Capabilities Great and Small: CHERI, CHERIoT, and LLVM

Owen Anderson

[RFC] Upstream target support for CHERI-enabled architectures

LLVM Project llvm, clang

What?

Why?

Why are we making everything about pointers in LLVM so complicated?

[RFC] Upstream target support for CHERI-enabled architectures

LLVM Project llvm, clang

What?

Why?

~~Why are we making everything about pointers in LLVM so complicated?~~

How does this affect me?

A brief introduction to CHERI

CHERI is a hardware capability system

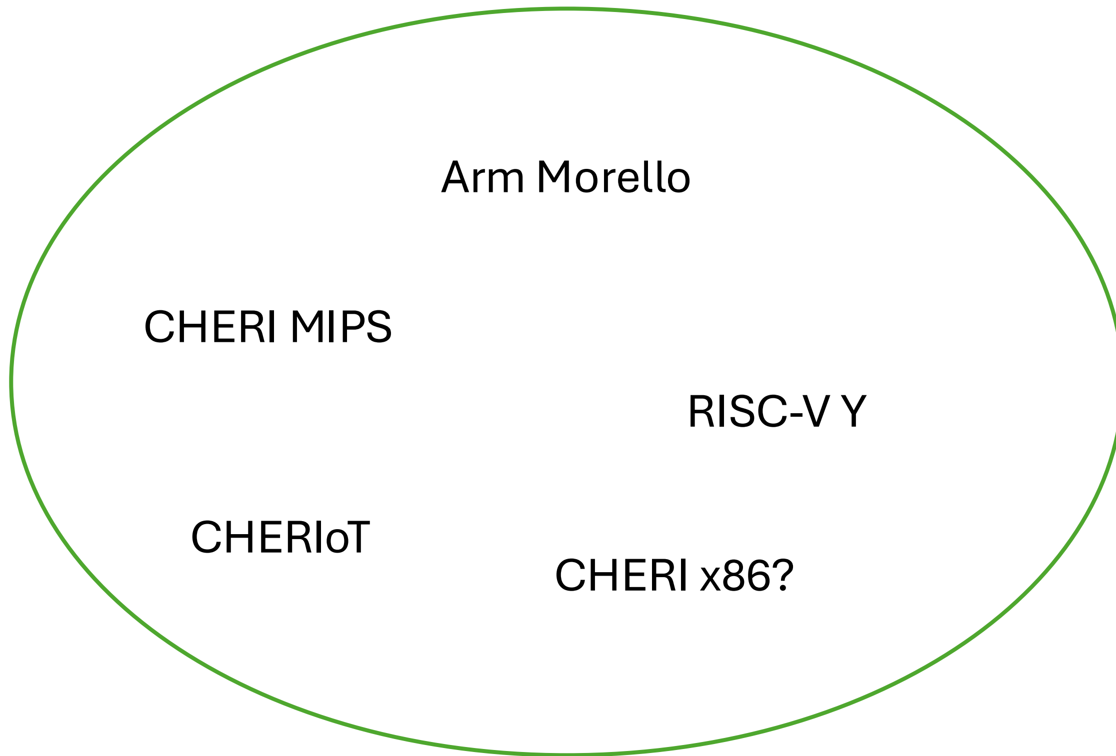
- Memory safety in hardware

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

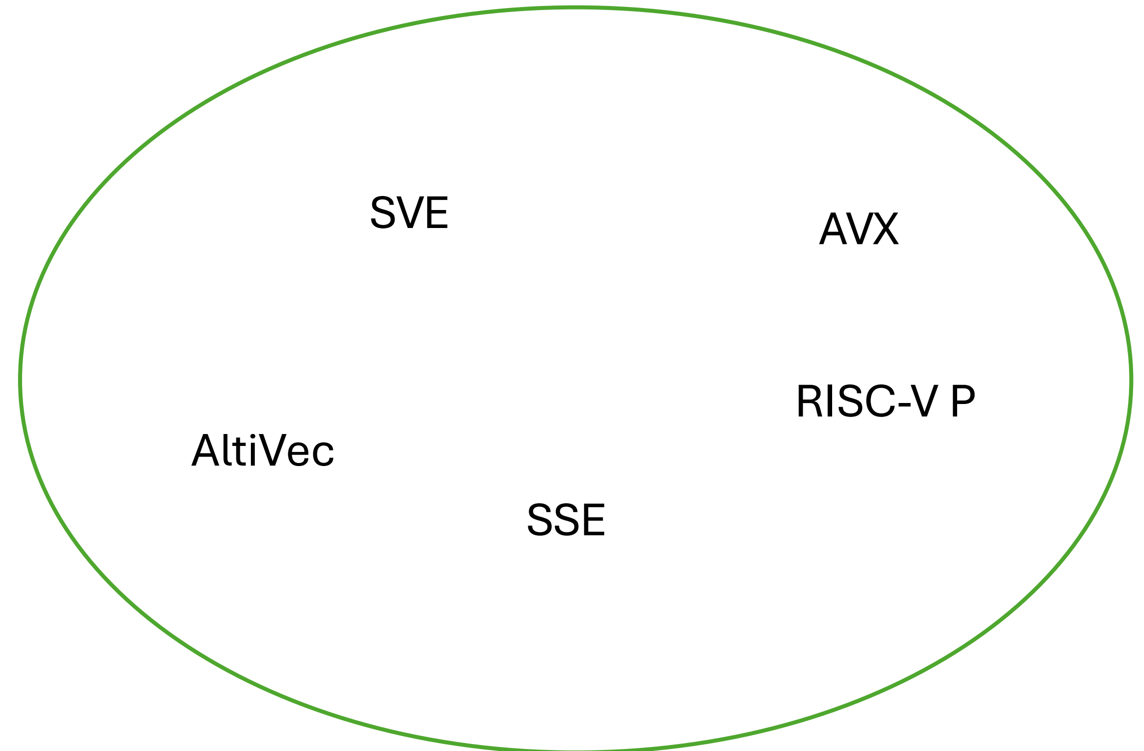
All memory access instructions require a valid pointer operand

