

Coenobita: Static Capability Types for Side Effect Safety in Rust

George Berdovskiy and Caleb Stanford

Abstract

Rust is increasingly used to develop secure low-level software. Unfortunately, arbitrary Rust code (even in the *safe* subset) can have destructive side effects, such as interaction with the filesystem, which makes it equally as dangerous as code written in C. To protect against these side effects, we propose Coenobita, a system for static capability types in Rust. Our design generalizes a standard system for static information flow with *author* and *provider* annotations, which inhabit the same lattice as the information flow labels; the type annotation (a, c, i) indicates sets of crates a , c , and i which are allowed to author, explicitly contribute to, or implicitly influence a value; and the provider annotation p on a function indicates the set of crates that are allowed to call it. We show that this design is sufficient to implement capability safety in a purely static way. It leads to different variants of capabilities depending on what integrity properties are desired. We report on a preliminary formalization of this design in an extension to the simply-typed lambda calculus, and an implementation of a type checker as an extension to the Rust compiler. Our evaluation on a selection of popular Rust crates shows that capabilities are zero-cost (statistically insignificant overhead at runtime) and unobtrusive (for example, requiring only minor changes such as replacing `PathBuf` with `Capability<PathBuf>`), with an average 4% overhead in compile times.

Keywords: capabilities, type systems, Rust

Reference format:

George Berdovskiy and Caleb Stanford. 2026. Coenobita: Static Capability Types for Side Effect Safety in Rust. Presented at *International Workshop on Aliasing, Capabilities, and Ownership (IWACO 2026)*, October 9, Oakland, California, USA. 15 pages.

1 Introduction

Rust [21] is increasingly used to develop secure low-level software. As a *systems* language, its relevant features include type safety and memory safety by default, as well as the ability to access low-level operating system resources (e.g., file handles and syscalls). However, this access is a double-edged sword: it can silently introduce destructive *side effects*, even into otherwise *safe* code, including in the standard library. For example, it is possible on Linux to access a file path called `proc/self/mem`, which is mapped to a process’s own memory, and this can be exploited to bypass memory safety in safe Rust [18, 42]. Similarly, supply chain attacks such as the `rustdecimal` attack [40] and recent `arrayref`

attack [12, 25] rely on silently injecting side effects (in this case, compromising a bitcoin wallet) into otherwise pure code [47].

One approach to prevent supply chain attacks would be to insist that all code is marked *pure* by default, and cannot invoke side effects. For example, this approach was pioneered by the Safe Haskell project [41], and this was even supported in some versions of Rust prior to 1.0 [20]. However, this approach would require drastic changes to the Rust library ecosystem, `crates.io`, since systems side effects have permeated many existing crates (Rust libraries) and their dependents.

In this paper, we instead focus on whether *capabilities* can be used to harden Rust against silent destructive side effects, without drastic changes to existing crates. The object-capability model dates back to Van Horn in 1966 [11] and others [16, 24, 37, 46] and provides a useful conceptual paradigm for designing secure computer systems. The main idea is that side effects may occur freely so long as they are invoked through a *capability*, or unforgeable proof-of-access token, to the system resource in question. These tokens must be provided by a trusted source – typically, either directly or indirectly from the “root” user – and such that this trusted access cannot be spoofed. However, one thing that has perhaps limited the adoption of existing capability-based type systems [2, 5–7, 15, 17, 27, 29, 30, 43] is that many impose runtime overhead, for example in dynamic languages like JavaScript [29] (see also [37]). Moreover, the usefulness of capabilities is debatable in lower-level languages like C/C++, since code can freely access and modify arbitrary pointers, and this threatens the “unforgeability” of the capability access tokens. In Rust, types are immutable when passed by reference, and we don’t have to worry about unauthorized access via pointers.

