Zero-Cost Capabilities: Retrofitting Effect Safety in Rust

GEORGE BERDOVSKIY, University of California, Davis, USA

1 BACKGROUND AND MOTIVATION

Over the last several years, the Rust programming language has gathered a following among software developers for its robust memory safety features. These features are attractive because memory safety vulnerabilities are widespread, *especially* in C and C++ codebases. Four years ago, the Chromium security team reported that memory safety problems were responsible for around 70% of the browser's severe security bugs [14]. By enforcing ownership and lifetime rules, Rust eliminates a broad category of security vulnerabilities.

Nevertheless, the Rust language remains susceptible to potentially harmful *side effects* in untrusted code. Some languages mitigate this vulnerability using type systems, encapsulation, and abstraction. Haskell prevents *most* harmful side effects by confining them within monads like `IO`, and Safe Haskell eliminates them entirely by closing loopholes involving unsafe functions [15]. Therefore, Safe Haskell is impervious to *supply chain attacks*, which occur when programmers use malicious packages or libraries in their codebases. However, because Haskell is purely functional, it's not the ideal choice in contexts where performance is critical, including most low-level systems software.

Rust is more suited for such purposes, but since it doesn't guard against harmful side effects, it's vulnerable to supply chain attacks. We wish to investigate whether it's possible to prevent them by *retroactively* enforcing side effect safety. Our goal is to accomplish this using *capabilities*, unforgeable tokens that represent resources and the actions their holders can take. Capabilities have been studied numerous times in the context of programming languages; prominent examples include the E language for robust distributed systems [11], Shill for secure script execution [12], and an extension to Scala that limits polymorphic effects [13]. We have three specific design objectives:

(1) Capabilities should be enforced **statically** as much as possible to minimize runtime overhead, excluding aspects of object capabilities that require dynamic enforcement, like revocation.

(2) Capabilities should be **zero-cost**, consuming no more computational resources than their standard library equivalents.

(3) Capabilities should be as **unobtrusive** as possible.

To accomplish these goals, we introduce *Coenobita*, a Rust library that prevents undesirable side effects using capabilities. To make the problem tractable, we currently focus only on file system side effects, but plan to consider others in future work. Coenobita replaces standard library file system data structures, traits, and functions with capability-safe versions. For example, instead of using `Path` or `PathBuf` directly, we wrap it with `Capability<A, B, C>`, where the generic type parameters represent permissions associated with the underlying file system resource. We take advantage of Rust's type and trait systems to prevent crates with rule violations from compiling. Coenobita is developed under an open source license and will be made available through GitHub.

2 DESIGN

Coenobita leverages Rust's powerful type and trait systems to enforce capability safety. Its core data structure is `Capability<A, B, C>`, which wraps `PathBuf` and associates it with three generic type

Author's address: George Berdovskiy, gaberdovskiy@ucdavis.edu, University of California, Davis, Davis, California, USA.

```rust
pub fn read<P1: traits::Read, P2, P3>(
    cap: &Capability<P1, P2, P3>
) -> io::Result<Vec<u8>> {
    fs::read(cap.get_path())
}
```

(a)

```rust
// Creating and using a capability
let result = read(
   &cap!("some/path.txt" with (Read, Write, Append))
);
```

(b)

Fig. 1. (a) Function taking a capability and (b) capability creation using `cap!`

parameters. These parameters represent intransitive resource permissions, intransitive descendant permissions, and transitive descendant permissions respectively. The first type describes allowed actions on the resource itself. The second type describes actions permitted on the resource's immediate descendants. The third describes allowed actions on any descendant of a resource. Each permission set communicates whether holders of this capability are permitted to create, view, read, write, append, copy, move, or delete the referenced resource and its children.

2.1 Zero-Cost

Coenobita's abstractions aim to be zero-cost. In other words, they should have the same performance as their standard library counterparts. Coenobita achieves this by making it possible to **statically enforce capability safety rules during compilation,** leaving runtime unaffected. One reason for that is the use of empty structures and wrappers that require the same amount of memory as their wrapped value. Another reason is that programs with invalid capability interactions fail to compile until the violations are resolved because of unsatisfied trait bounds, missing type arguments, mismatched types, and other errors.

Due to clever inlining and optimization on behalf of the Rust compiler, Coenobita's functions and data structures typically simplify to their wrapped standard library equivalents. In other words, a compiled program using `std::fs` will have nearly identical performance using Coenobita instead. For instance, the compiler will simplify both `std::fs::read` and `coenobita::fs::read` to `std::fs::read`'s internal logic, which can be determined by inspecting the call graph.

2.2 Safety

Programmers create capabilities using the `cap!` macro, which takes a resource path and chosen permissions, as in Figure 1. This macro expands to the initialization of a `Capability` struct, which would be tedious to write manually. For untrusted crates to be free from harmful side effects, we must prevent them from creating capabilities.

To solve this problem, we plan to implement a simple program that parses untrusted crates and disallows compilation if they use `cap!` or manually initialize capabilities. The program will also prevent crates that import `std::fs` or `std::path` from being compiled since they are unsafe.

