RUST OR CHERI?

Edoardo Marangoni · SCI Semiconductor

EuroLLVM · April 2026

WHAT?

CHERI

Addresses the lack of memory safety

RUST

A solution to the same problems?

“Choose one”

WHAT?

CHERI

Addresses the lack of memory safety

RUST

A solution to the same problems?

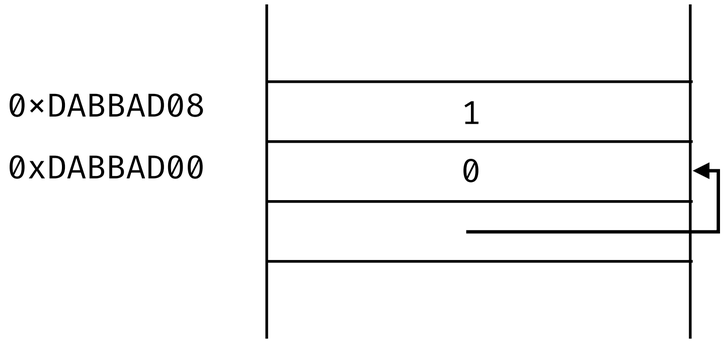
Choose both!

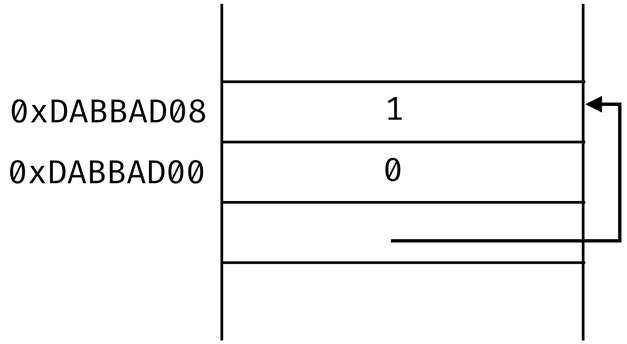
WHAT?

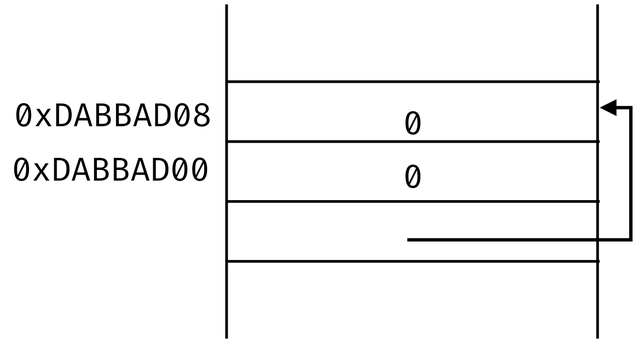
```
int32_t foo = 1;
int32_t bar = 0;
int32_t *baz = &bar;
```

```
…
baz = baz + (sizeof(bar))
```

```
…
*baz = 0
```







Pointers are plain integers: if you know where to point them you can do what you want

```
baz = baz + (sizeof(bar))
```

```
%0 = load ptr, ptr %baz, align 8
%add.ptr = getelementptr inbounds nuw i32, %0, i32 8
store ptr %add.ptr, ptr %baz, align 8
```

```
ct.clc      a1, -48(s0)
ct.cincoffset          a1, a1, 64
ct.csc      a1, -48(s0)
```

```
*baz = 0
```

```
%1 = load ptr, ptr %baz, align 8
store i32 0, ptr %1
```