CHERI is not *an* architecture

CHERI Architectures



SIMD Architectures



CHERI merges fat pointer and capability concepts

Capabilities

Unforgeable

Can be delegated

Permissions can be reduced

Must be explicitly presented to use their rights

Fat pointers

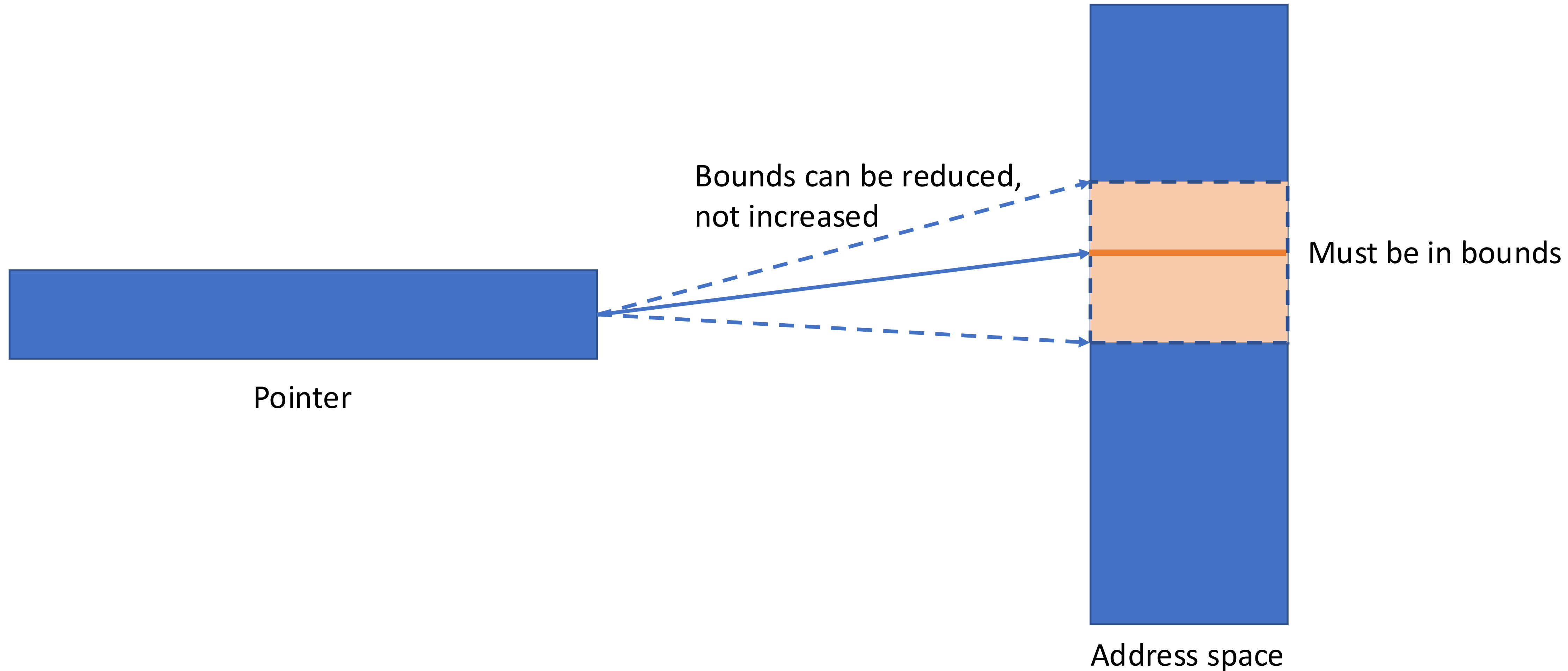
Refer to an address

Add metadata

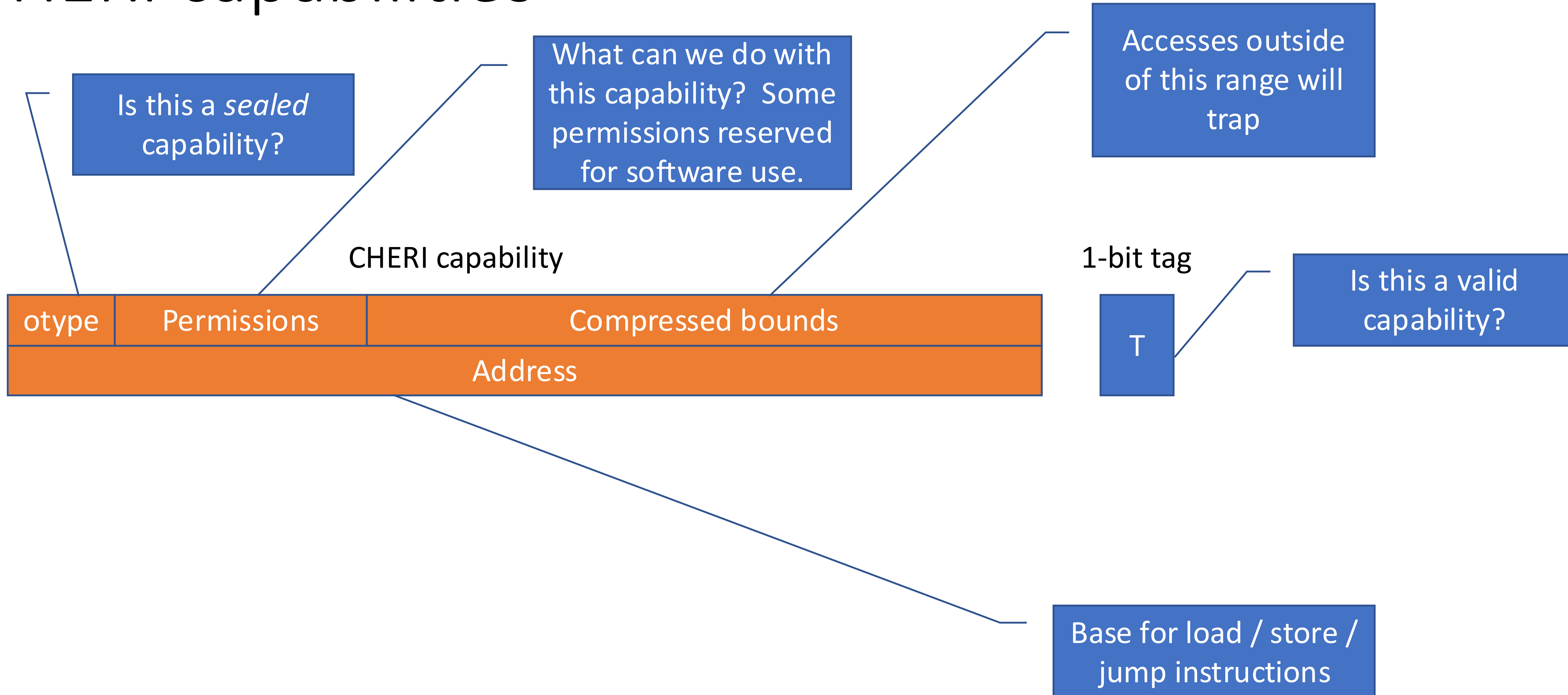
Can be passed or embedded in objects

Must be explicitly dereferenced

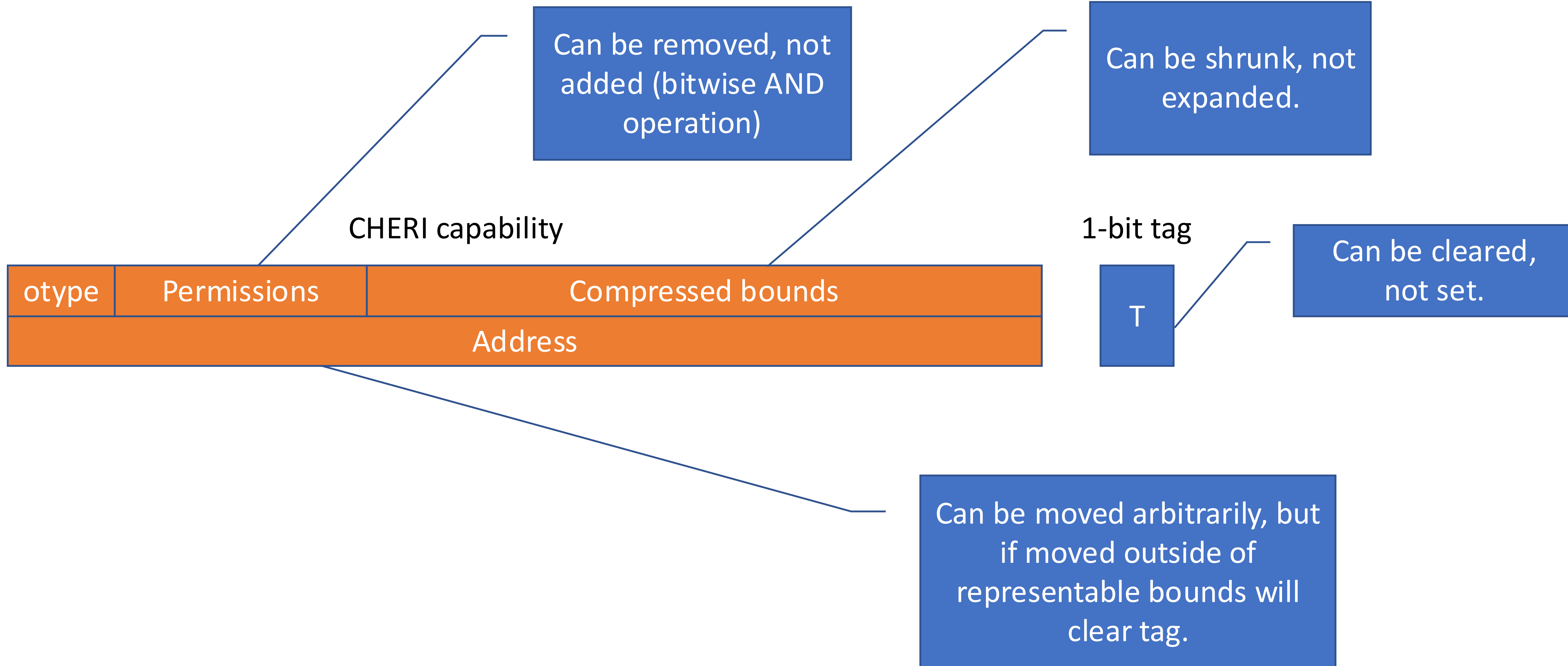
Supporting memory-safe C in hardware



CHERI capabilities



CHERI capabilities are monotonic



Compressed bounds take advantage of redundancy

To-top and to-bottom displacements are smaller than a full address. Expressed as a shared exponent and a small(ish) mantissa

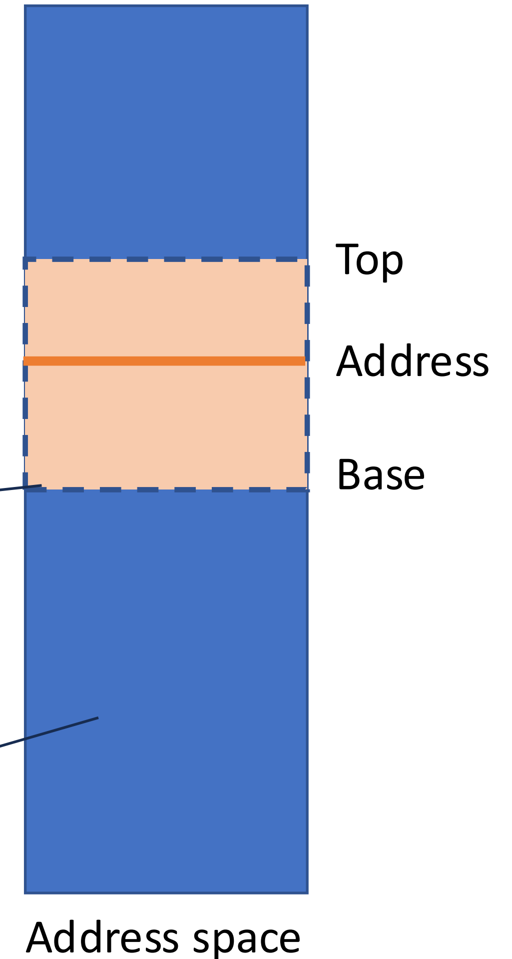
0x12345648 + 0x1b8 = 0x12345800

0x12345648

0x12345648 + 0x248 = 0x12345400

Large allocations must have more strongly aligned top and base, but allocators prefer that anyway

High bits are all the same!



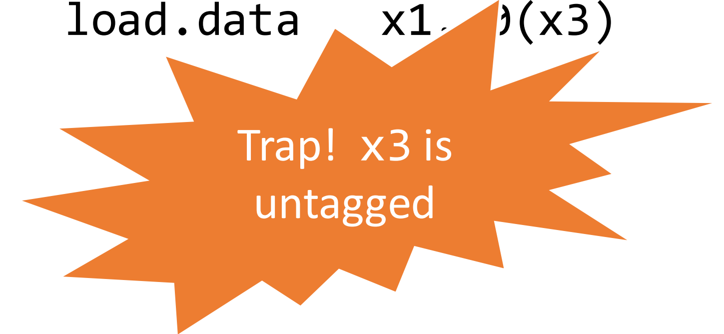
Taking the address too far out of bounds may make it *unrepresentable*, tag will be cleared

CHERI protects capabilities in memory and registers

Physical memory	Valid?
Data	0
Data	0
Capability	1
Capability	1
Data	0
Capability	1

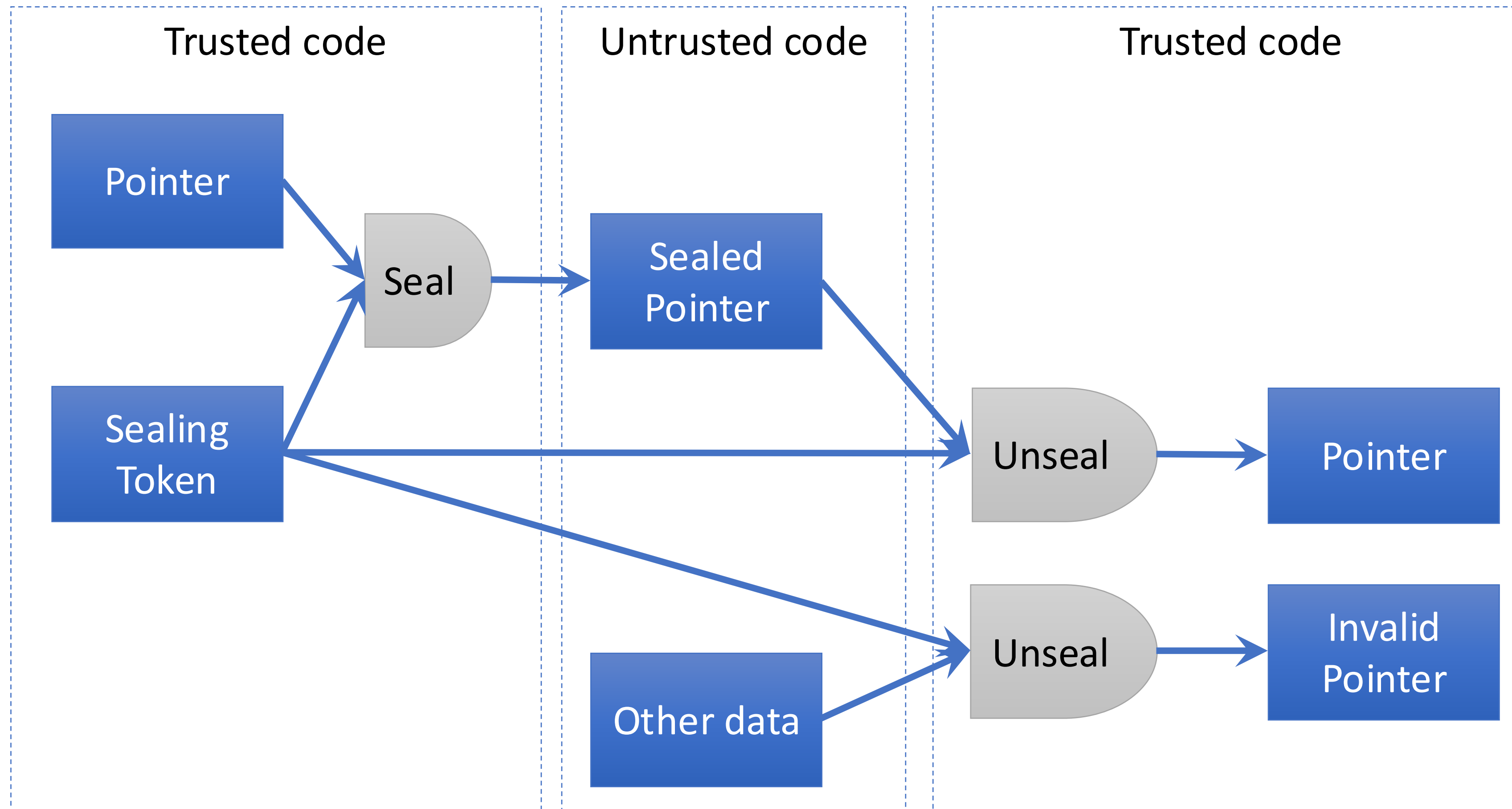
	Register	Valid?
x1	Data	0
x2	Data	0
x3	Capability	1
x4	Capability	1
x5	Capability	1
x6	Capability	1

```
load.cap    x2, 0(x6)
store.data  x1, 0(x6)
load.cap    x3, 0(x6)
load.data   x1, 0(x3)
```