Capability types for Rust. We propose Coenobita, a system for static capability types in Rust. Our hypothesis is that *most Rust crates can be adapted to be capability-safe – so that side effects cannot occur except behind capability-safe abstractions*. For example, the capability type

`Capability<PathBuf>`

can be used as a drop-in replacement for Rust’s `PathBuf` file path type; the capability wrapper indicates that the object can only be constructed by the root user. We identify four design goals for our type system:

1. *Capabilities should be static and zero-cost.* That is, the running time should be equivalent to capability-free version wherever possible.
2. *Capabilities should be unobtrusive.* That is, they require minimal programming overhead as measured by changes to the source code, and they should act as a drop-in replacement for their capability-free counterparts.
3. *Capabilities should be local,* meaning that functions which do *not* invoke side effects require no changes.
4. *Capability tokens should be unforgeable* – constructed only by authorized users.

To support the above goals, we propose a type system which extends a standard information flow system for integrity (where labels are thought of as crates, and integrity is the set of crates that influence a value) with *author* and *provider* annotations, which are also sets of crates.

- First, for Rust values, the *author annotation* a provides a set of crates which may be allowed to construct the value; this is combined with the standard information flow labels of a set of explicit contributors c and influencers i to form an *integrity triple* $\kappa = (a, c, i)$, where $a \subseteq c \subseteq i$. The reason for including the additional information flow labels c, i will become more clear in Section 2: although Rust prevents capability tokens from being modified directly, the object-capability model alone does not prevent third-party crates from *indirectly* influencing the value of a capability token (by providing an argument or a callback to the *root* user), and this can leave capabilities vulnerable to confused deputy attacks.
- Second, for functions, the *provider annotation* p provides the set of crates which may be allowed to call the function. We show that provider set p is enough to enforce a basic capability-safe system: as we can enforce that the constructor for `Capability<PathBuf>` is only called by the root user.

We have completed a preliminary formalization of our type system as an extension to the simply typed lambda calculus. We omit the details of the formalization from the main paper, but include them in the Appendix. This includes an operational semantics involving *membranes* to track what crate values come from, and polymorphism in the form of parametric types, which can model Rust generics.

Turning to the evaluation, we implemented a checker for capability types as an extension to the Rust compiler, and adapted a selection of 10 Rust crates which interact with the file system to be capability-safe. For the crates we tested, we found that capabilities can be added as a drop-in replacement for standard library types like `PathBuf`, with zero cost to performance as the capability type wrappers are eliminated at compile time. Additionally, on this selection of crates, capabilities require few changes to the code, have

```

1 // Defined by Coenobita
2 pub struct Capability<T> {
3     #[cnbt::providers({R})]
4     payload: T
5 };
6
7 // Appears in the root crate
8 fn main() {
9     let p = PathBuf::from("file.txt");
10    let c = Capability::new(p);
11
12    // Crate 'foo' can access "file.txt"
13    foo::bar(c);
14 }
```

Figure 1. Basic capability abstraction and example for file paths. The annotation on line 3 ensures that only the root origin R can supply the capability payload.

statistically indistinguishable runtime from the original code, and have a compile-time overhead of 3.6% on average.

Among prior work on capabilities, `cap-std` [1] is of note as a prior system for capabilities in Rust; in Section 6, we compare Coenobita qualitatively with `cap-std` and two other selected systems: Shill [30], a capability system for writing secure shell scripts; and Joe-E [27], a capability-safe subset of Java. Prior work has explored the interaction between capabilities and information flow in different ways [4, 13, 14]. Wyvern [26] shares similar goals to our system, and is statically enforced, but its capabilities are enforced at a module level in a somewhat different way, by providing wrappers around effectful APIs to individual modules (similar to our higher-order capabilities), and it does not incorporate information flow. We compare more broadly with related work in Section 7.

In summary, our contributions are as follows:

- A new static type system which combines information flow with *author* annotations (*who* created a value) and *provider* annotations (*where* it came from) (Section 2);
- A preliminary formalization of our type system as an extension to the simply typed lambda calculus, with details in the Appendix (Section 3);
- An implementation of our capabilities as Coenobita, an extension to the Rust compiler (Section 4); and
- An evaluation of Coenobita on real-world Rust crates, which shows that capabilities are unobtrusive (requiring minimal code changes) and zero-cost on the crates that we tested. (Section 5).