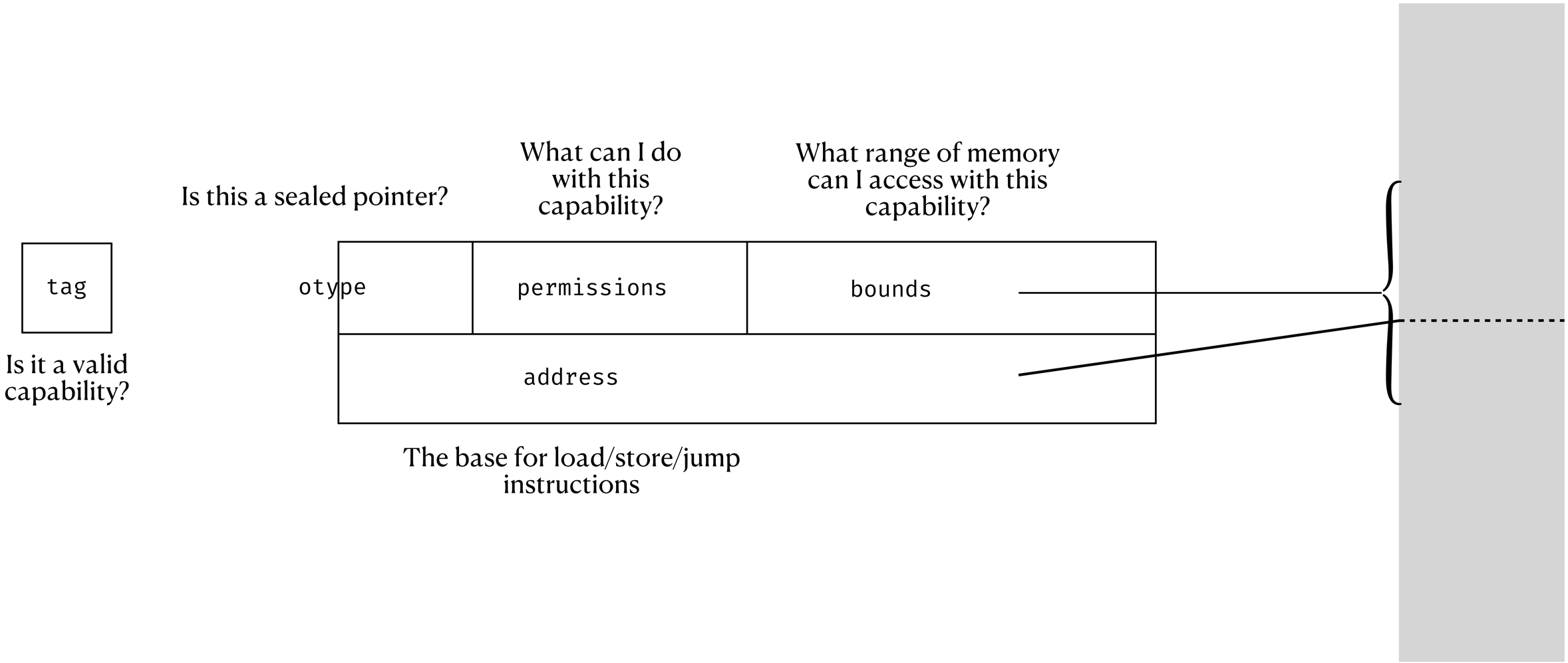
```
ct.clc   a1, -48(s0)
ct.csw   a0, 4(a1)
```

```
BoundsViolation(0x1) error at … with capability …: 0x800067e8 (v:1
0x800067e8-0x800067f0 l:0x4 o:0x0 p: - RWcgml -- ---)
```

On CHERI platforms pointers are more than that!

WHAT?

Anatomy of a capability



CHERIoT is a RV32E platform, and capabilities are 64 bits.

CHERI gives you spatial safety!

```
char stack[] = "short buffer”;
stack[sizeof(stack)] = ‘\0’;
```

```
static char stack[] = "short buffer”;
stack[sizeof(stack)] = ‘\0’;
```

```
char *heap = new char[16];
heap[sizeof(heap)] = ‘\0’;
```

These all deterministically trap on CHERI.

CHERI also gives you temporal safety!

```
…
free(heap);
heap[0] = 'H';
```

This deterministically traps on CHERIoT.

CHERIoT and its RTOS also let you handle these traps.