- CHERI embodies a very simple (1-bit) “dynamic type” system:
  - Every word is *either* a capability *or* an integer
  - Using an integer where a capability is required traps
  - Operations also perform permission and bounds checks

Sealing gives unforgeable opaque tokens



CHERI Landscape

CHERI platforms may support two ABIs

Pure-capability

- Every pointer is a capability
- Including stack pointer, program counter, PLT entries

Hybrid

- Binary compatible with existing ABI
- Pointers are addresses by default
- Annotated pointers are capabilities
- Mostly used for interworking

CHERI platforms may support two ABIs

Pure-capability

- Every pointer is a capability
- Including stack pointer, program counter, PLT entries

Focus for upstream support

Hybrid

- Binary compatible with existing ABI
- Pointers are addresses by default
- Annotated pointers with are capabilities
- Mostly used for interworking

Implementations Available – 64-bit

Morello is an AArch64-based CHERI implementation. Development boards are available through CHERI Alliance.

Codasip X730 is a licensable RV64 Xcheri implementation.

buffer-overflow-stack.c — Kate

File Edit Selection View Go Projects LSP Client Debug Sessions Tools Settings Help

New Open Save Save As Close Undo Redo Toggle Breakpoint / Break Step In Step Over

buffer-overflow-stack.c

... src > exercises > buffer-overflow-stack > buffer-overflow-stack.c

```
5   #include <assert.h>
6   #include <stddef.h>
7   #include <stdio.h>
8
9   #pragma weak write_buf
10  void
11  write_buf(char *buf, size_t ix)
12  {
13      buf[ix] = 'b';
14  }
15
16  int
17  main(void)
18  {
19      char upper[0x10];
20      char lower[0x10];
21
```

Locals

Symbol	Value
buf	0xffffff7ff40 [rwRW, 0xfffffff7ff40-0xfffffff7ff50] "5\t\021"
ix	16

Nr	Frame
0	buffer-overflow-stack.c:13
1	buffer-overflow-stack.c:31

GDB Output Settings

```
Continuing.
Program received signal SIGPROT, CHERI protection violation.
Capability bounds fault.
write buf (buf=0xfffffff7ff40 [rwRW,0xfffffff7ff40-0xfffffff7ff50] "5\t\021", ix=16)
at buffer-overflow-stack.c:13
13                  buf[ix] = 'b';
```

Output Search Project Terminal LSP Debug 13:1 INSERT Tab Size: 4 UTF-8 C

Home — Dolphin

Home

Places: Home, Desktop, Documents, Downloads, Music, Pictures, Videos

cheri-exercises, Desktop, Documents, Music, Pictures, Public

102.4 GiB free

Demo

Search...

Favorites, All Applications, Development, Graphics, Internet, Office, Settings, System, Utilities

Firefox Web Browser, System Settings, Dolphin

Applications Places

~ : bash — Konsole

File Edit View Bookmarks Plugins Settings Help

New Tab Split View Copy Paste Find

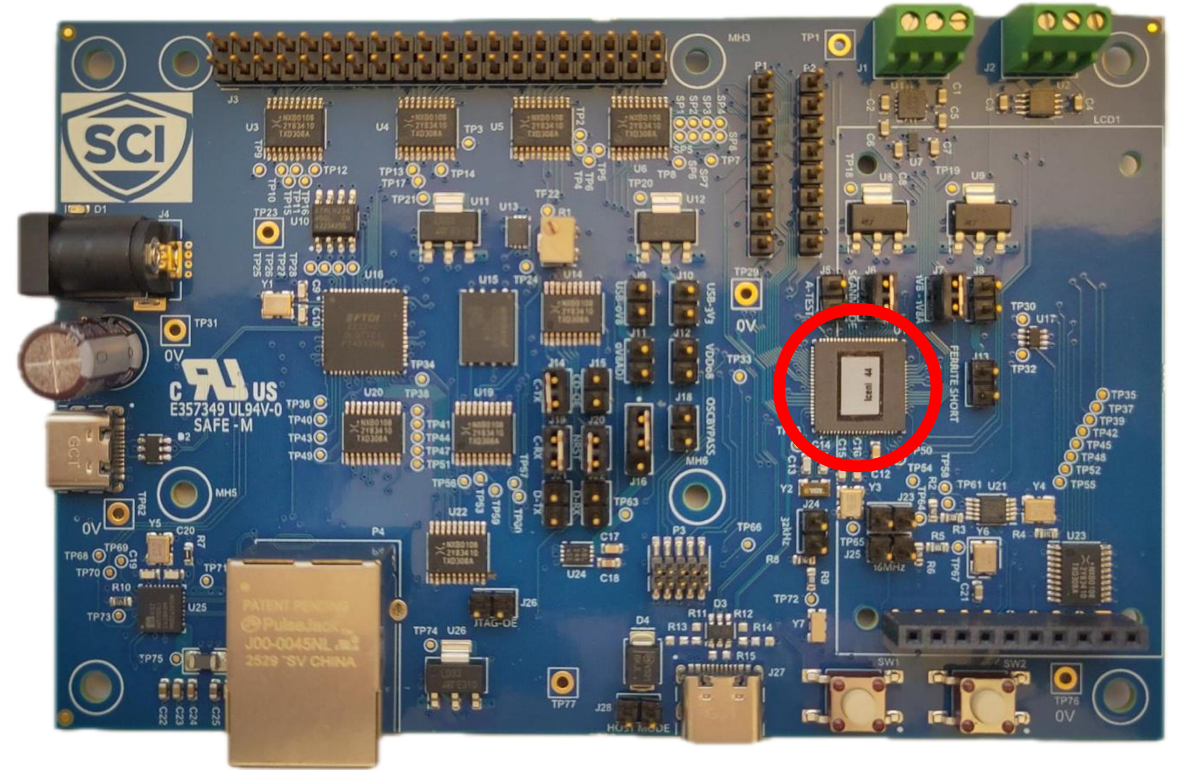
```
[demo@morello ~]$ sysctl hw.machine_arch hw.model
hw.machine_arch: aarch64c
hw.model: Research Morello SoC r0p0
[demo@morello ~]$ file /usr/local/bin/konsole
/usr/local/bin/konsole: ELF 64-bit LSB pie executable, ARM aarch64, C64, CheriABI, version 1 (SYSV), dynamical
ly linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 14.0 (1400064), FreeBSD-style, with debug_info, not s
tripped
[demo@morello ~]$
```

17:29
10 Jan 2023

Implementations Available – 32-bit

CHERIoT is an RV32E-based HW/SW embedded platform built on assuming CHERI from the ground up.

ICENI is the first commercial implementation of CHERIoT from SCI Semi, available commercially in 2026.



RV32Y and RV64Y Base ISAs

Standardized encoding of CHERI registers and instructions

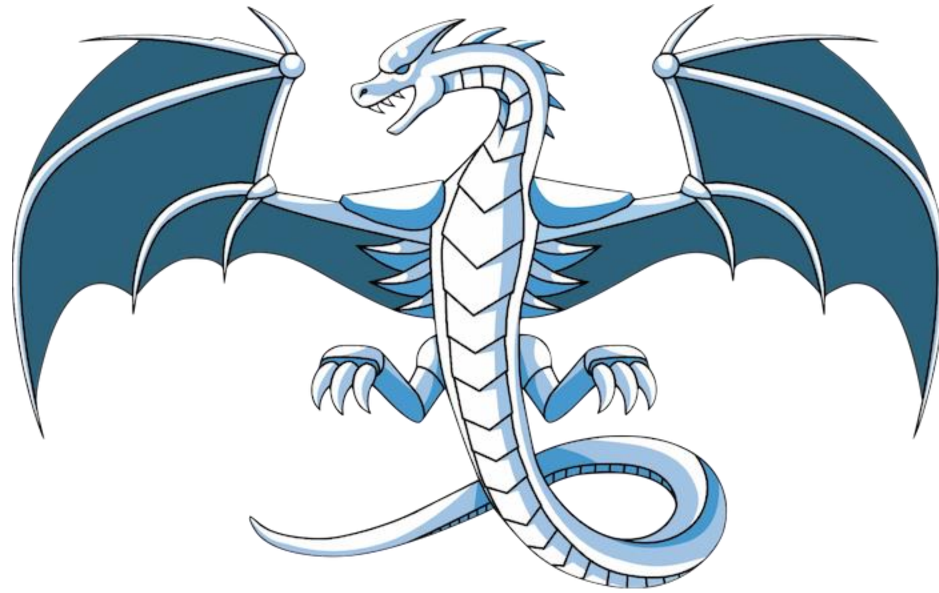
- NOT backwards compatible with existing Xcheri extension
- Hybrid mode will be standardized as a separate Yhybrid
- Draft specification under ARC review

Standardization of ABI is WIP

Shared baseline makes it easier to collaborate across CHERI variants

- Such as upstreaming LLVM patches!

+



= ?

CHERI and LLVM

CHERI capabilities are `addrspace(200)`

```
define ptr addrspace(200) @align_down(ptr addrspace(200) %ptr) addrspace(200) #0 {
entry:
        %0 = call i64 @llvm.cheri.cap.address.get.i64(ptr addrspace(200) %ptr)
        %and = and i64 %0, -16
        %1 = call ptr addrspace(200)
                @llvm.cheri.cap.address.set.i64(ptr addrspace(200) %ptr, i64 %and)
        ret ptr addrspace(200) %1
}
```

`addrspace(200)` represents CHERI capabilities

- In “pure capability” ABI, this is the only address space used
- In “hybrid” ABI, they co-exist with `addrspace(0)` normal pointers

Using the same address space in both modes dramatically simplifies middle- and back-end

Hybrid ABI is not being upstreamed at this time, but is maintained downstream

CHERI touches all layers of the toolchain!

