Making Rust work on a CHERI platform

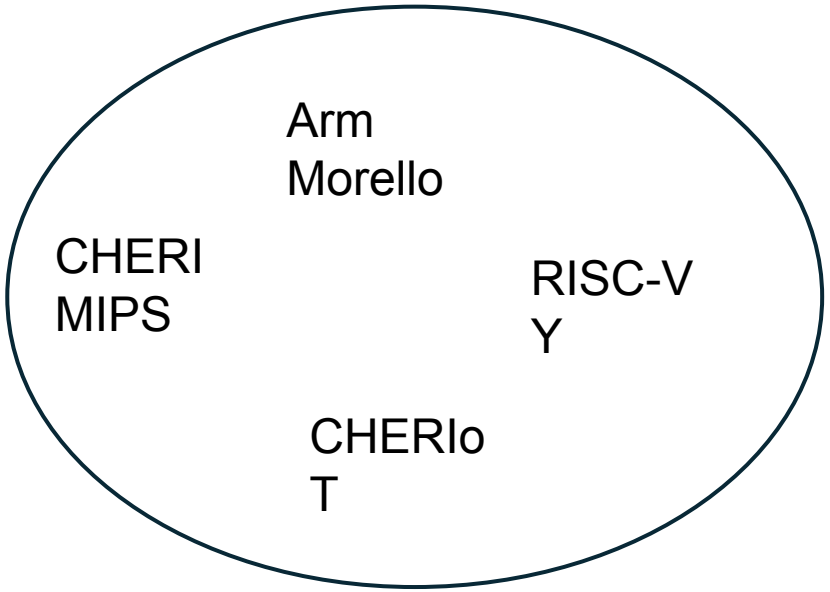
The plan

1. What is CHERI?
2. Why is it useful if we already have Rust?
3. What does it take for Rust to support a non-integral-pointer platform?
4. What can I do with Rust on CHERIoT today?
5. What's on the horizon?

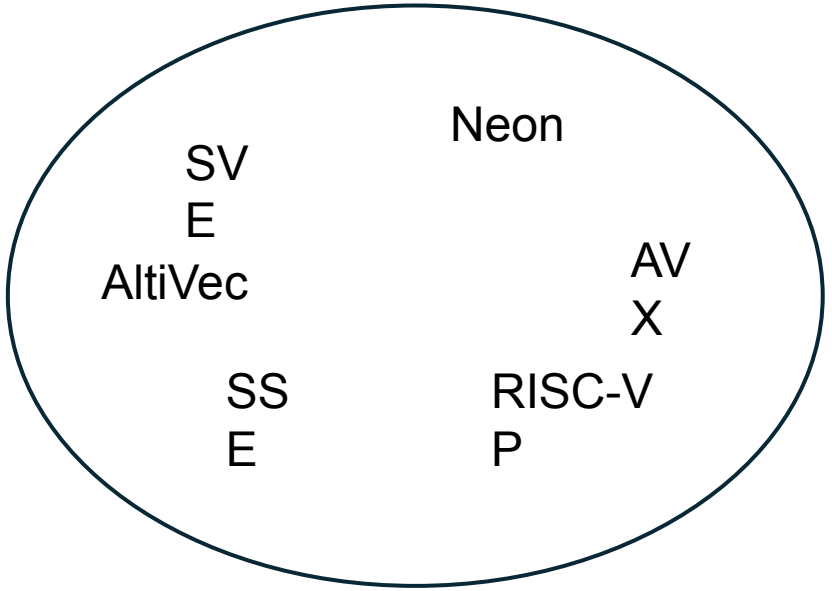
What is CHERI?

Introduction to CHERI

- The backronym: Capability Hardware Enhanced RISC Instructions
- A mature technology, >15 years of work
- Not an architecture: it is a feature that an architecture can implement!