- A general-purpose systems language
- Builds on previous research on programming languages
- ...But isn't a research programming language!
- Automatic memory management without GC
- Expressive type system (System F, type classes, regions, linear/affine)
- You've heard of it at least once

Rust gives you spatial safety!

```rust
let mut stack = [0, 1, 2, 3];
stack[4] = 0;
```

```rust
static mut stack: [u8; 4] = [0, 1, 2, 3];
stack[4] = 0;
```

```rust
let mut heap = vec![0, 1, 2, 3];
heap[4] = 0;
```

```
$ cargo build
error: this operation will panic at runtime
index out of bounds: the length is 4 but the index is 4
```

Rust also gives you temporal safety!

```rust
drop(heap);
heap[0] = 10;
```

```
$ cargo build
error[E0382]: borrow of moved value: `heap`
```

WHY?

WHY?

SOUNDS LIKE
RUST WITH
EXTRA STEPS.

JUST
RIIR.

You don't need CHERI or CHERIoT if:

1. Your entire software stack is written in safe Rust (Rust's libraries as well) only, and
2. Your Rust implementation does not have any soundness holes.

WHY?

Greater than the sum of its parts

	RUST	CHERI
Your code	Rich compile-time guarantees for safe fragments Expressive type system	Non-bypassable runtime guarantees Easy compartment restart for availability Guarantees extend to any language
Dependencies	Rich interfaces for non-malicious libraries Mature package management tools like cargo	Strong isolation of untrusted code Auditable rights for third-party components

RUST & CHERI

Non-bypassable runtime guarantees

Rich compile-time guarantees for safe fragments

Easy compartment restart for availability

Expressive type system

Guarantees extend to any language

Rich interfaces for non-malicious libraries

Strong isolation of untrusted code

Mature package management tools like cargo

Auditable rights for third-party components

Demo 1: off-by-one

```
let v = [0, 0];
let indices = [1, 2];
let mut count = 0;
for i in indices {
  unsafe {
    count = count + *v.get_unchecked(i);
  }
}
```

On other platforms:

> *Calling this method with an out-of-bounds index is undefined behaviour even if the resulting reference is not used.*
>
> source

Demo 2: Rust-to-C FFI

```c
extern "C" int verify(char *s) {
  int len = strlen(s);
  …
  return valid_len;
}
```

```rust
unsafe {
 let payload: String = get();
 let valid_len = verify(payload.as_ptr());
 let valid = str::from_raw_parts(payload.as_ptr(), valid_len);
}
```

On other platforms this is, again, undefined behaviour

RUST & CHERI

Non-bypassable runtime guarantees

Rich compile-time guarantees for safe fragments

Easy compartment restart for availability

Expressive type system

Guarantees extend to any language

Rich interfaces for non-malicious libraries

Strong isolation of untrusted code

Mature package management tools like cargo

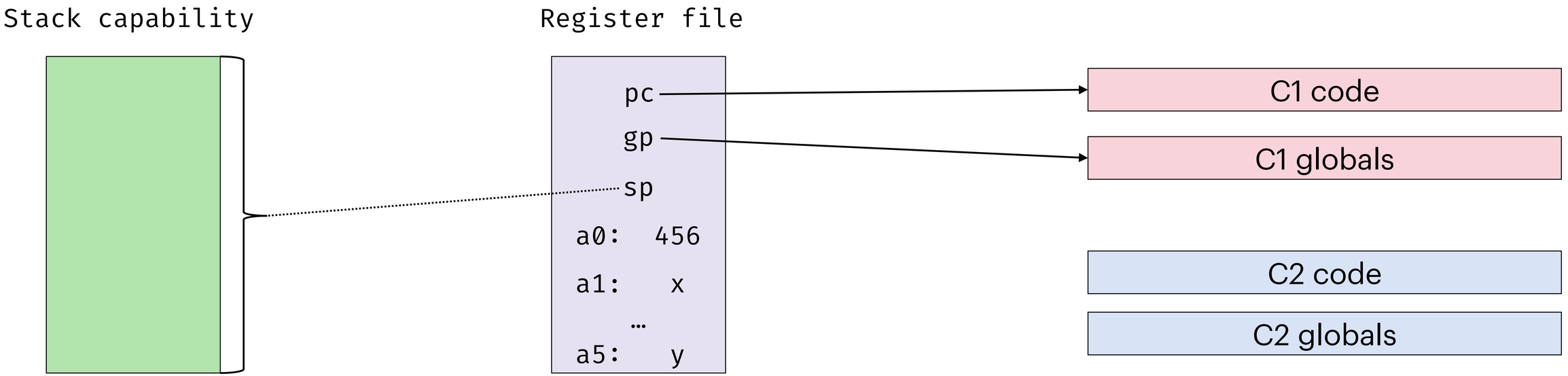
Auditable rights for third-party components