Layer	Changes
Clang	• **Different lowering, new features**
Mid-level optimisers	• Preserving pointer ≠ integer • Lowering
Code generation	• New ISA
LLD	• New relocations, compartment linkage model
Debugger	• More information about pointers

Clang changes lowering for intptr_t

7.22.1.4 Integer types capable of holding object pointers

The following type designates a signed integer type, other than a bit-precise integer type, with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type, other than a bit-precise integer type, with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
uintptr_t
```

Representation is capability, operations work on address

```
void *align_down(void *ptr)
{
        uintptr_t pointerAsInteger = (uintptr_t)ptr;
        pointerAsInteger &= ~0xf;
        return (void*)pointerAsInteger;
}
```

```
define ptr addrspace(200) @align_down(ptr addrspace(200) %ptr) addrspace(200) #0 {
entry:
        %0 = call i64 @llvm.cheri.cap.address.get.i64(ptr ad
        %and = and i64 %0, -16
        %1 = call ptr addrspace(200)
                @llvm.cheri.cap.address.set.i64(ptr addrspace(200) %ptr, i64 %and)
        ret ptr addrspace(200) %1
}
```

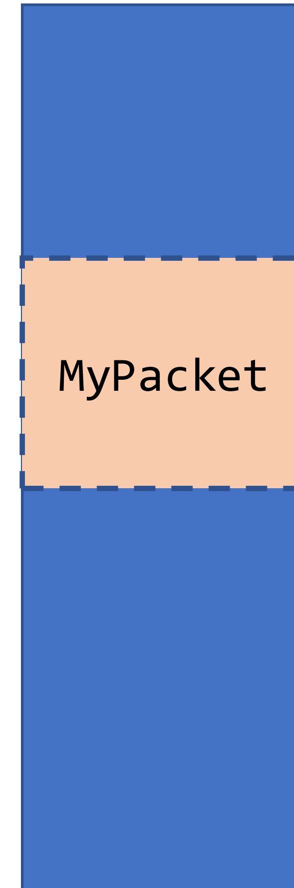
Extract the address

Compute the new address

Rederive a capability using the new address

Sub-Object Bounds

```
struct MyPacket
{
        uint32_t src;
        uint32_t dst;
        uint32_t chksum;
        char payload[1024];
};

void packet_handler(MyPacket *pkt)
{
        if (compute_chksum(&pkt->payload) == pkt->chksum)
                process_packet(pkt);
}
```

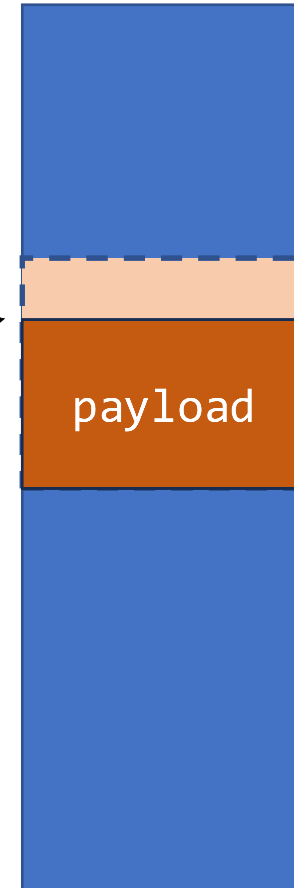


Sub-Object Bounds

```
struct MyPacket
{
        uint32_t src;
        uint32_t dst;
        uint32_t chksum;
        char payload[1024];
};

void packet_handler(MyPacket *pkt)
{
        if (compute_chksum(&pkt->payload) == pkt->chksum)
                process_packet(pkt);
}
```

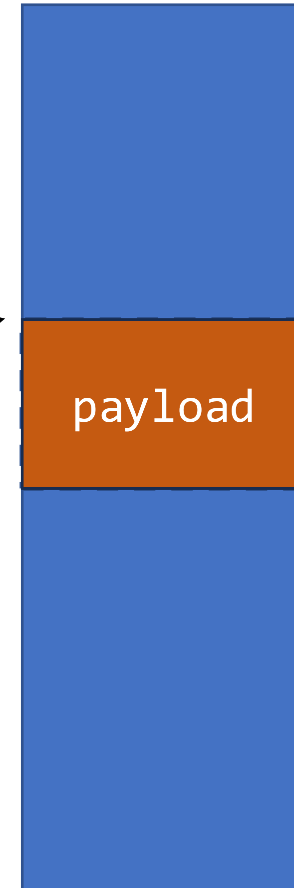
Sub-Object Bounds

```
struct MyPacket
{
        uint32_t src;
        uint32_t dst;
        uint32_t chksum;
        char payload[1024];
};

void packet_handler(MyPacket *pkt)
{
        if (compute_chksum(&pkt->payload) == pkt->chksum)
                process_packet(pkt);
}
```



Sub-Object Bounds

```
struct MyPacket
{
        uint32_t src;
        uint32_t dst;
        uint32_t chksum;
        char payload[1024];
};

void packet_handler(MyPacket *pkt)
{
        if (compute_chksum(&pkt->payload) == pkt->chksum)
                process_packet(pkt);
}
```



Per-TU option to explicitly restrict bounds when deriving a sub-pointer

Annotations available to relax it as needed
- For example, does not work with intrusive linked lists

Some other things in clang

- Correct address spaces everywhere
  - Mostly upstreamed for GPUs
- New calling conventions
- Pointer annotations – hybrid mode, sealing
- Language extensions for compartmentalisation

CHERI touches all layers of the toolchain!

Layer	Changes
Clang	• Different lowering, new features
Mid-level optimisers	• **Preserving pointer ≠ integer** • **Lowering**
Code generation	• New ISA
LLD	• New relocations, compartment linkage model
Debugger	• More information about pointers

LLVM understands pointers are not integers

```
define void @cpy(ptr %src, ptr %dst) {
entry:
       %val = load ptr, ptr %src
       store ptr %val, ptr %dst
}
```

≠

```
define void @cpy(ptr %src, ptr %dst) {
entry:
       %val = load i128, ptr %src
       %val2 = inttoptr i128 %val to ptr
       store ptr %val2, ptr %dst
}
```

Non-Integral Pointers with External Storage

- No new `inttoptr` / `ptrtoint` may be introduced
- VNCoercion has been worst offender, but mostly fixed now
  - Real scenarios often involve partial values

Some optimisations are unsound on CHERI

```
define void @cpy(ptr %src) {
entry:
       %val1 = load i8, ptr %src
       %src1 = gep i32, ptr %src, i32 1
       %val2 = load i8, ptr %src1
       %src2 = gep i32, ptr %src, i32 2
       %val3 = load i8, ptr %src2
       …
}
```

≠

```
define void @cpy(ptr %src) {
entry:
       %wide = load i32, ptr %src
       %val1 = trunc i32 %wide to i8
       %lshr1 = lshr i32 %wide, 8
       %val2 = trunc i32 %lshr1 to i8
       %lshr2 = lshr i32 %wide, 16
       %val3 = trunc i32 %lshr1 to i8
       …
}
```

Load widening is the most pervasive example

- Manifests in many more subtle forms than seen here

memcpy loses valuable type information

Guaranteed not to copy capabilities

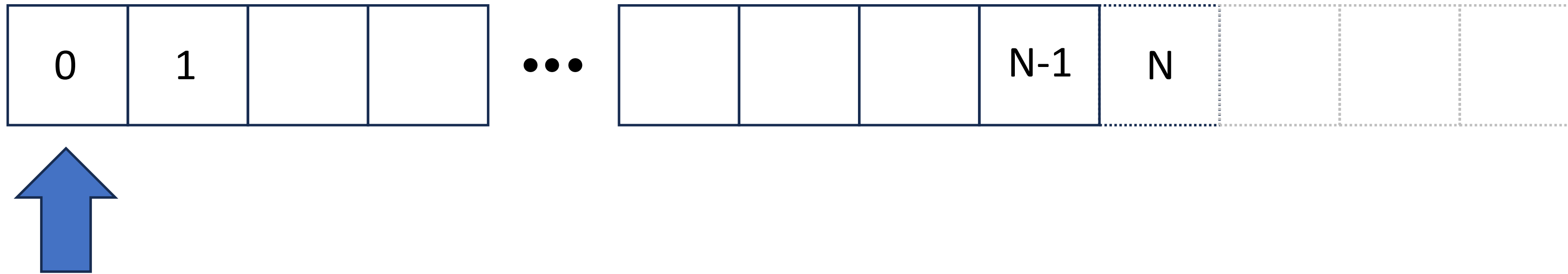
```
define void @cpy(ptr %dst, ptr %src) {
entry:
       %val1 = load i64, ptr %src
       store i64 %val1, ptr %dst
       %src2 = gep i32, ptr %src, i32 8
       %dst2 = gep i32, ptr %dst, i32 8
       %val2 = load i64, ptr %src2
       store i64 %val2, ptr %dst2
}
```

Might be copying capabilities