2 A Tour of Coenobita

2.1 The Capability Construct

Suppose we are developing an application using the fictional `foo` crate, which must read a configuration file on our machine. If we do not trust this crate, we can protect our system resources – such as files and directories – by requiring

```

1 fn main() {
2     // Get the configuration file path
3     let p: PathBuf = foo::config::path();
4
5     // Wrap the path inside of a capability
6     let c: Capability<PathBuf> = Capability::new(p);
7
8     // Load the configuration
9     foo::config::load(c);
10 }

```

Figure 2. This code appears in the root crate. If we use the capability defined in Figure 1, the foo crate can manipulate the root crate into granting it access to arbitrary files.

```

1 pub struct Capability<T> {
2     #[cnbt::providers({R})]
3     #[cnbt::integrity({R},{R},{R})]
4     payload: T
5 };

```

Figure 3. Capability with information flow. The annotation on line 3 uses integrity to specify that only the root R is allowed to create, contribute to, or influence the payload.

```

error: expected {root} {root} {root} | {root} but found
{foo} {foo} {foo,root}
--> src/main.rs:8:50
|
8 |     let c: Capability<PathBuf> = Capability::new(p);
|

```

Figure 4. If we use the capability design from Figure 3, Coenobita will emit this error for the program in Figure 2 because $\{foo\} \not\subseteq \{root\}$ and $\{foo, root\} \not\subseteq \{root\}$.

utilities like `fs::read` and `fs::write` to accept capabilities instead of paths. Our system can be used for this purpose.

In our system, a capability is a data structure that can only be initialized by certain privileged crates. We present an example of this data structure in Figure 1. The `providers` label specifies which crates are allowed to "provide" values for this field. Crates absent from this set are considered unauthorized and cannot construct capabilities. In this case, the root crate (denoted by R) is the only authorized crate. For example, this construct ensures that the fictional crate `foo`, which needs to read a configuration file on startup, cannot interact with that resource unless the root user has supplied it with a capability `Capability<PathBuf>`. With this design, crates are forbidden from interacting with system resources unless they have *explicit* access via the relevant type, such as `Capability<PathBuf>` for filesystem paths.

2.2 Capability Integrity

However, this design is not adequate for all use cases. In the example in Figure 2, the user must explicitly supply `foo` with a capability for the configuration file, which wraps a path provided by `foo` itself. Since the user does not know the correct path, they call `foo::config::path` to obtain the path and trust that it references the intended resource. Should `foo` be compromised by malicious actors, this function may instead reference a sensitive resource, such as `~/cargo/credentials.toml`, which can be used to publish crates under your name. 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, demonstrating that static capability safety may fail to deter supply chain attacks on its own. In other words, the user is still vulnerable to a *confused deputy attack* on the root crate.

Static information flow can solve this problem. For every value Coenobita encounters, it computes an *integrity triple* that contains three sets representing its authors, contributors, and influencers. The *author set* contains crates that may have constructed the value. The *contributor set* enumerates crates that may have contributed data to the value. Finally, the *influencer set* contains crates that may have indirectly impacted the value via control flow. For a value to be trustworthy, it must have high integrity, meaning each set in its integrity triple must contain only the root crate. The new annotation in Figure 3 communicates this requirement. Our changes prevent developers from constructing a capability using an arbitrary string by detecting the violation during type checking and aborting compilation. In other words, the code in Figure 2 will not type check. Instead, Coenobita emits the error in Figure 4.

Capabilities with integrity are suited for applications in domains where security is critical and using untrusted data from dependencies is unacceptable. However, the programmer can still accept a path from `foo` by choosing a weaker point in the design space (described in Section 2.3) such as the basic capability in Figure 1. Alternatively, the programmer can explicitly acknowledge the risk of using untrusted data by *endorsing* the data using our `trust` macro, which tells Coenobita to elevate its integrity. Endorsement is similar to declassification, which is used in information flow control systems to intentionally bypass secrecy constraints.