CHERI
architectures

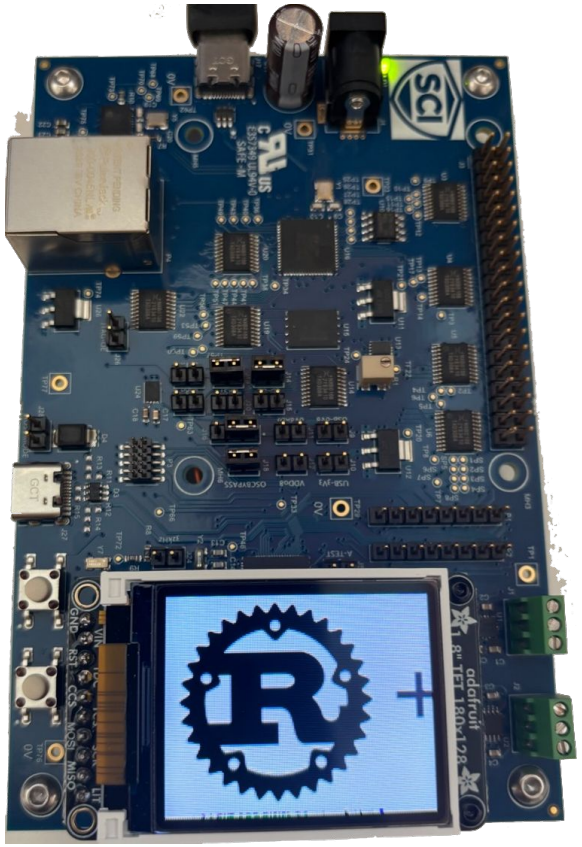


SIMD
architectures

Introduction to CHERI

CHERIoT:

- A particular instance of CHERI-enabled architectures
- Targeted at resource-constrained environments
- RISC-V32E base
- Originally developed at Microsoft, open-sourced in 2023



Introduction to CHERI

- CHERI enables memory safety directly at the hardware level
  - The hardware *knows* about pointers!
  - Pointers are called *capabilities*
- Capabilities cannot be created out of thin air
  - They must follow a valid chain of derivation
- Capabilities carry memory addresses and metadata
  - On a CHERI system all memory access instructions require a valid capability
- On CHERI platforms registers are generally capability-sized
  - On CHERIoT registers are 64+1 bits
  - CHERI processors do not have “lookaside” mechanisms to translate capabilities into pointers: capabilities *are* pointers!