```
define void @cpy(ptr %dst, ptr %src) {
entry:
       call i32 @memcpy(ptr %dst, ptr %src,
                        i32 16)
}
```

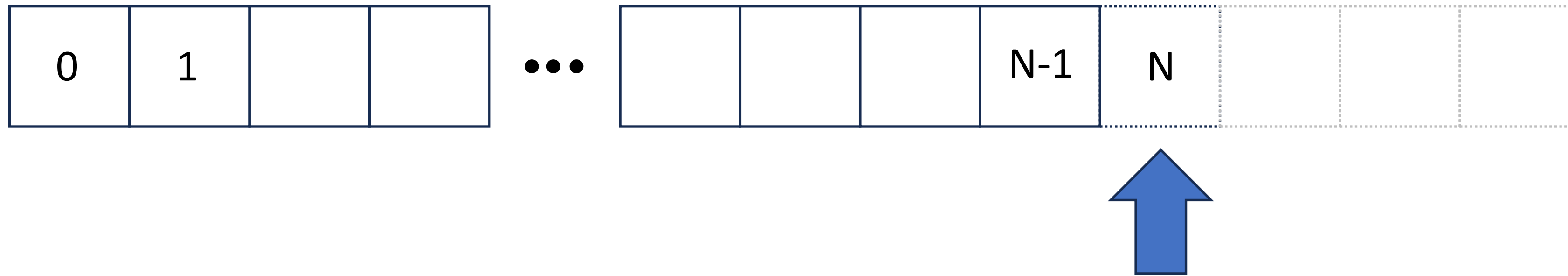
Dynamic capability-preserving memcpy can
be large and expensive!

LSR: Imprecise bounds restrict transforms



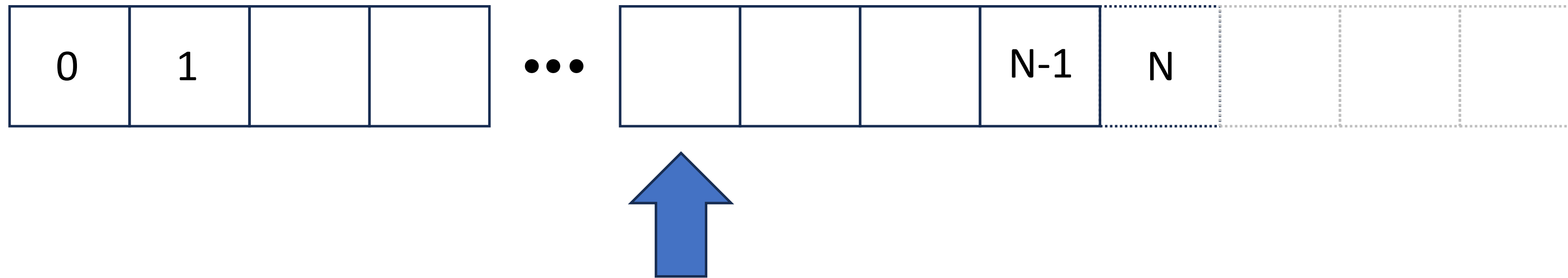

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

LSR: Imprecise bounds restrict transforms



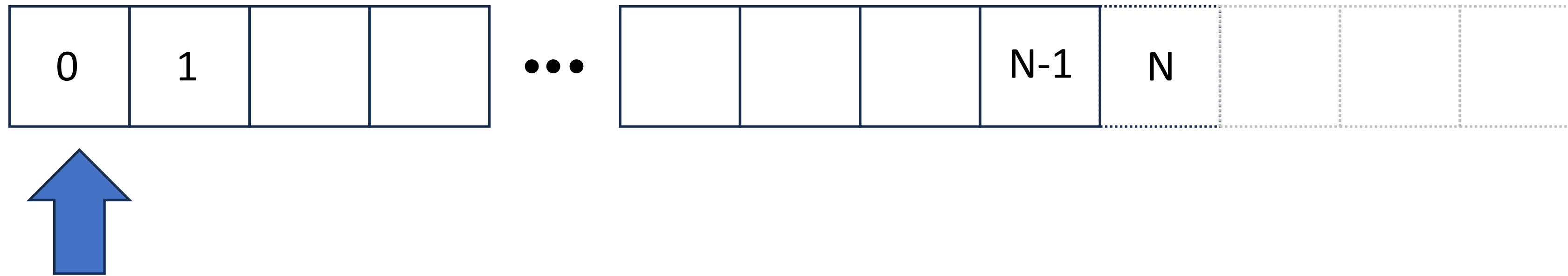

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

LSR: Imprecise bounds restrict transforms



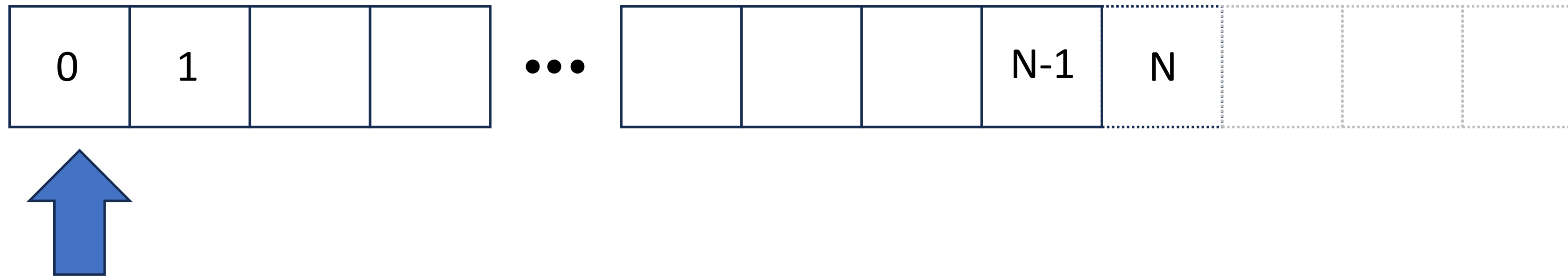

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

LSR: Imprecise bounds restrict transforms




```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

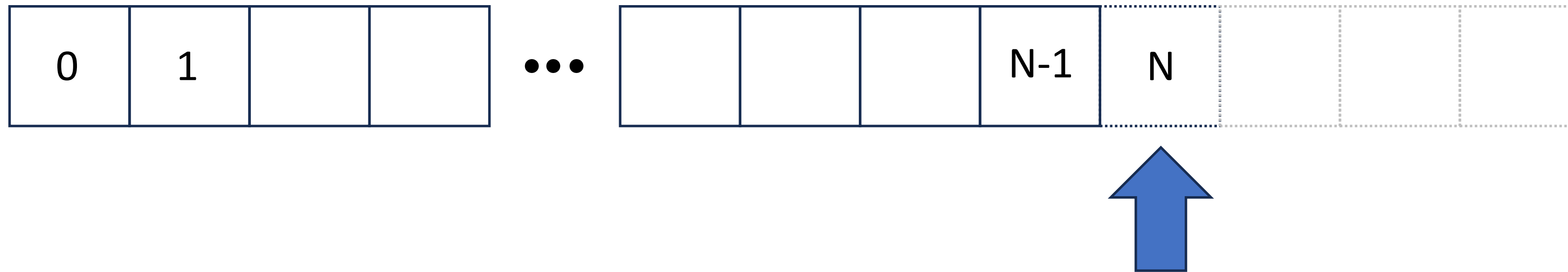
LSR: Imprecise bounds restrict transforms




```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

```
for (int *p = base + N; p >= base; p -= 4) { use(p- …); }
```

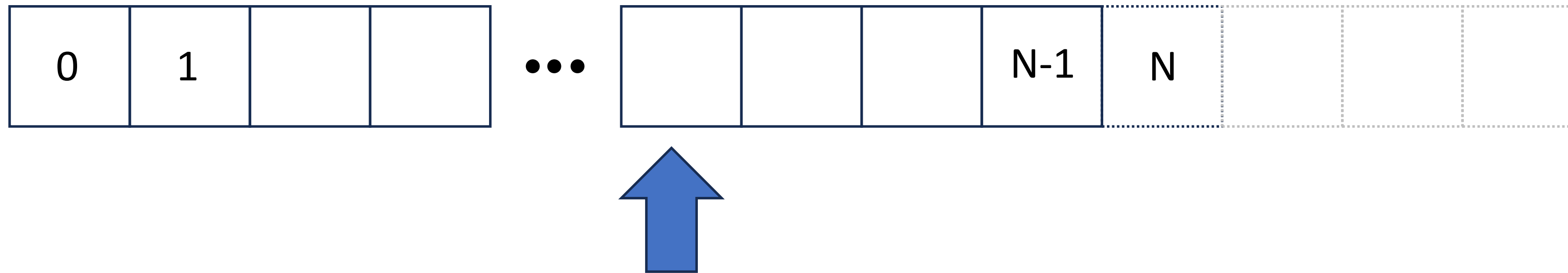
LSR: Imprecise bounds restrict transforms




```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }
```

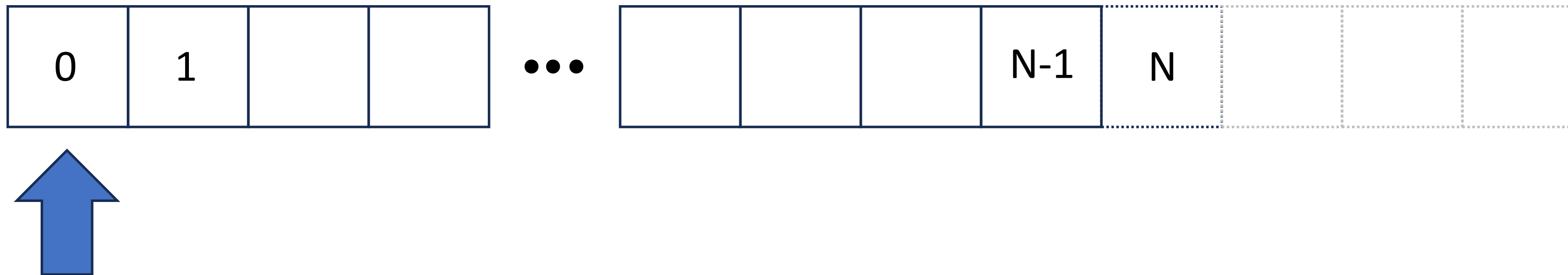
```
for (int *p = base + N; p >= base; p -= 4) { use(p- …); }
```

LSR: Imprecise bounds restrict transforms



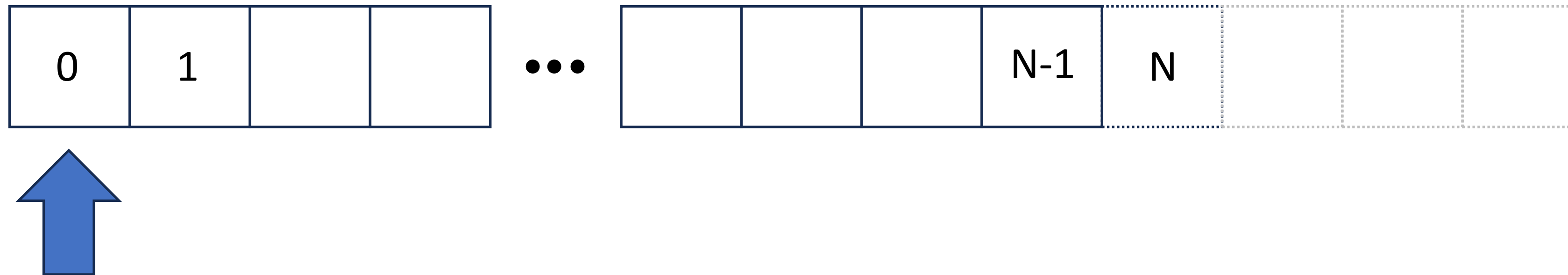

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }

for (int *p = base + N; p >= base; p -= 4) { use(p- …); }
```

LSR: Imprecise bounds restrict transforms




```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }

for (int *p = base + N; p >= base; p -= 4) { use(p- …); }
```

LSR: Imprecise bounds restrict transforms



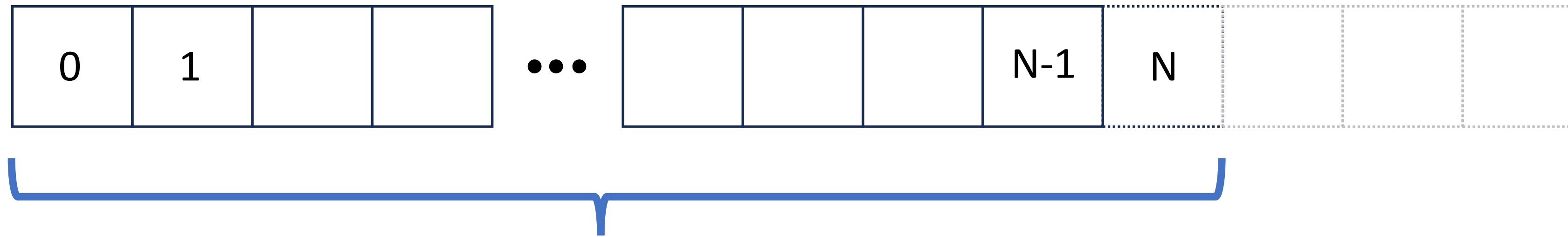

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }

for (int *p = base + N; p >= base; p -= 4) { use(p- …); }
```

`int *p = base + N + 4; do { p -= 4; use(p - …); } while (p >= base);`

LSR: Imprecise bounds restrict transforms

