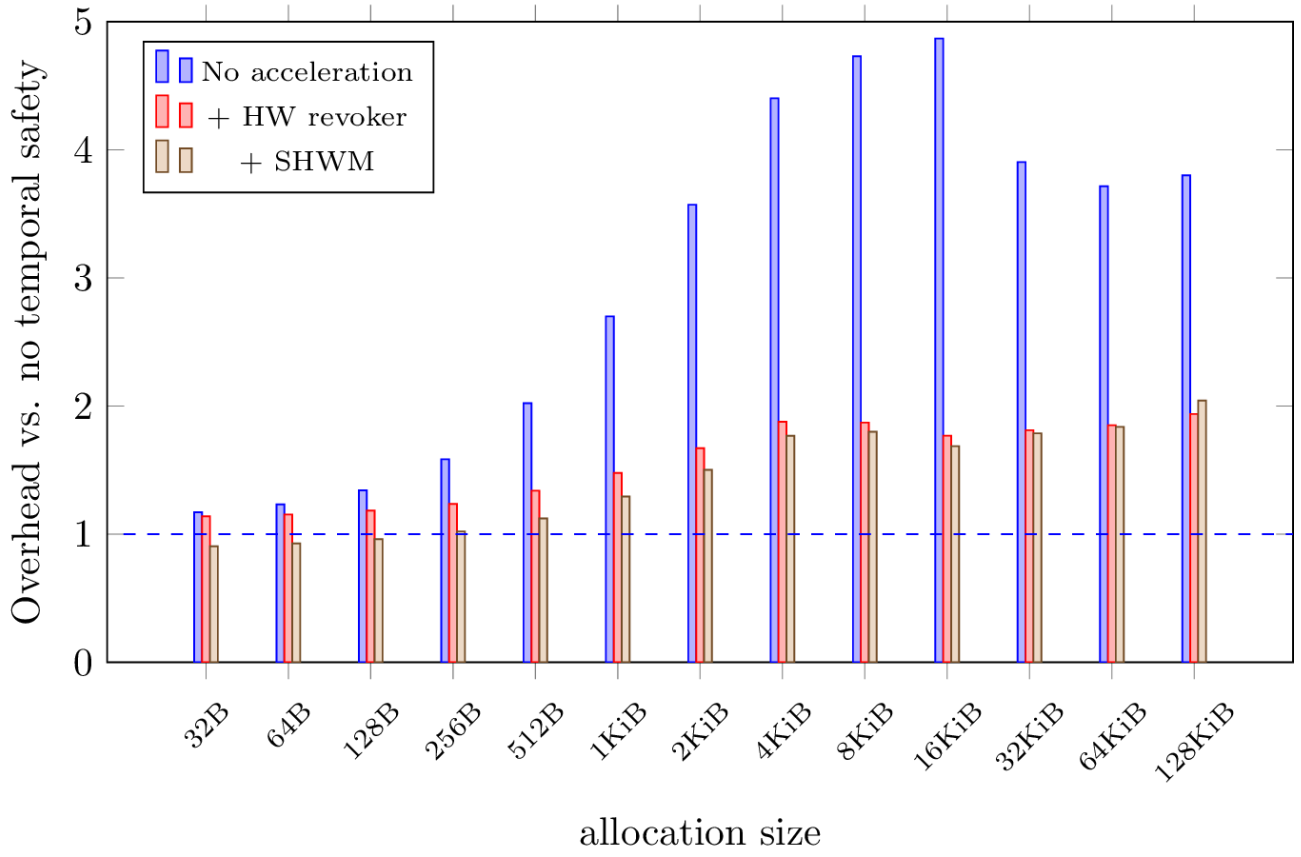
Coremark Overhead (2 implementations)

Configuration	Flute		Ibex	
	Score	Overhead	Score	Overhead
RV32E	2.017	-	2.086	-
+ Capabilities	1.892	5.73%	1.811	13.18%
+ Load filter	1.892	5.73%	1.624	21.28%

- Room for further compiler optimisations to reduce overhead
- Ibex overheads larger due to retaining 32-bit bus and shorter pipeline
- Still acceptable for many use cases!