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

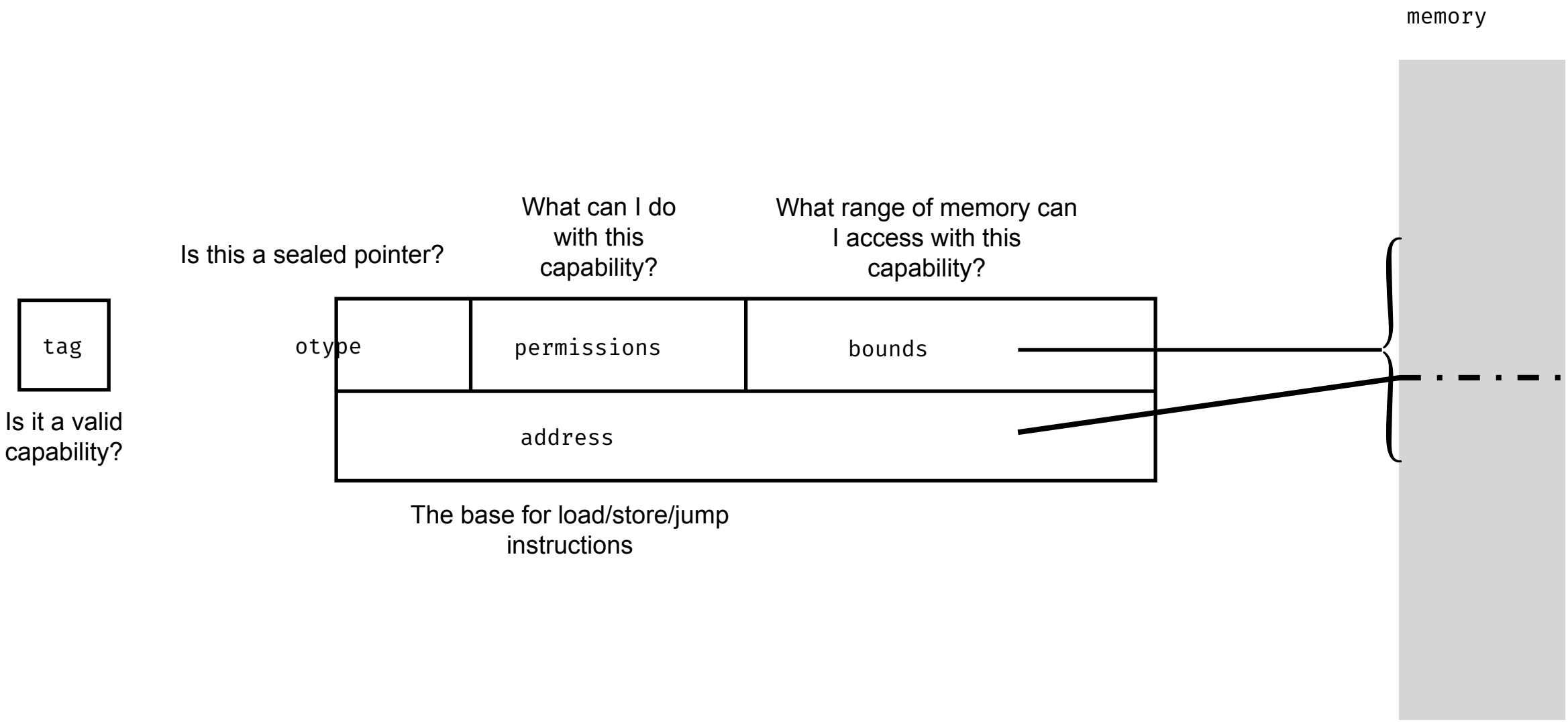
Refer to an address

Add metadata

Can be passed or embedded in objects

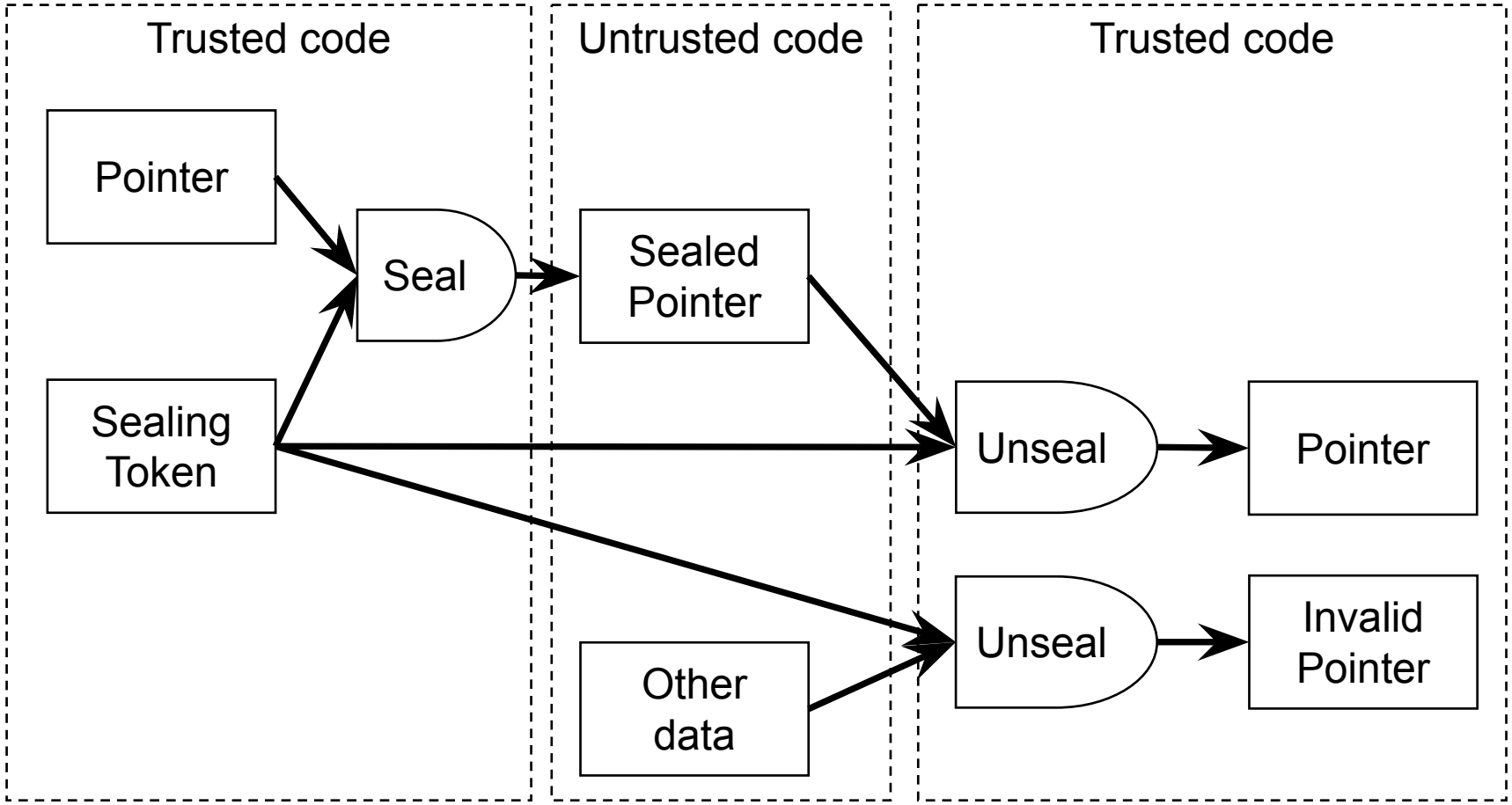
Must be explicitly dereferenced

Anatomy of a capability



Sealed capabilities

Creating type-safe opaque pointers



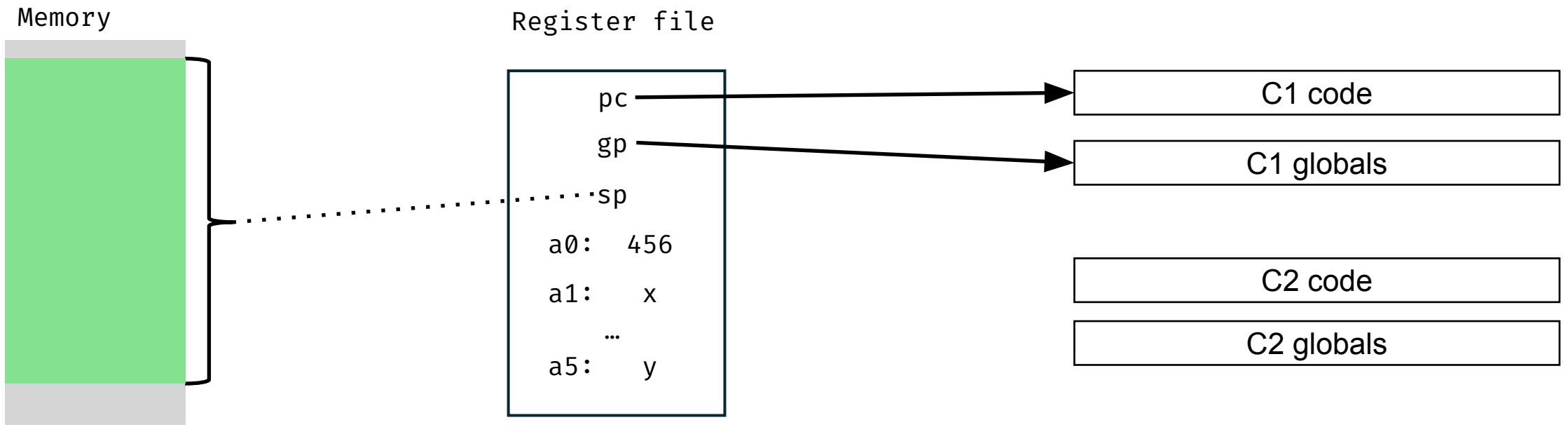
Compartmentalization

Defining security boundaries

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

Switcher

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



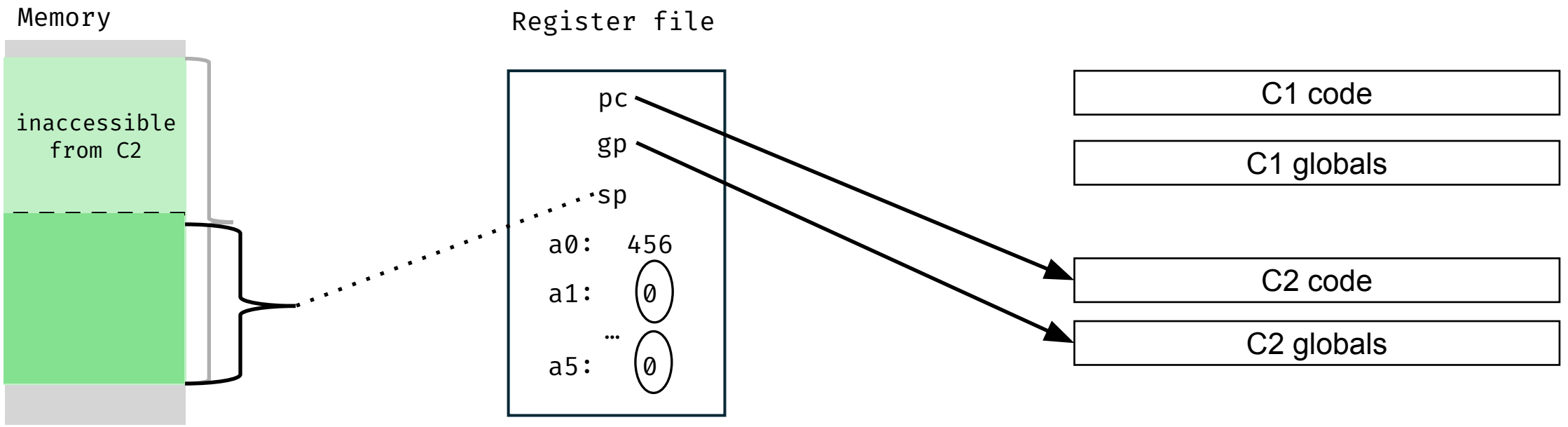
Compartmentalization

Defining security boundaries

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

Switcher