```
for (int p = N; p >= 0; p -= 4) { use(base[p- …]; }

for (int *p = base + N; p >= base; p -= 4) { use(p- …); }

int *p = base + N + 4; do { p -= 4; use(p - …); } while (p >= base);
```

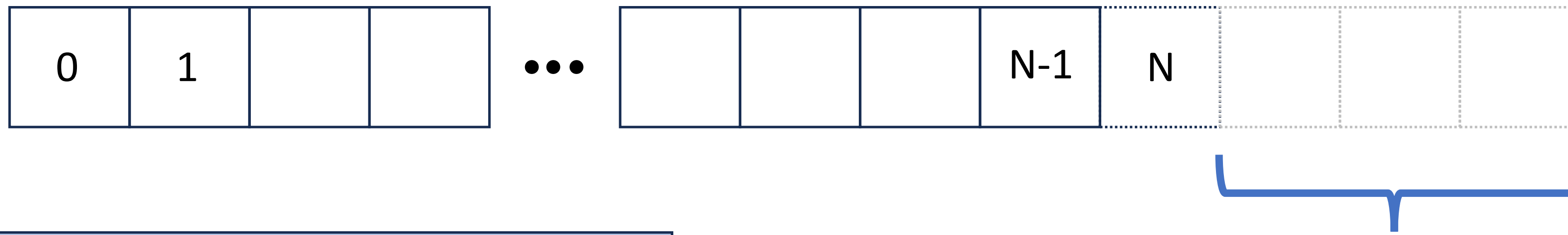
LSR: Imprecise bounds restrict transforms



Array offsets up to one-past-the-end
are always representable

Bounds checking happens at
dereference time

LSR: Imprecise bounds restrict transforms



Deterministic, but depends on the specific capability format!

Easier to hit with smaller capability sizes, like CHERIoT

Eventually array offsets exceed the representable range

The capability is marked invalid during offsetting, and can never be restored even if it logically comes back in range

CHERI touches all layers of the toolchain!

Clang	• Different lowering, new features
Mid-level optimisers	• Preserving pointer ≠ integer • Lowering
Code generation	• **New ISA**
LLD	• New relocations, compartment linkage model
Debugger	• More information about pointers

CodeGen Infrastructure Changes

SelectionDAG changes that mirror optimizer changes

- Capability MVTs
- ISD::PTRADD
- DAGCombine CHERI-safety
- CHERI-aware lowering of atomics, memcpy, etc.

Most changes are in the ISA extensions, and associated support.

CHERI extensions in LLVM downstreams

Only supported in simulators

Deprecated but still provides test coverage

CHERI-Mips

Mips

CHERI extensions in LLVM downstreams

Simulators and
development boards
available, but limited

CHERI-Mips

CHERI-Morello

Mips

AArch64

CHERI extensions in LLVM downstreams

Available in simulators and in Codasip licensable cores

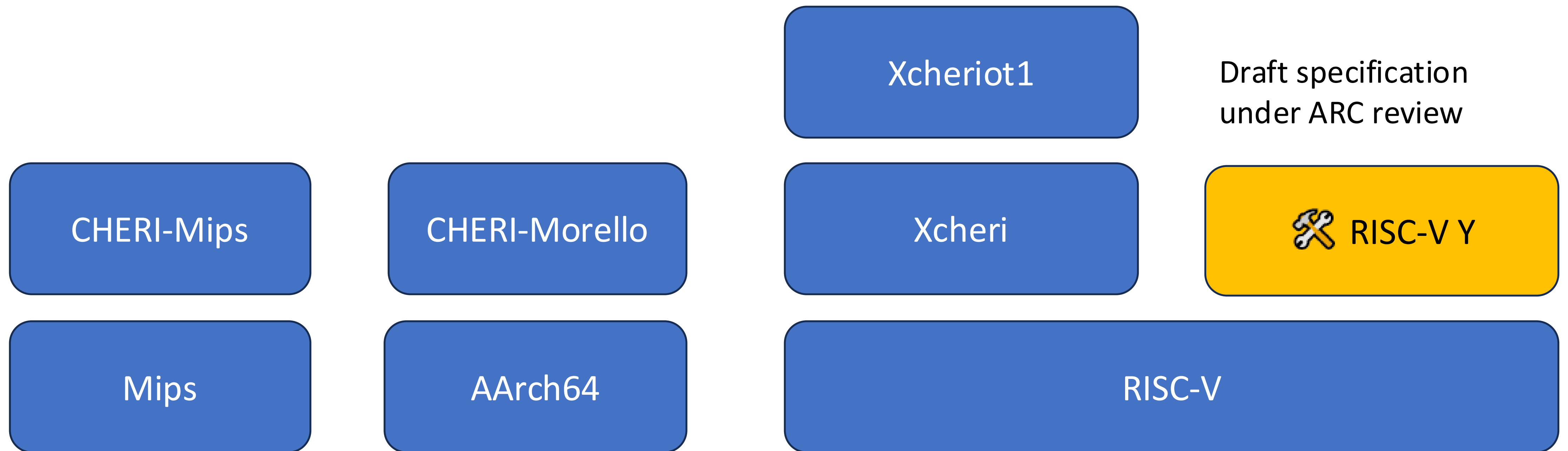
CHERI-Mips	CHERI-Morello	Xcheri
Mips	AArch64	RISC-V

CHERI extensions in LLVM downstreams

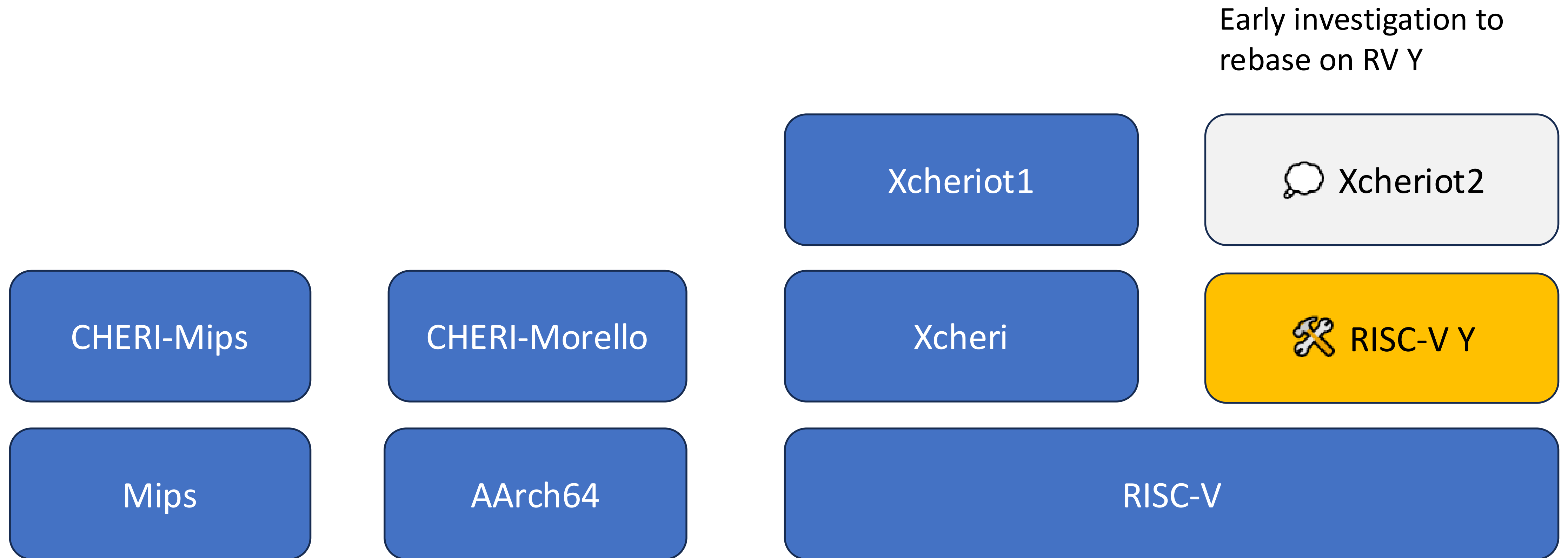
Available in simulators,
SCI ICENI available
commercially in 2026

		Xcheriot1
CHERI-Mips	CHERI-Morello	Xcheri
Mips	AArch64	RISC-V

CHERI extensions in LLVM downstreams



CHERI extensions in LLVM downstreams



CHERI touches all layers of the toolchain!

Layer	Changes
Clang	• Different lowering, new features
Mid-level optimisers	• Preserving pointer ≠ integer • Lowering
Code generation	• New ISA
LLD	• **New relocations, compartment linkage model**
Debugger	• More information about pointers

CHERI relocations must do more

Address / displacement can be baked into the binary

Dynamic relocations must provide information for deriving a valid capability (address, base, top, permissions)

CHERI relocations must do more

```
int x = 1;
void foo() {
      x = 0;
      static void * volatile p = &&label;
      label: use(p);
}
```

CHERI relocations must do more

```c
int x = 1;
void foo() {
      x = 0;
      static void * volatile p = &&label;
      label: use(p);
}
```

```
foo:
    lui     a0, %hi(x)
    sw      x0, %lo(x)(a5)
.Ltmp0:
    lui     a0, %hi(foo.p)
    lw      a0, %lo(foo.p)(a0)
    tail    use
```

```
.data
x:
    .word   1
foo.p:
    .word   .Ltmp0
```

CHERI relocations must do more

```c
int x = 1;
void foo() {
      x = 0;
      static void * volatile p = &&label;
      label: use(p);
}
```

```
foo:
    lui     a0, %hi(x)
    sw      x0, %lo(x)(a5)
.Ltmp0:
    lui     a0, %hi(foo.p)
    lw      a0, %lo(foo.p)(a0)
    tail    use
```

```
.data
x:
    .word   1
foo.p:
    .word   .Ltmp0
```

CHERI relocations must do more

```c
int x = 1;
void foo() {
      x = 0;
      static void * volatile p = &&label;
      label: use(p);
}
```

```
foo:
    auicgp    a0, %hi(x)
    csw       x0, %lo(foo)(a0)
.Ltmp0:
    auicgp    a0, %hi(foo.p)
    clc       a0, %lo(.Ltmp0)(a0)
    ctail     use
```

```
.data
x:
    .word   1
foo.p:
    .chericap %code(foo + (.Ltmp0 - foo))
```

CHERI relocations must do more