Allocation microbenchmark results (Ibex core)

- Microbenchmark designed to stress allocator as much as possible: just allocating and freeing 1MiB with no useful work in between
- Worst case for revocation. Demonstrates performance characteristics.
- Allocation size determines number of iterations and hence compartment crossing overhead (allocator is in separate compartment from main loop)
- Small sizes: compartment crossing cost in baseline is significant, revocation costs small by comparison. SHWM outperforms baseline.
- Large sizes: fewer compartment crossings makes revocation cost more visible. Benefits of hardware acceleration clear.
- Relative overhead in real applications more reasonable.



Area and Power (TSMC 28nm HPC+):

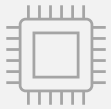
	Ibex 300MHz	
	Gates	**Power** (mW)
RV32E	26988	1.437

- Comparable to PMP with much stronger security!

The whole CHERIoT stack is open source

Start here

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/CHERIoT-Platform/llvm-project/

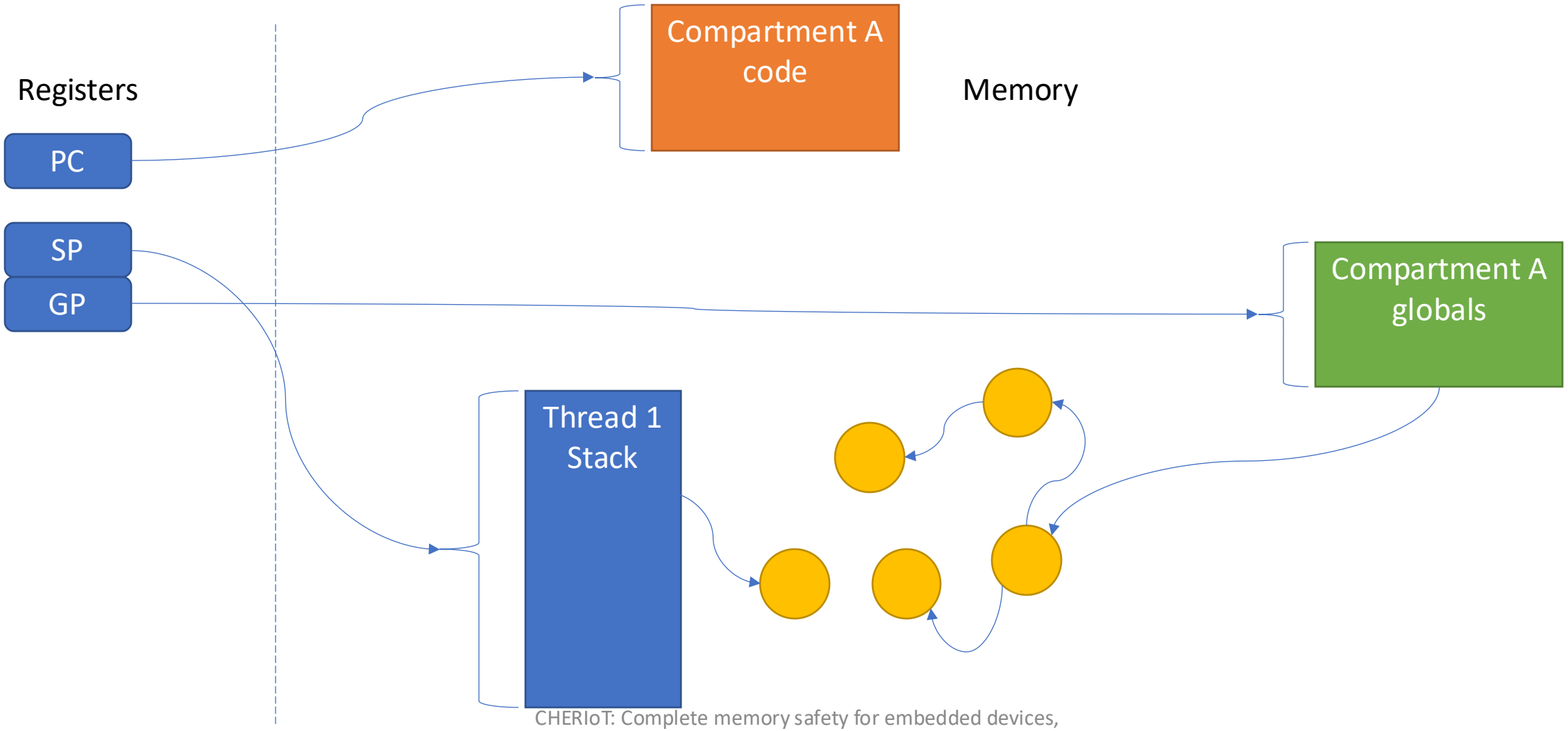
Any more questions, please ask in the GitHub Microsoft/CHERIoT-RTOS Discussions!
https://github.com/microsoft/cheriot-rtos/discussions/categories/q-a

Thanks!