Strong isolation of untrusted code (compartmentalisation)

```
int __cheriot_compartment("C1") entry()
{
        print(456);
        return 0;
}
```

→ Switcher

```
int __cheriot_compartment("C2") print(int msg)
{
        Debug::log("{}", msg);
        return 0;
}
```

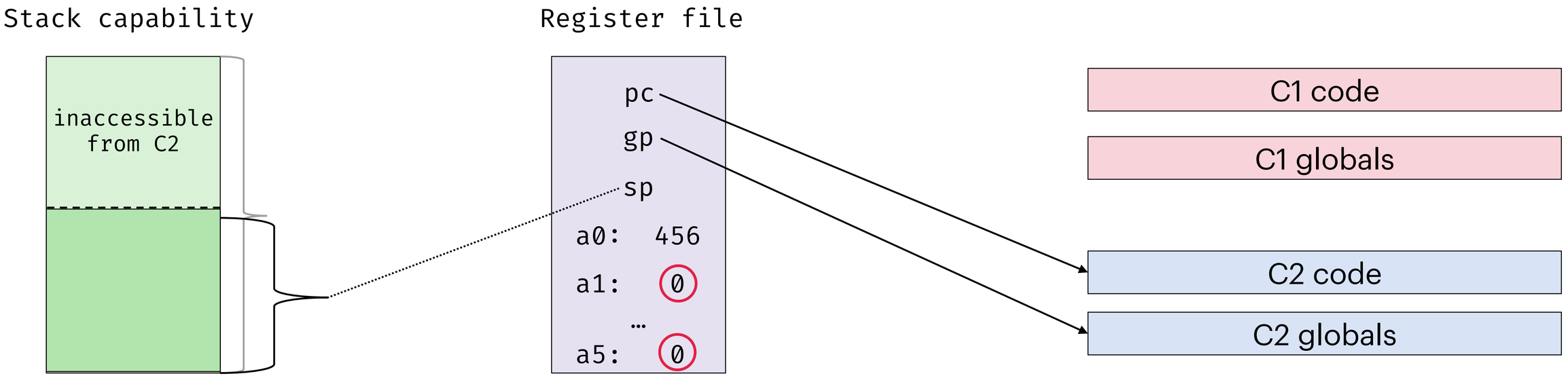


Strong isolation of untrusted code (compartmentalisation)

```
int __cheriot_compartment("C1") entry()
{
        print(456);
        return 0;
}
```

→ Switcher →

```
int __cheriot_compartment("C2") print(int msg)
{
        Debug::log("{}", msg);
        return 0;
}
```



Demo 3: supply chain attack

```
let password = get_password();
let user = get_user();

log4r::record(“auth.request”, &user);

if allow(&user, &password) {
  . . .
}
```

c1

```
fn record(key: &str, value: &str) {
 println!("[kvlog] {key}-> {value}");
 if key == "auth.request" {
  unsafe {
   let addr = frameaddr();
   let ptr = addr.sub(..);
   let password: &String = transmute(ptr);
   send_username_password(&value, &password)
  }
 }
}
```

c2

On other platforms there’s nothing stopping `record` from accessing the caller’s stack

How?

In 2025 DSbD granted SCI Semiconductor funding to make Rust work on CHERIoT.

1. Adapt previous work on Morello to latest Rust releases
2. Add the CHERIoT target and CHERI intrinsics
3. Make `core` and `alloc` compile to it
4. Test `core` and `alloc` (might require building std too)
5. Add CHERIoT-specific frontend attributes (CC, MMIO, SharedObjects, ..)
6. Integration tests with the rest of the CHERIoT platform (the RTOS, networking stack..)