3 EVALUATION

To evaluate Coenobita's practicality and effectiveness, we conducted two case studies porting popular Rust crates `walkdir` and `remove_dir_all` to Coenobita. Our evaluation seeks to address three questions: **(Q1)** Is Coenobita feasible as a drop-in replacement for `std::fs` in Rust crates, **(Q2)** what is the programming overhead, and **(Q3)** what is the performance overhead?

Our case studies followed this general process: we replaced `PathBuf` and `Path` with `Capability<A, B, C>` or a reference to `Capability`. Since functions in Coenobita and `std::fs` have otherwise

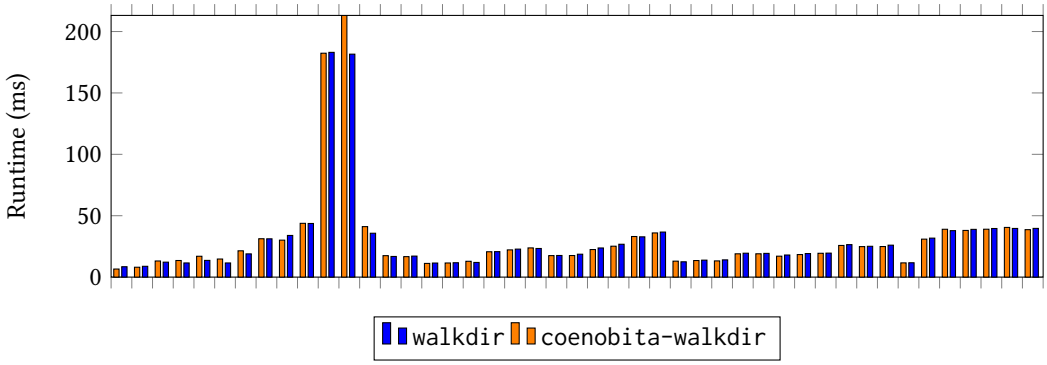

Fig. 3. Comparison of corresponding function runtime between `walkdir` and `coenobita-walkdir`

identical signatures, rewriting calls to functions in `std::fs` was unnecessary in most cases. Thus, we believe Coenobita is **feasible as a drop-in replacement for `fs` in practical Rust crates (Q1)**.

3.1 Programming Overhead

Implementing capability safety with Coenobita requires few additional lines. In both crates, line additions were usually the result of implementations requiring specific permission trait bounds. Many lines needed *modification* as seen in Figure 2, but since most edits were made using simple search and replace commands in Vim, we believe **Coenobita introduces only minor programming overhead (Q2)**.

3.2 Performance Overhead

We evaluated performance overhead for the modified `walkdir` and `remove_dir_all` crates by running benchmarks on their provided test cases. Our results showed that Coenobita **doesn't significantly increase runtime performance (Q3).** Figure 3 compares runtimes for all 46 tests in `walkdir`'s test suite. All benchmarks ran on a MacBook Pro 2021 using macOS Monterey 12.5.1 equipped with an Apple M1 Pro processor and 16 GB of memory.

File	Lines Changed
lib.rs	138
dent.rs	46
error.rs	58

Fig. 2. Number of lines changed in each file of `walkdir`, including additions and formatting

4 RELATED WORK

The concept of capabilities first arose in discussions involving access control, including works by Henry M. Levy on secure computer systems [9] and Butler W. Lampson on operating systems and confinement [8] [7]. Early ideas on the subject were unified into the *object-capability model* by Mark S. Miller in his dissertation on the robust composition of distributed systems and the E language [11]. Few projects have explored the potential of capabilities in Rust aside from the `cap-std` crate, which only provides coarse-grained capabilities at the directory level, lacking fine-grained control at the file level [1].

Aside from the E language, capabilities as a tool for enforcing safety in language design have also been used in Shill, a modified version of Racket that adapts capabilities and introduces the concept of *contracts* [12]. The Safe Haskell language extension is broader in focus but has significant conceptual overlap and could easily implement capabilities [15]. Most investigations into Rust safety involve, for example, sandboxing and isolating unsafe code [2–6, 10] or proving its memory safety [3, 4] but do not emphasize language design and do not consider more general side effects.

REFERENCES

[1] Bytecode Alliance. cap-std. https://github.com/bytecodealliance/cap-std, 2023. Last commit on November 1, 2023.

[2] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, August 2023. USENIX Association.

[3] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[5] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 132–148, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS '17, page 51–57, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, oct 1973.

[8] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, jan 1974.

[9] H.M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[10] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, USA, 2006. AAI3245526.

[12] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 183–199, Broomfield, CO, October 2014. USENIX Association.

[13] Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. Scoped capabilities for polymorphic effects, 2022.

[14] Chromium Project. Memory safety — chromium.org. https://www.chromium.org/Home/chromium-security/memory-safety/. [Accessed 08-11-2023].

[15] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. *ACM SIGPLAN Notices*, 47, 09 2012.