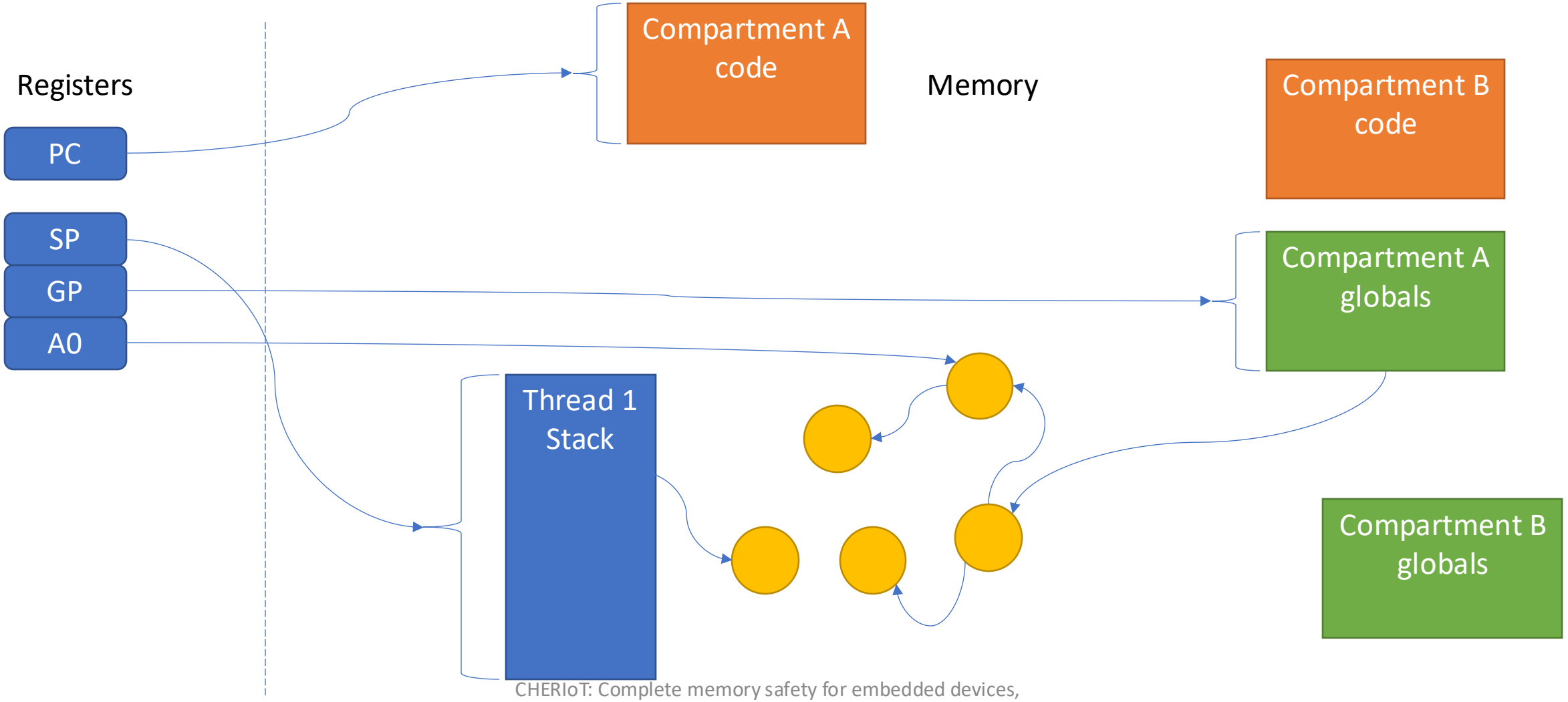
Memory overheads

- Tag bits: 1.6% for memory that might hold pointers
- Revocation bits: 1.6% for heap memory. Optional, but may save memory vs. static memory allocation.
- Stack use: pointer and register spills doubled in size ☹
- Code size: some extra instructions (not measured in this paper)
- Compartments: some overhead for control structures (not in this paper)
- Pointer size:
  - depends on application
  - most data is not pointers (see prior CHERI work)
  - Pointer heavy applications already use non-standard pointers e.g. js interpreter (see microvium)
- Capability representability padding