Required changes to the compiler

```
pub enum Primitive {
        Int(Integer, bool),
        Float(Float),
        Pointer(AddressSpace),
}
```

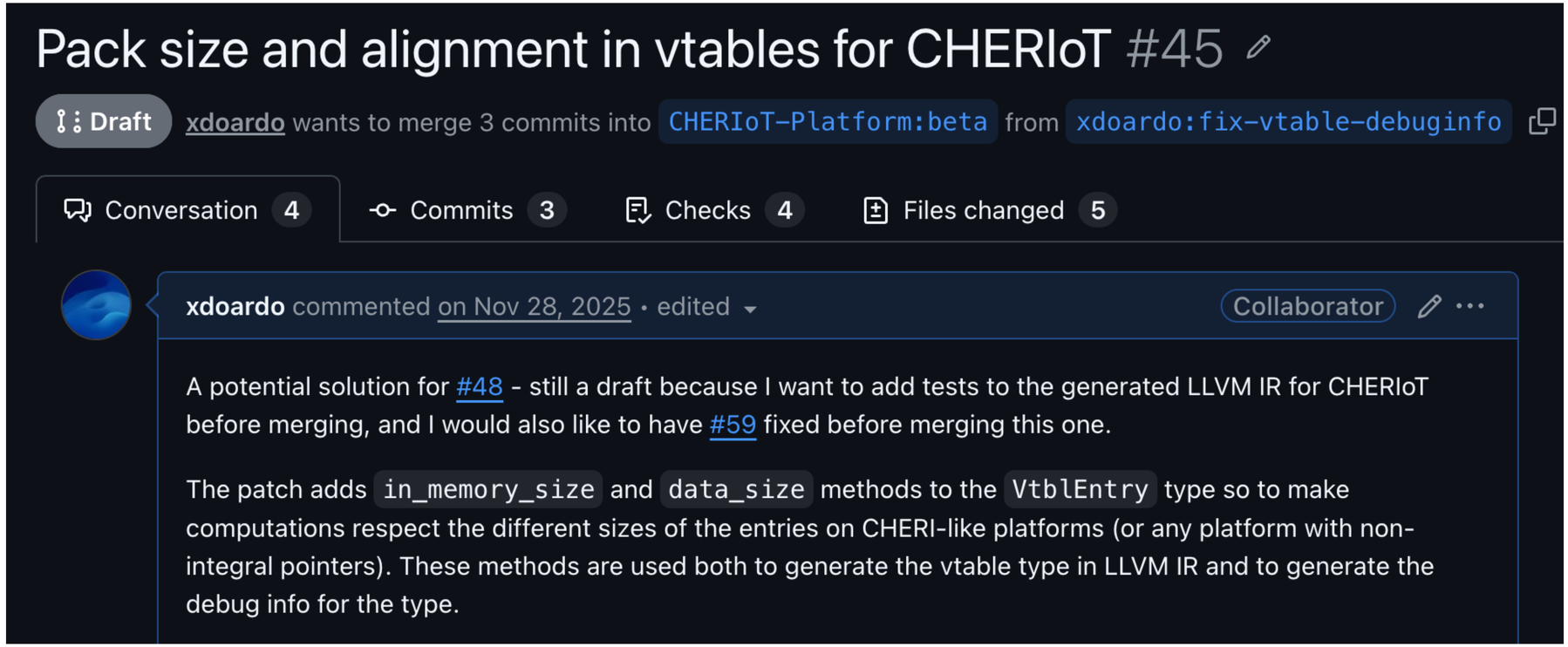
```
impl Primitive {
        pub fn size<C: HasDataLayout>(self, cx: &C) → Size
}
```

Required changes to the compiler

```rust
pub enum Primitive {
        Int(Integer, bool),
        Float(Float),
        Pointer(AddressSpace),
}
```

```rust
impl Primitive {
        // How many bits to store this value?
        pub fn in_memory_size<C: HasDataLayout>(self, cx: &C) -> Size
        // How many bits _of data_ can be stored here?
        pub fn capacity<C: HasDataLayout>(self, cx: &C) -> Size
}
```

