Zero Cost Capabilities

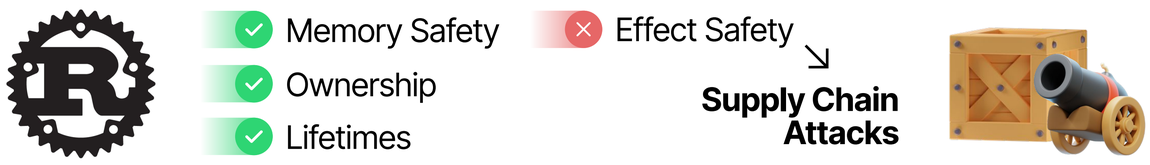
Retrofitting Effect Safety in Rust

George Berdovskiy

UC DAVIS UNIVERSITY OF CALIFORNIA

DAVIS PL

Background and Motivation



- Investigate whether it's possible to prevent supply chain attacks by retroactively enforcing side effect safety
  - Accomplished using **capabilities** – unforgeable tokens representing file system resources and permitted actions
- Three design objectives – **static** enforcement, **zero-cost** abstractions, and **unobtrusiveness**
- We introduce *Coenobita,* a Rust library that prevents undesirable **file system side effects** using capabilities

Coenobita

Design

- Capabilities wrap paths
  - **`Cap`** wraps **`Path`** and **`CapBuf`** wraps **`PathBuf`**
- Implementation prevents crates from **modifying capabilities,** which are provided by the user as function arguments, and a script prevents **capability creation**
- Three generic type parameters describe capability permissions
  - Represented by tuples of permission types → **enforced by type system**

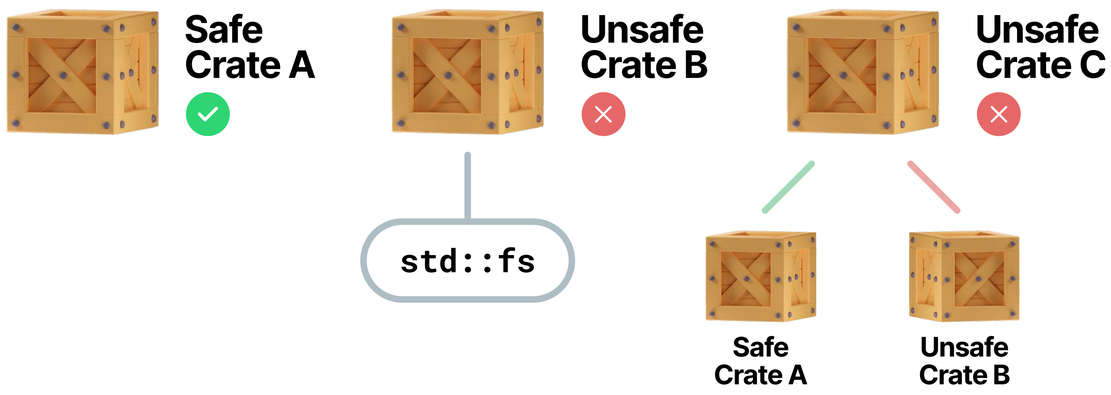
```
impl<P1, P2, P4, P5, P6, P7, P8> traits::Read
  for (P1, P2, Read, P4, P5, P6, P7, P8) {}
```

- Thus, crates can only access **specific, immutable** locations with **specific, immutable permissions** that are checked by the compiler
  - Crates should only compile when **all capability safety rules are followed!**

```
Cap<A, B, C> {
    path: Path
}
```

```
Path
```

- We introduce the intuition of a **safe crate,** which...
  - Doesn't create capabilities
  - Doesn't use **`std::fs`**, functions accessing the file system in **`std::path`**, or **unsafe crates**
- These rules can be enforced with a script
- Thus, crates that compile (via the script) should be **safe crates**



Example

Trusted Program

```
use suspicious_crate::foo;
foo(cap!("some/path.txt" with (View, Read)));
```

Suspicious Crate

```
pub fn foo<A1: traits::Read, A2, A3, C>(cap: C)
where
    C: AsRef<Cap<A1, A2, A3>>
{
    let _result = fs::read(cap);
}
```

Coenobita

```
pub fn read<A1: traits::Read, A2, A3, C>(cap: C) -> Result<Vec<u8>>
where
    C: AsRef<Cap<A1, A2, A3>>
{
    fs::read(cap.as_ref().to_path())
}
```

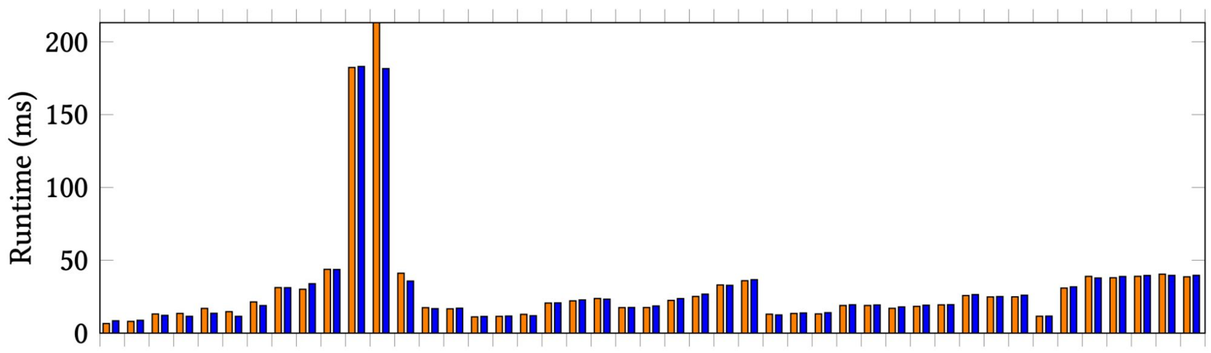
The trusted program calls **`foo`**, passing a capability with **view** and **read** permissions as an argument. This function is defined in the "suspicious crate" and only accepts capabilities with *at least* read permissions.

Assume we've verified that the suspicious crate is safe. Therefore, it must use Coenobita to access the file system. Because **`foo`** cannot create capabilities or modify the argument **`cap`**, it can only pass **`cap`** to *other* functions that accept capabilities with read permissions.

Therefore, **`foo`** can **only read file system resources**. Attempts to circumvent this rule will result in compiler errors.

Evaluation

We ported the crates **`walkdir`** and **`remove_dir_all`** to Coenobita for evaluation. We focus our efforts on **`walkdir`** because it's more complex.



Running benchmarks on **`walkdir`**'s test suite indicates **minimal difference in performance** between the original and ported crates.

File	Original Lines	Lines Added	Lines Modified
`lib.rs`	**1186**	**3**	**65**
`dent.rs`	**352**	**2**	**30**
`error.rs`	**262**	**25**	**20**

Many line modifications were required, but most were achieved using simple search-and-replace commands and involved the addition of generic type parameters or trait bounds → Coenobita is **practical.**

Related Work

- Languages (and language extensions) enforcing capability safety...
  - Safe Haskell - David Terei, Simon Marlow, Simon Peyton Jones, David Mazières
  - Shill - Scott Moore, Christos Dimoulas, Dan King, Stephen Chong
  - Scala (language extension) - Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, Ondřej Lhoták
  - E - Mark Samuel Miller

shill

Scala