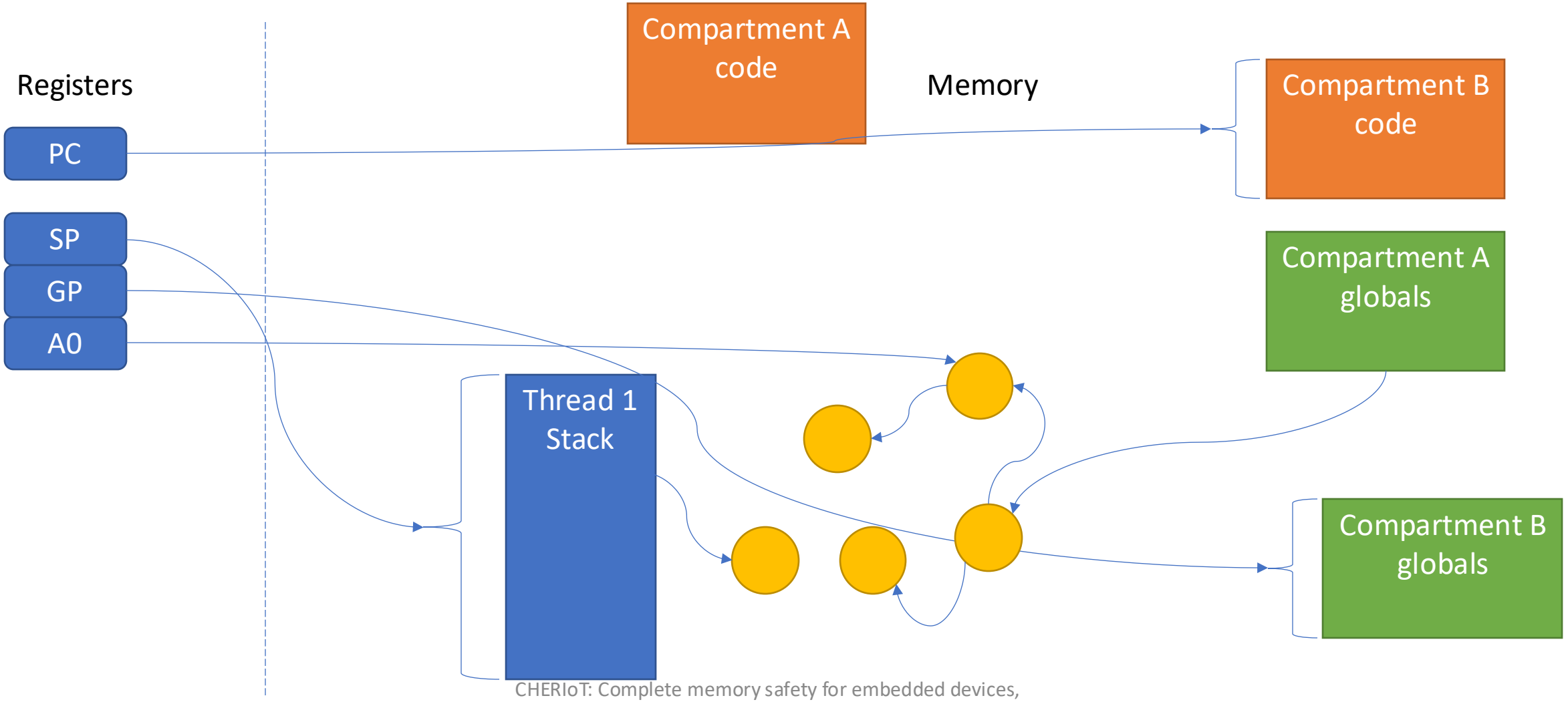
From unforgeable pointers to compartments



From unforgeable pointers to compartments



From unforgeable pointers to compartments



From unforgeable pointers to compartments