```
foo:
    auicgp    a0, %hi(x)          (struck through)
    csw       x0, %lo(foo)(a0)    } Relaxable Sequence
.Ltmp0:
    auicgp    a0, %hi(foo.p)
    clc       a0, %lo(.Ltmp0)(a0)
    ctail     use
```

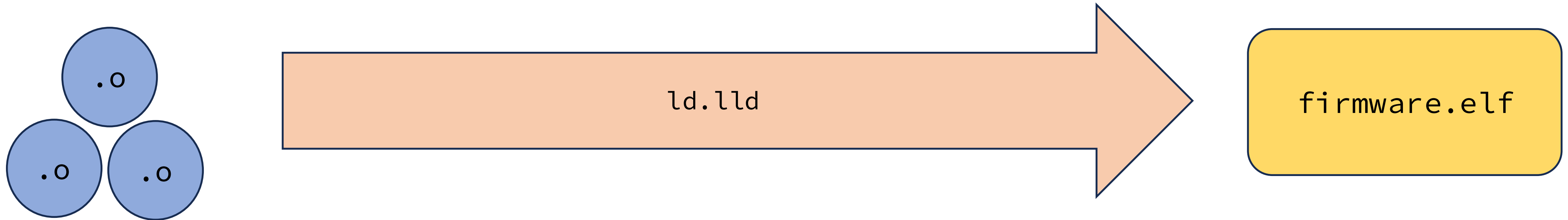
```
.data
x:
    .word   1
foo.p:
    .chericap %code(foo + (.Ltmp0 - foo))
```

`foo` → Bounds + Permissions; `(.Ltmp0 - foo)` → Symbolic Offset

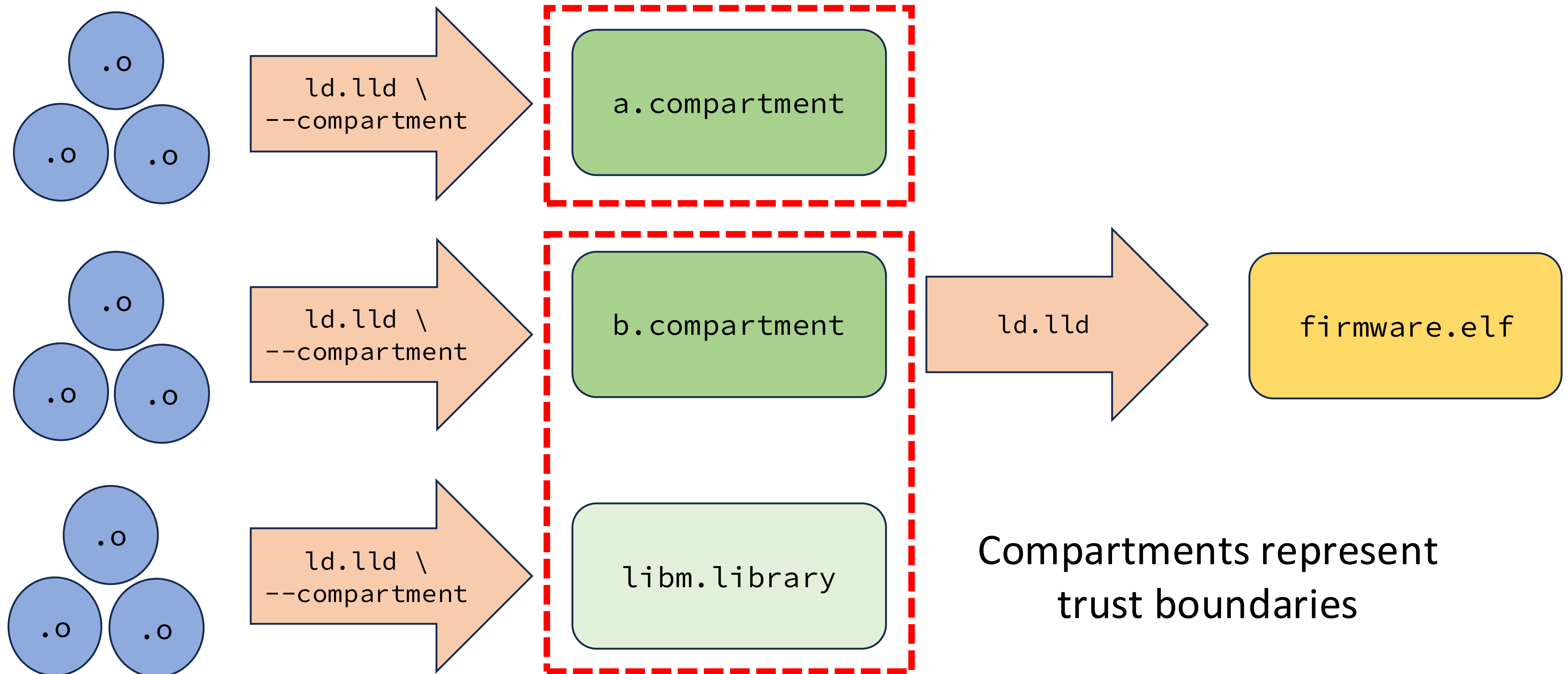
- Label pointer needs bounds and permissions → Needs correct provenance
- Correct bounds and permissions (provenance) comes from the function pointer
- The offset from the function pointer depends on link-time relaxation → Must be symbolic

- But LLVM's `MCValue` can only represent `SymA - SymB + Const` today!

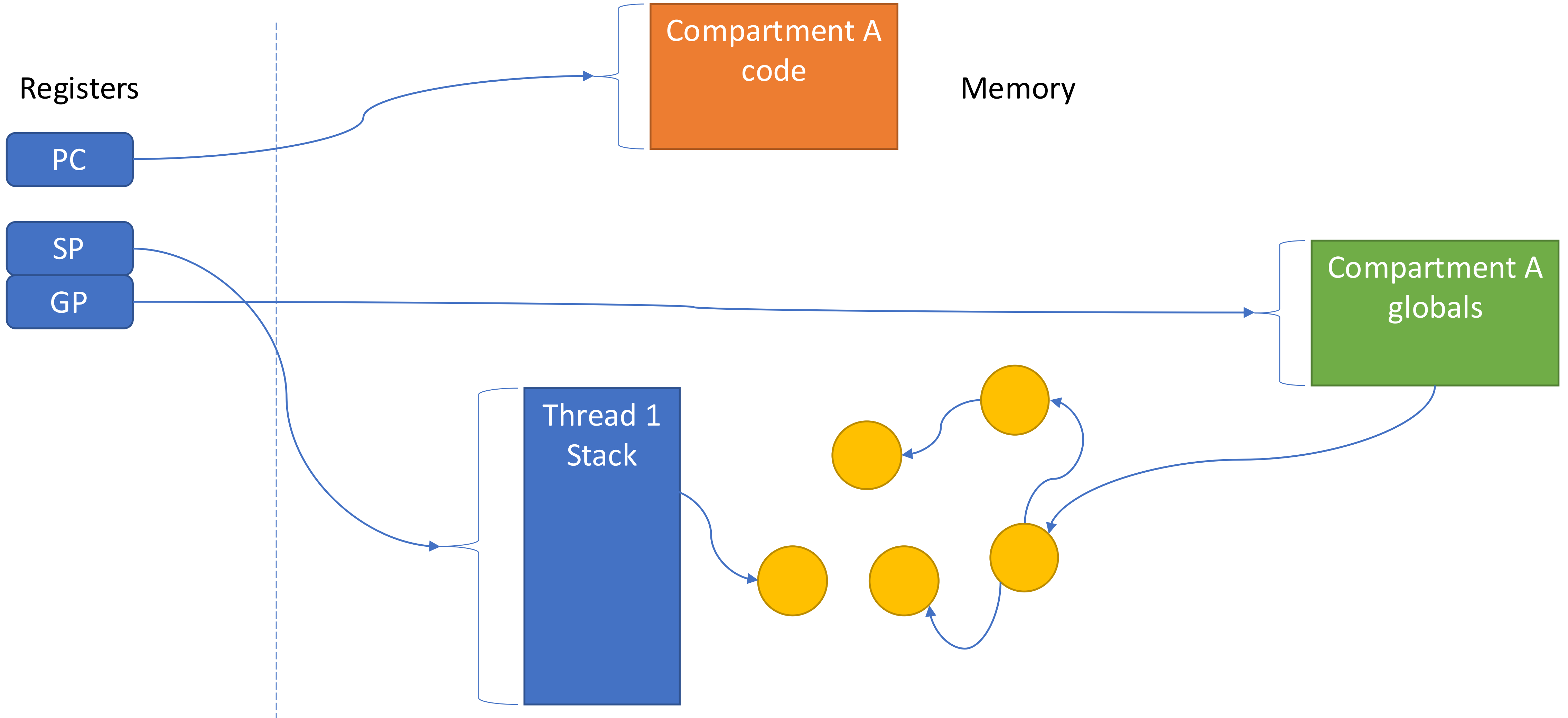
Standard Unix-like Linking Model



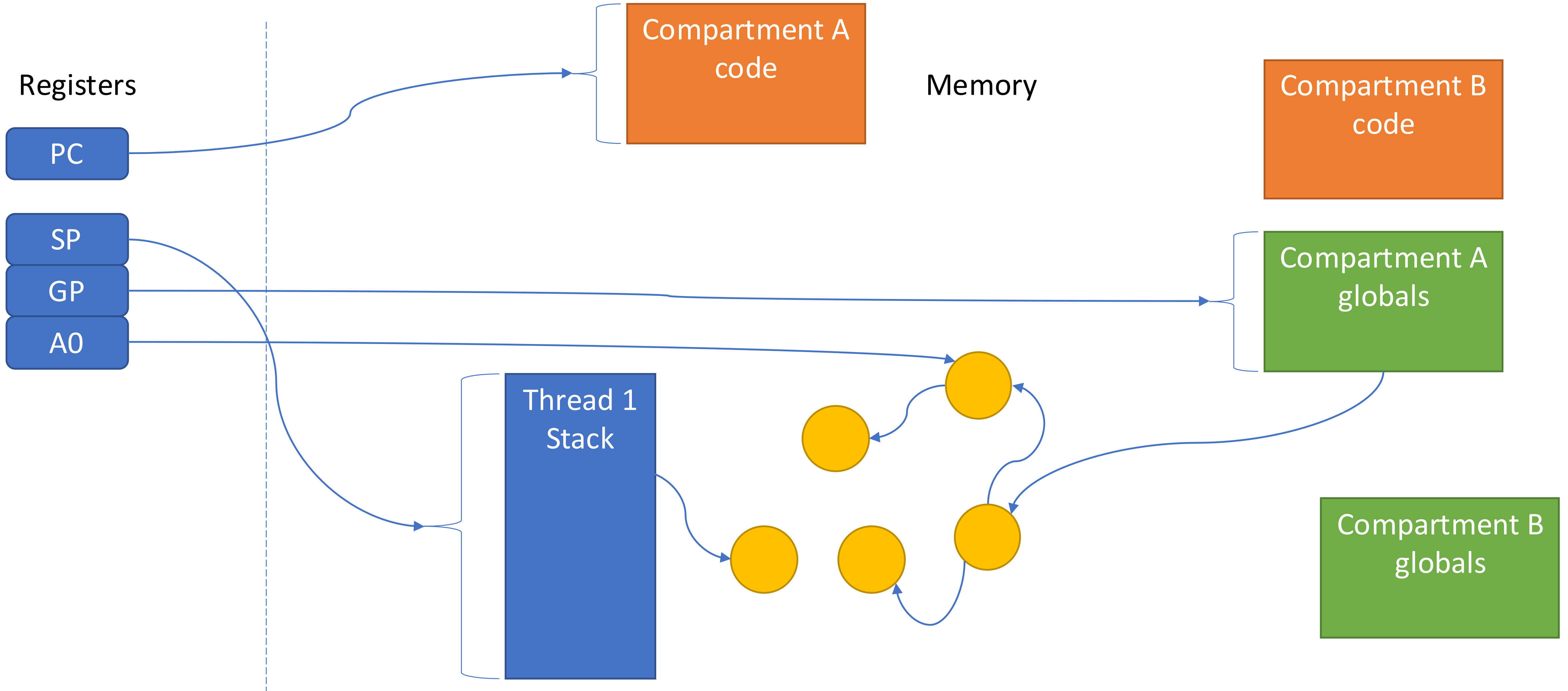
CHERIoT Compartment Linking Model



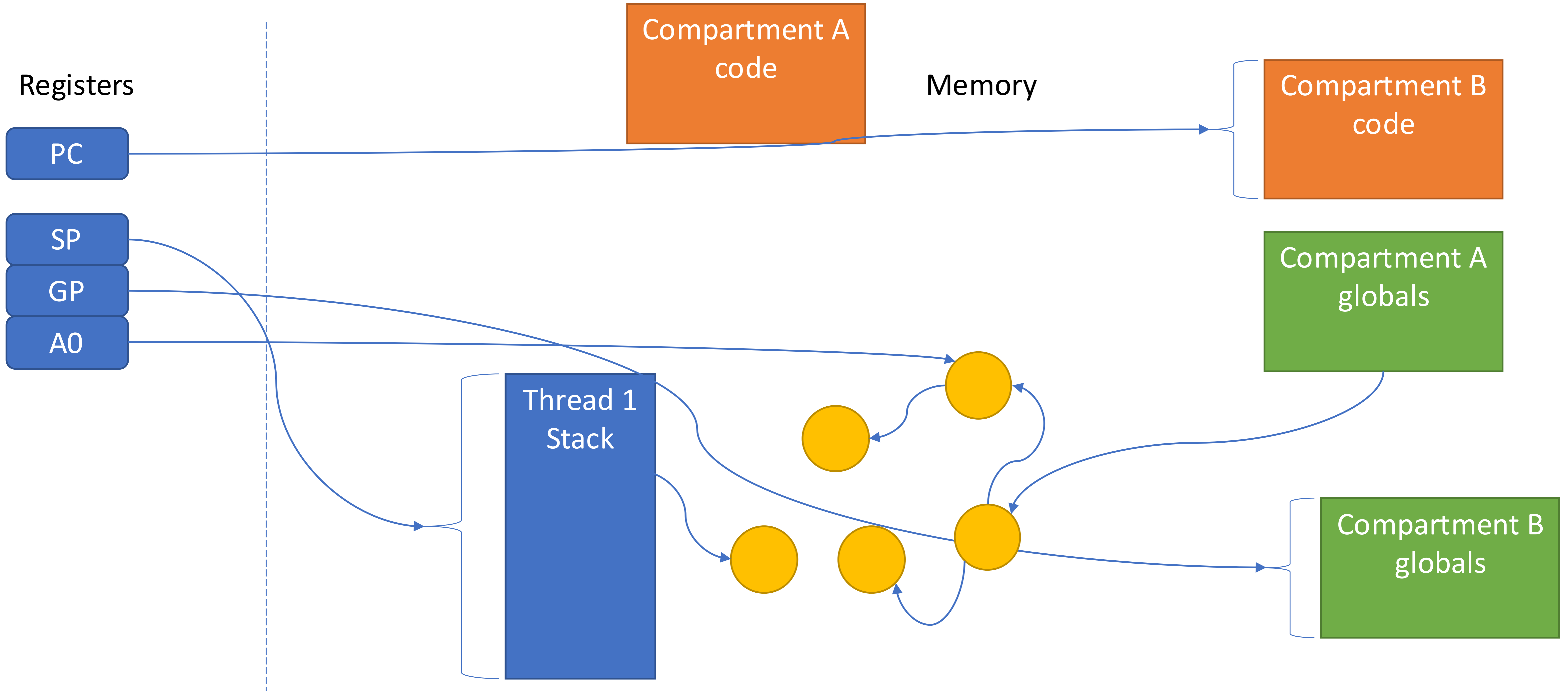
Cross-compartment calls are trust boundaries



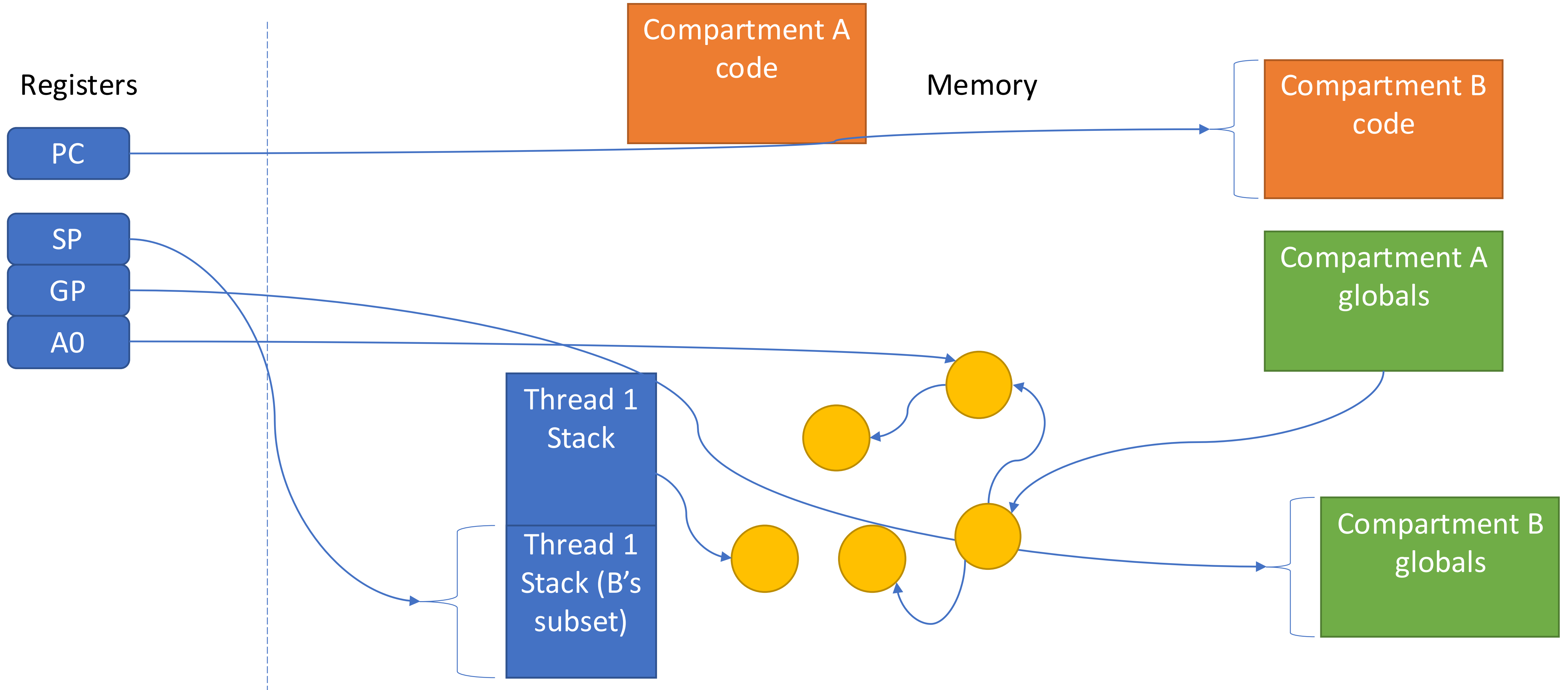
Cross-compartment calls are trust boundaries



Cross-compartment calls are trust boundaries



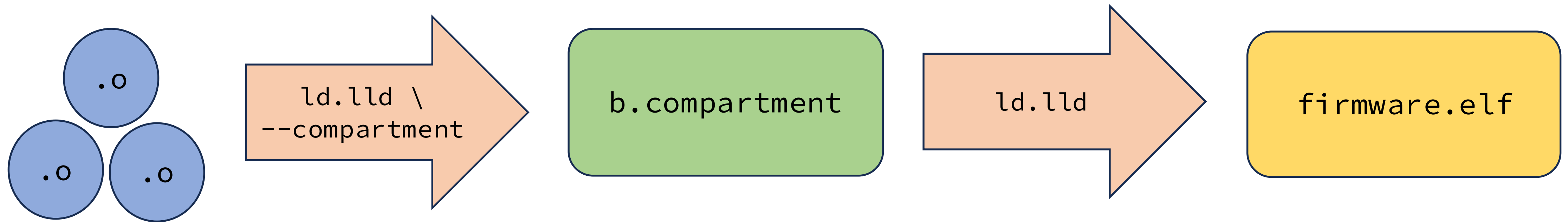
Cross-compartment calls are trust boundaries



Add compartmentalization to C/C++

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


// Call site looks like normal C.
uint8_t buffer[BUFFER_SIZE];
publish("key_id", buffer, sizeof(buffer));
```