Required changes to the compiler



How? ...evolving the crab...

```
$ ./x test tests/codegen-llvm –target=riscv32cheriot-unknown-cheriotrtos
test result: ok. 602 passed; 16 failed; 402 ignored
```

60% of codegen-llvm tests pass, 39% can't run on CHERIoT, 1% fail

```
$ cargo test library/coretests –target=riscv32cheriot-unknown-cheriotrtos
test result: ok. 1709 passed; ? failed; 1257 ignored
```

57% of coretests pass, 43% need investigation

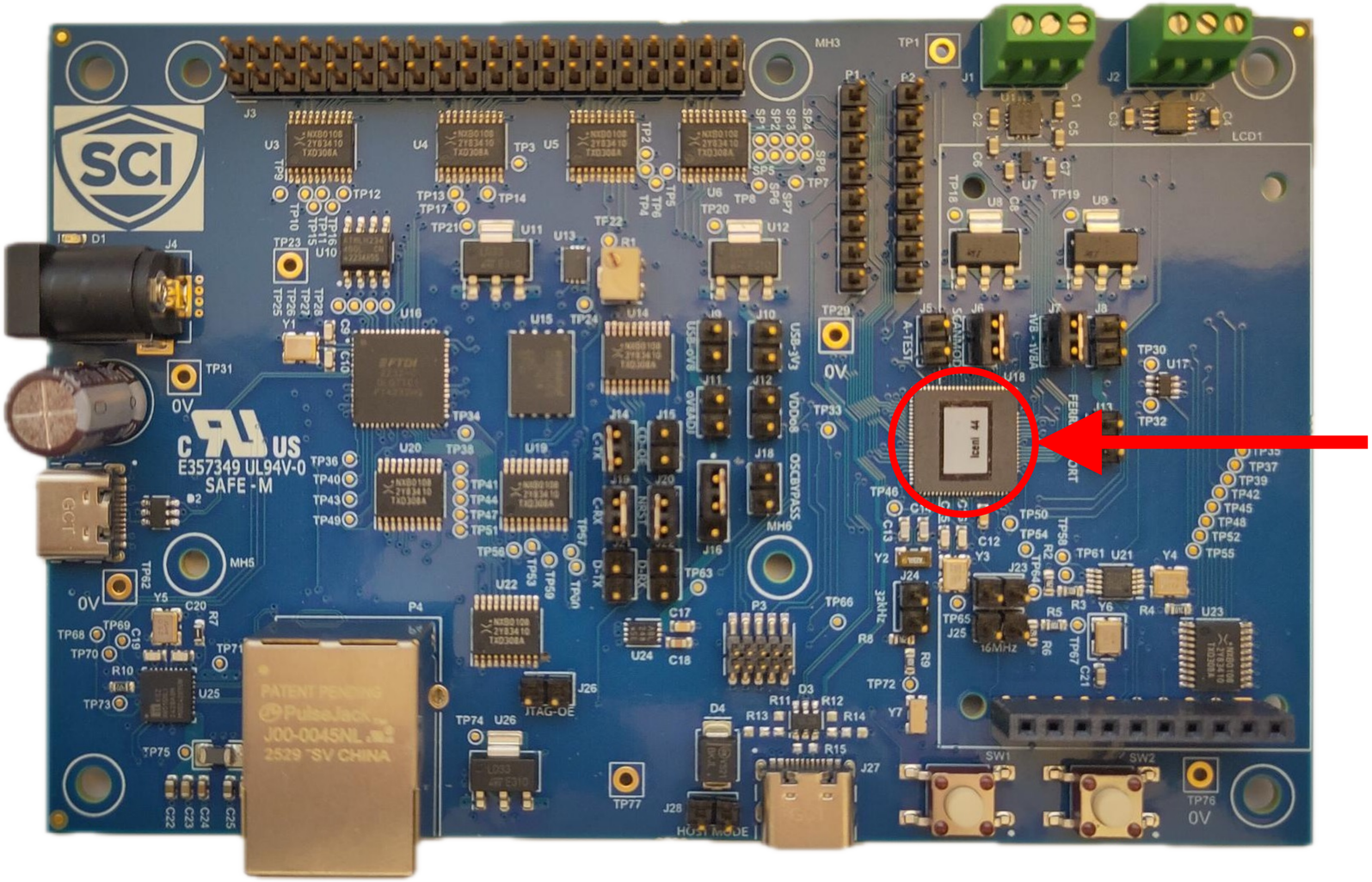
Making it work is only part of the story. We also need to have a plan for the future.