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



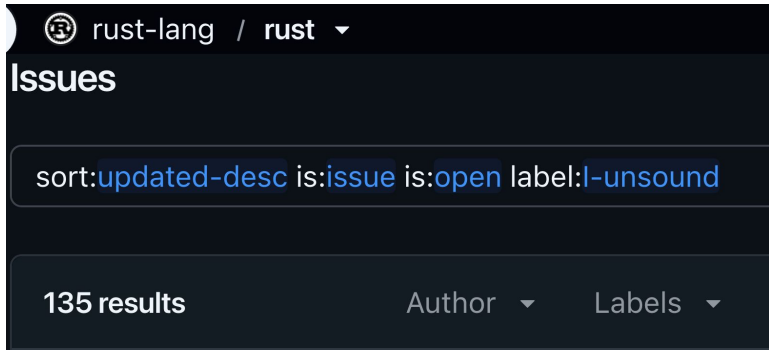
Why CHERI?

Why CHERI?

You don’t need CHERI if:

1. Your entire software stack is written in safe Rust (Rust’s libraries as well), and
2. Your Rust implementation does not have any soundness holes
3. You trust your third-party dependencies

```
$ @library/core: rg "unsafe \{" | count
1879 // Contains doc-comments too, however!
```



Why CHERI?



Addresses the lack of memory safety



…a solution to the same problems?

“Choose one”

Why CHERI?



Addresses the lack of memory safety in `safe` blocks



Addresses the lack of memory safety in *every* `unsafe` block

Choose both!

Rust on CHERIoT

Rust on CHERIoT

- In 2025 UKRI's DSbD granted SCI Semiconductor and University of Kent funding to make Rust work on CHERIoT
- The objective: a production-ready compiler by early 2028
- SCI Semiconductor's ICENI for sale in 2027
- It is all open-source!

Rust on CHERIoT

1. Adapt previous work on Morello to latest Rust releases
   - (Morello 1.72) -> 1.89 -> 1.100
2. Add the CHERIoT target and CHERI intrinsics
3. Make `core` and `alloc` compile to it
   - CHERIoT as a no-std, no host tools tier-3/2 platform
4. Test `core` and `alloc`
5. Add CHERIoT-specific frontend attributes (MMIO, compartments, …)
6. Integration tests with the rest of the CHERIoT platform (the RTOS, networking stack..)