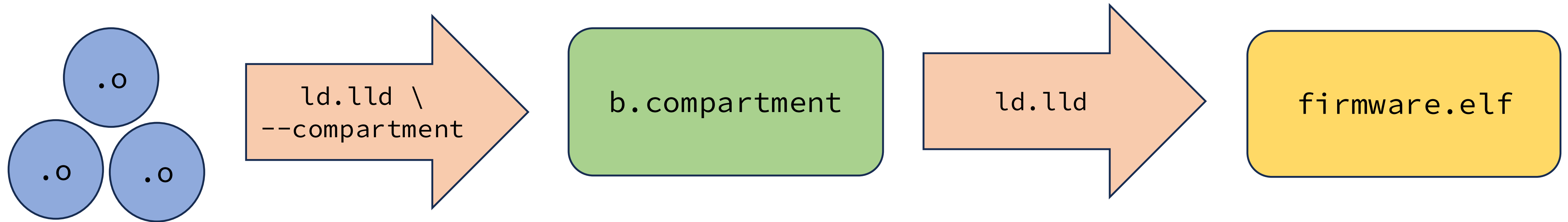
CHERIoT Compartment Linking



- Internalize non-exported symbols
- Layout `.text`, `.rodata`, `.data` individually per compartment
- Deduplicate COMDATs

- Relaxation
- Layout final image
- Emit compartment audit report

CHERIoT Compartment Linking - Future



- Internalize non-exported symbols
- Relaxation
- Layout `.text`, `.rodata`, `.data` ~~individually~~ together per compartment
- Deduplicate COMDATs

- ~~Relaxation~~
- Layout final image
- Emit compartment audit report

CHERI touches all layers of the toolchain!

Clang	• Different lowering, new features
Mid-level optimisers	• Preserving pointer ≠ integer • Lowering
Code generation	• New ISA
LLD	• New relocations, compartment linkage model
Debugger	• **More information about pointers**

Next Steps

CHERI in LLVM Upstream

Prior Work

- addrspace correctness, non-integral pointers
- ISD::PTRADD

Already Landed

- Calling convention enums, MVT capability types
- Xcheriot vendor relocations
- RISC-V Y base register file

Upcoming

- RISC-V Y base instructions, Xcheriot instructions

Q&A

Co-authors

- Owen Anderson (SCI Semi)
- David Chisnall (SCI Semi)
- Jessica Clarke (Cambridge)
- Alex Richardson (Google)

Many other contributions

- University of Cambridge
- Microsoft Research
- ARM
- Google
- Codasip
- SCI Semi