Digital Security
by Design

A Rusty CHERI

The path to hardware capabilities in Rust

Lewis Revill
<lewis.revill@embecosm.com>

Introduction

Lewis Revill

- Compiler engineer working for Embecosm
- LLVM backends for constrained architectures

Embecosm

- Software services company
- Operate on the boundary between hardware and software
- Solve difficult and interesting problems, like compilers

EMBECOSM®

What is CHERI?

- **C**apability **H**ardware **E**nhanced **R**isc **I**nstructions
- Instruction set extension
- Encodes access constraints on addresses* with capabilities
- Capability operations replace pointer operations
  - Purecap mode: all pointers are capabilities
  - Hybrid: some pointers are capabilities
- Spatial, referential & temporal safety enforced at runtime

Integrating CHERI & Rust

- Project led by our customer CyberHive
  - Funded by Digital Security by Design

- Goal: produce a Rust compiler that can target CHERI-based architectures
  - Long term goal: production-ready code for security purposes
  - Initially targeting ARM's Morello platform

Motivation

- Another layer of protection
- Constraints identified at compile-time, enforced at runtime
- Unsafe Rust code can have safety constraints
- Other side benefits
  - Hardware bounds checking
  - Replace Rust arrays/’fat pointers’ with capabilities

Motivating Example

```
use std::io::stdin;

fn main() {
    let arr = [1, 2, 3, 4];
    let mut ptr = arr.as_ptr();

    let mut input = String::with_capacity(1);
    stdin().read_line(&mut input).expect("Error reading input");
    let idx: usize = input.trim().parse()
                                 .expect("Error parsing number");

    unsafe {
        ptr = ptr.add(idx);
        print!("{}\n", *ptr);
    }
}
```

How to Integrate CHERI & Rust

- Account for pointer (capability) types where type size != addressable range
- Different address space for capabilities in datalayout
- Different pointer types for different address spaces in datalayout
- Provenance and bounds propagated with capabilities
- Optional stuff
  - Replace bounds checking with capabilities
  - Replace arrays/'fat pointers' with capabilities

Progress

- Datalayout changes completed
  - Can specify capability sizes and address spaces for purecap or hybrid mode
- Mandatory address space parameters in APIs
- Differentiate `ty_size`/`total_size` and `val_size` in APIs
  - EG `LayoutS`, `AllocRange`, `Primitive`, ...
  - Most fundamental change by far
- Create invalid (non-dereferencable) capabilities from `mem::transmute`
- Current state: fixing assertion failures when building core/compiler_builtins

Future Changes

- Modifications to core/compiler_builtins/std...
- How to specify capability types in hybrid mode?
  - Rust annotations don't seem the right tool
  - Library solution?
- Comprehensively evaluate uses of `ty_size` - should they be using `val_size`?
- Testing, polish

Similar Work

- Nicholas Sim (Univ. of Cambridge) Master's thesis 2020
  - Series of patches which set both pointer & usize width to capability width
- University of Kent
  - Implementation compiles and runs for Morello purecap
  - Based on Rust 1.56.0
- Others?

Digital Security
by Design

Thank you

github.com/lewis-revill/rust-cheri