Rust on CHERIoT

We need a compiler back-end!

```
[submodule "src/llvm-project"]
    path = src/llvm-project
    url = https://github.com/rust-lang/llvm-project.git
    branch = rustc/22.1-2026-03-22
    url = https://github.com/CHERIoT-Platform/llvm-project.git
    branch = cheriot
```

Rust on CHERIoT

We need a compiler back-end!

```
pub(crate) fn LLVMRustBuildMemCpy<'a>(
    B: &Builder<'a>,
    Dst: &'a Value,
    DstAlign: c_uint,
    Src: &'a Value,
    SrcAlign: c_uint,
    Size: &'a Value,
    PreserveTags: PreserveCheriTags,
    IsVolatile: bool,
) -> &'a Value;
```

```
pub(crate) fn LLVMRustBuildMemMove<'a>(
    B: &Builder<'a>,
    Dst: &'a Value,
    DstAlign: c_uint,
    Src: &'a Value,
    SrcAlign: c_uint,
    Size: &'a Value,
    PreserveTags: PreserveCheriTags,
    IsVolatile: bool,
) -> &'a Value;
```

Rust on CHERIoT

The new CHERIoT target

```
pub(crate) fn target() -> Target {
    Target {
        // The below `data_layout` is explicitly specified by the ilp32e ABI in LLVM. See also
        // `options.llvm_abiname`.
        data_layout: "e-m:e-p:32:32-pfe200:64:64:64:32-i64:64-n32-S128-A200-P200-G200".into(),
        llvm_target: "riscv32-unknown-cheriotrtos".into(),
        metadata: TargetMetadata {
            description: Some("CHERIoT RISC-V (RV32E ISA)".into()),
            tier: Some(3),
            host_tools: Some(false),
            std: Some(false),
        },
        pointer_width: 64,
        arch: Arch::RiscV32,
```

Rust on CHERIoT

Teaching Rust that *sometimes* `size_of<*const ()>() ≠ size_of<usize>()`

```rust
/// Get the pointer size in the default data address space.
#[inline]
pub fn pointer_size(&self) -> Size {
    self.default_address_space_pointer_spec.pointer_size
}
```

```rust
/// Get the pointer index in the default data address space.
#[inline]
pub fn address_size(&self) -> Size {
    self.default_address_space_pointer_spec.address_size
}
```

Rust on CHERIoT

Teaching Rust that *sometimes* `size_of<*const ()>() ≠ size_of<usize>()`

```rust
#[derive(Clone, Copy, Eq, PartialEq, TyEncodable, TyDecodable, Hash)]
#[derive(StableHash)]
pub enum Scalar<Prov = CtfeProvenance> {
    /// The raw bytes of a simple value.
    Int(ScalarInt),

    /// A pointer.
    ///
    /// We also store the size of the pointer, such that a `Scalar` always knows how big it is.
    /// The size is always the pointer size of the current target, but this is not information
    /// that we always have readily available.
    Ptr(Pointer<Prov>, u8),
    /// We also store the in-memory size and capacity of the pointer, such that a `Scalar` always knows how
    /// big it is. The in-memory size is always the pointer size of the current target, but this is not
    /// information that we always have readily available.
    ///
    /// On targets with integral pointers - that is, where pointers are represented simply as the memory
    /// address they refer to - in-memory size and capacity are equivalent. On other systems, such as CHERI,
    /// that is not the case, and pointers have an in-memory size (intended as their the size of their
    /// in-memory representation) which is different from their capacity (intended as the size of
    /// the memory address that they can hold in them).
    Ptr { ptr: Pointer<Prov>, in_memory_size: u8, capacity: u8 },
}
```

Rust on CHERIoT

Teaching Rust that *sometimes* `size_of<*const ()>() ≠ size_of<usize>()`

```
impl Primitive {
    pub fn size<C: HasDataLayout>(self, cx: &C) -> Size {
    /// Retrieve how many bits it takes to represent in memory this [`Primitive`].
    pub fn in_memory_size<C: HasDataLayout>(self, cx: &C) -> Size {
    /// Retrieve how many bits of data this [`Primitive`] contains.
    pub fn capacity<C: HasDataLayout>(self, cx: &C) -> Size {
```