The cherry on top would be that of being able to upstream a new CHERI target and be able to maintain it in the future.

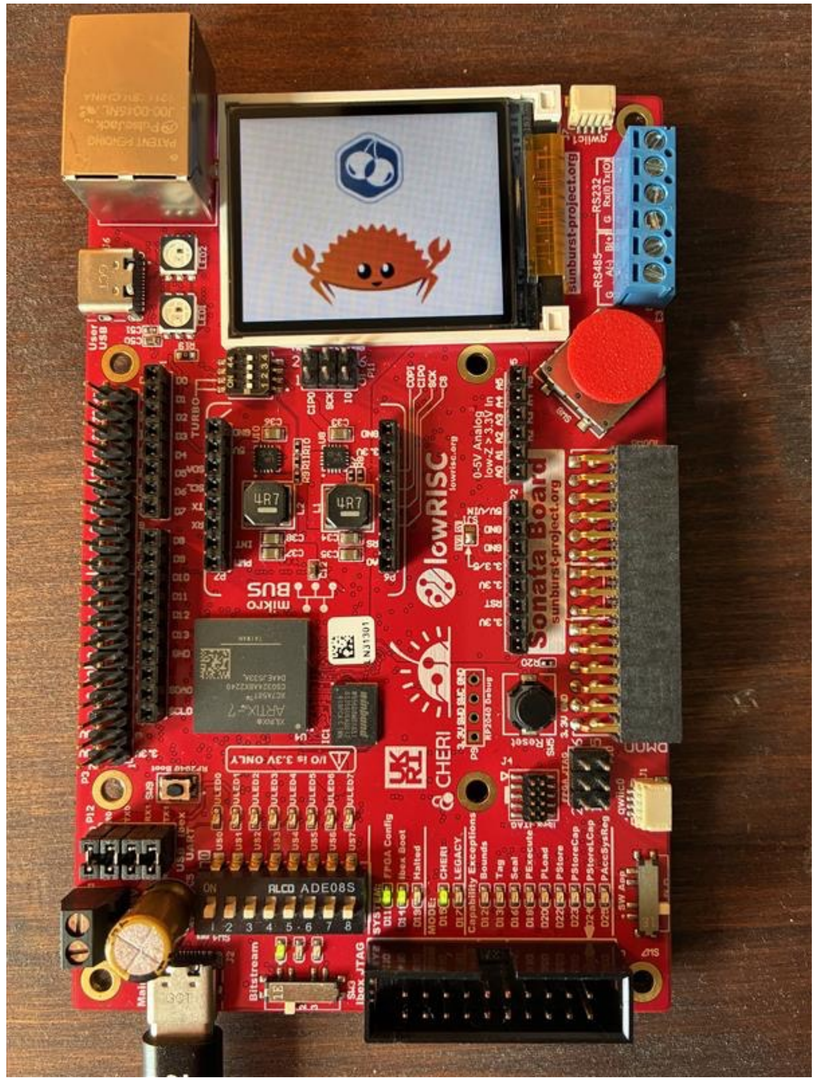
This is not an easy task: it requires a lot of expertise, designs and social skills!

WHERE?

WHERE?


SCI
E357349 UL94V-0
SAFE-M
PATENT PENDING
Pulse Jack
J00-0045NL
2529 SV CHINA
LCD1
SW1
SW2
JTAG-OE
HOST MODE
16MHz

WHERE?



Devcontainer with simulators, clang and rustc

[lowRISC Sonata](#) running embedded-graphics

WHERE?

All the work is open source, and lives in the CHERIoT-Platform/cheri-rust repository on GitHub.

THANK YOU!