Rust on CHERIoT

Teaching Rust that *sometimes* `size_of<*const ()>() ≠ size_of<usize>()`

```
#[inline]
fn target_usize_max(&self) -> u64 {
    self.pointer_size().unsigned_int_max().try_into().unwrap()
    self.data_layout().address_size().unsigned_int_max().try_into().unwrap()
}
```

Rust on CHERIoT

Teaching Rust that *sometimes* `size_of<*const ()>() ≠ size_of<usize>()`

core::ptr::non_null

```rust
pub const fn without_provenance(addr: NonZero<usize>) -> Self {
    let pointer: *const T = crate::ptr::without_provenance(addr.get());
    // SAFETY: we know `addr` is non-zero and all nonzero integers are valid raw pointers.
    unsafe { transmute(addr) }
    unsafe { transmute(pointer) }
}
```

core::fmt

```rust
pub struct Arguments<'a> {
    template: NonNull<u8>,
    args: NonNull<rt::Argument<'a>>,
}
```

```rust
pub const fn as_str(&self) -> Option<&'static str> {
    #[cfg(not(target_family = "cheri"))]
    let bits: usize = unsafe { mem::transmute(self.args) };

    #[cfg(target_family = "cheri")]
    let bits: usize = unsafe {
        // On CHERIoT we can force this into a `ptrtoint` by transmuting it into an u64 and
        // casting it into a usize.
        #[cfg(target_family = "cheriot")]
        {
            crate::mem::transmute::<_, u64>(self.args) as usize
        }
```

Rust on CHERIoT

Testing: an ad-hoc façade target with std support

```rust
//! This target is only meant to be used internally for running tests, which
//! require some level of standard library support.

use crate::spec::{Env, Target};

pub(crate) fn target() -> Target {
    let mut target = super::riscv32cheriot_unknown_cheriotrtos::target();

    // Use "sim" environment so to differentiate from our target proper
    // without needing to rename the OS.
    target.env = Env::Sim;
    // Testing infrastructure expects this.
    target.executables = true;
    // Avoid any confusion.
    target.entry_name = "rust_main".into();
    // Set it as singlethread for now.
    target.singlethread = true;

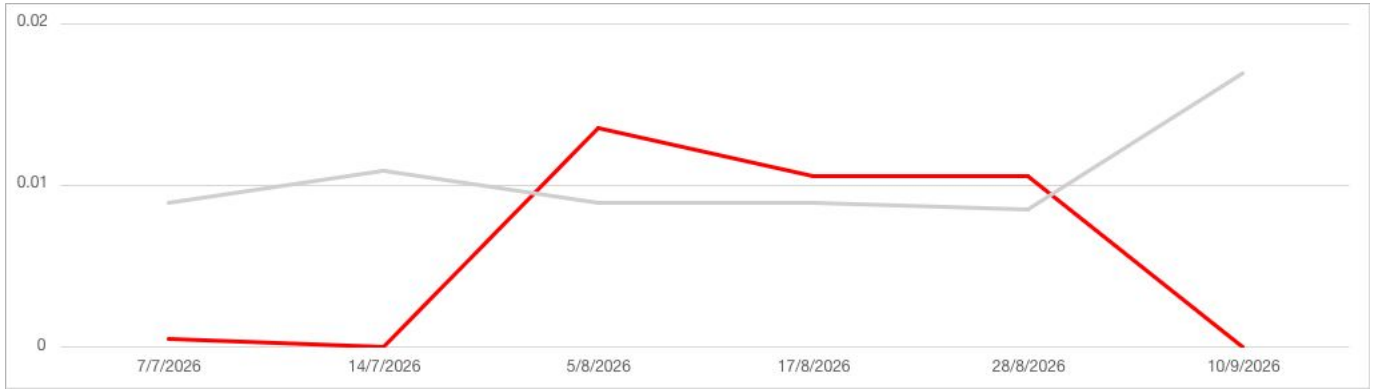
    target
}
```

Rust on CHERIoT

Testing: more than 2.5k tests passing



Total passing tests over time



Failing and ignored tests over total, over time

Rust on CHERIoT

New CHERI-specific feature:
MMIO

```rust
/// Get the UART #0 from the board (address 0x80100000).
pub unsafe fn uart0() -> Uart<'static> {
    #[allow(static_mut_refs)]
    unsafe {
        Uart {
            ptr: core::mem::transmute(0x80100000),
phantom: Default::default(),
        }
    }
}
```

Rust on CHERIoT

New CHERI-specific feature:
MMIO

```rust
/// Get the UART #0 from the board (address 0x80100000).
pub unsafe fn uart0() -> Uart<'static> {

    unsafe extern "Rust" {
        #[cheriot_mmio(name = "uart", permissions = "RW")]
        pub static mut UART0: InnerUart;
    }

    #[allow(static_mut_refs)]
    unsafe {
        Uart {
            ptr: &mut UART0,
            phantom: Default::default(),
        }
    }
}
```

Rust on CHERIoT

New CHERI-specific feature:
compartments

```rust
#![no_std]
#![feature(cheriot_attributes)]
#![cheri_compartment = "my_app"]

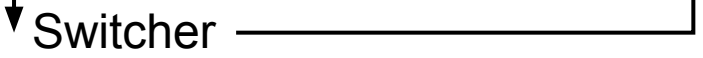
#[allow(unused)]
use cheriot::*;

#[unsafe(export_name = "_Z9call_rustv")]
pub fn call_rust() -> core::ffi::c_int {
    log::record("hello", "from rust");
    core::ffi::c_int::from(0)
}
```

```rust
#![no_std]
#![feature(cheriot_attributes)]
#![cheri_compartment = "log4rust"]

use cheriot::*;

/// Log a new entry.
pub fn record(key: &str, value: &str) {
    println!("[log] {key} -> {value}");
}
```



Rust on CHERIoT

On the horizon: future features for linting of unsafe blocks

```rust
#[repr(transparent)]
struct Socket { ... }
unsafe extern "C" {
    fn socket_send(...);
}
fn say_hello() {
    let socket = Socket::new(...);
    let buffer = "Hello!";
    let mut res = 0;
    unsafe {
        res = network_socket_send (Timeout::new(), socket,
            buffer.as_ptr(), buffer.len());
    }
    ...
 }
```

```c
/* Send data over a TCP socket. */
ssize_t network_socket_send (Timeout *timeout,
                             Socket socket,
                             void *buffer,
size_t length);
```

Rust on CHERIoT

On the horizon: future features for linting of unsafe blocks

```
#[deny(cheriot_unsafe)]

#[repr(transparent)]
struct Socket { ... }
unsafe extern "C" {
    fn socket_send(...);
}
fn say_hello() {
    let socket = Socket::new(...);
    let buffer = "Hello!";
    let mut res = 0;
    unsafe {
        res = network_socket_send (Timeout::new(), socket,
            buffer.as_ptr(), buffer.len());
    }
    ...
 }
```

```
/* Send data over a TCP socket. */
ssize_t network_socket_send (Timeout *timeout,
                             Socket socket,
                             void *buffer,
size_t length);
```



“error: raw const pointer argument passed without explicitly stripping write and capture permissions”

Rust on CHERIoT

On the horizon: future features for linting of unsafe blocks

```
// rustc -Ccheriot-strip-permissions

#[repr(transparent)]
struct Socket { ... }
unsafe extern "C" {
    fn socket_send(...);
}
fn say_hello() {
    let socket = Socket::new(...);
    let buffer = "Hello!";
    let mut res = 0;
    unsafe {
       // Automatically inserted by the compiler for you
          let ro_arg = cheri::permissions_and(buffer.as_ptr(),CHERIPermissions::Write);
       res = network_socket_send (Timeout::new(), socket,
            ro_arg, buffer.len());
    }
    ...
 }
```

```
/* Send data over a TCP socket. */
ssize_t network_socket_send (Timeout *timeout,
                             Socket socket,
                             void *buffer,
size_t length);
```

Rust on CHERIoT

Conclusions

- We want to donate our code to upstream Rust, and offer to maintain the new target
  - If you are from the compiler team, let's chat
- If you want to contribute, we will be happy to review your PRs
  - If you want to add a new CHERI target reach out to us to coordinate
  - We don't accept LLM-generated code or issues, though!

`github.com/CHERIoT-Platform/cheri-rust`