This raises a concern – if any crate can invoke `trust`, a compromised dependency can use it to launder the provenance of a value it controls, defeating the purpose of integrity tracking. We avoid this scenario by rejecting any attempt to endorse data from crates that are not the root crate. In other words, we assert that only the root crate has authority to endorse data, and base our safety guarantees on that authority being exercised carefully. To help audit endorsement, we recommend that calls to `trust` include a brief justification, analogous to the convention of writing `SAFETY` comments alongside `unsafe` blocks in Rust. Figure 5 demonstrates our vision of safe endorsement.

```

1 fn main() {
2   for entry in glob::glob("./data/*.csv").unwrap() {
3     let path: PathBuf = entry.unwrap();
4
5     // Inspect the value before accepting it
6     if path.extension() != Some(OsStr::new("csv")) {
7       panic!("Path '{path}' is not a CSV file");
8     }
9
10    // SAFETY: We already checked the extension above
11    let trusted: Capability<PathBuf> = trust!(path);
12
13    fs::read(trusted)?;
14  }
15
16  Ok(())
17 }

```

Figure 5. The glob crate returns paths with low integrity, since it is not the root origin. The root crate inspects the path before endorsing it with `trust` and provides a comment explaining why the elevation is safe. Only the root crate may call `trust`. The same call from any other crate is rejected during type checking.

```

1 // This code appears in the 'foo' crate
2 pub fn suspicious() → PathBuf {
3   trust!(PathBuf::from("Desktop"))
4 }

```

error: 'trust' may only be invoked by the root crate
--> src/lib.rs:23:5

```

23 |   trust!(PathBuf::from("Desktop"))
    |   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

Figure 6. Only the root crate may call `trust`. The same call from `foo` is rejected during type checking, as shown above.

In Figure 6, we show what happens when `trust` is used by an unauthorized crate. The type checker emits an error, preventing packages from erasing provenance.

2.3 Design Space

Developers can combine provider and integrity annotations to obtain different levels of protection, as shown in Figure 7. Each node represents the provider set and integrity triple associated with a capability payload. The bottom-left corner corresponds to an unprotected value, since neither coordinate has a concrete upper bound. The top-right corner corresponds to the strongest policy, where only the root origin R may provide, create, contribute to, or influence the payload. Not every use case requires this strongest policy – the next subsection shows how trusted code can deliberately operate at weaker points in this lattice.

2.4 Higher-Order Capabilities

Coenobita prevents unauthorized packages from creating and modifying capabilities, which is safe but sometimes impractical because crates may legitimately need to construct capabilities themselves. For example, the `glob` crate appends `..` to arbitrary paths using the `join` method as it recursively tries to match a provided pattern. Although `glob` is harmless, permitting such behavior is generally unsafe, as malicious dependencies can use the same approach to access sensitive files. Suppose `parser` is a package that parses CSV files. If the crate becomes compromised and we provide it with capabilities for `..` and `fizz/buzz`, it may combine them to access `fizz` without our knowledge.

To resolve this tension, Coenobita provides a way for authorized crates to provide safe wrappers around utilities when necessary. We illustrate this approach in Figure 8, where we pass the `parser` crate a lambda that joins two paths together. Because the user (root) joins path capabilities c_1 and c_2 on behalf of the external crate, Coenobita does not emit any errors. We relied on this pattern to retrofit crates for our evaluation (see Section 5).

Providing crates with a capability for joining paths is not the same as letting them call `join` directly. When a crate depends on `join`, it must make this dependency explicit by requesting a join handle. If we allowed crates to call `join` directly, this dependency would be invisible to the user, providing attackers with a vector for path escalation. But even if a wrapper like `join` is provided, foreign crates may still be unable to create new paths since the payload may have authors, contributors, and influencers apart from the root origin. We overcome this obstacle by allowing users to explicitly *endorse* data as described in Section 2.2.

2.5 Beyond Path Capabilities

Finally, although we have discussed capabilities only in the context of the file system, the same syntax can be used to protect a wide range of system resources. For example, suppose the parsing library we discussed previously contains the snippet in Figure 9. While this particular command is benign, it would be trivial to replace it with a harmful one. We can handle this risk the same way we resolve the risk of arbitrary paths – by wrapping the command in a capability, as shown in Figure 10.

3 Formalization

We have formalized Coenobita’s type system in the simply typed lambda calculus, with details in the Appendix. The system tracks two kinds of information about values – provenance, through *integrity triples*, and authority, through *provider sets*. Each secure type T^κ annotates a base type T with an integrity triple $\kappa = (a, c, i)$, where a , c , and i record the origins that may have authored, explicitly contributed to, or implicitly influenced the value. Provider sets restrict

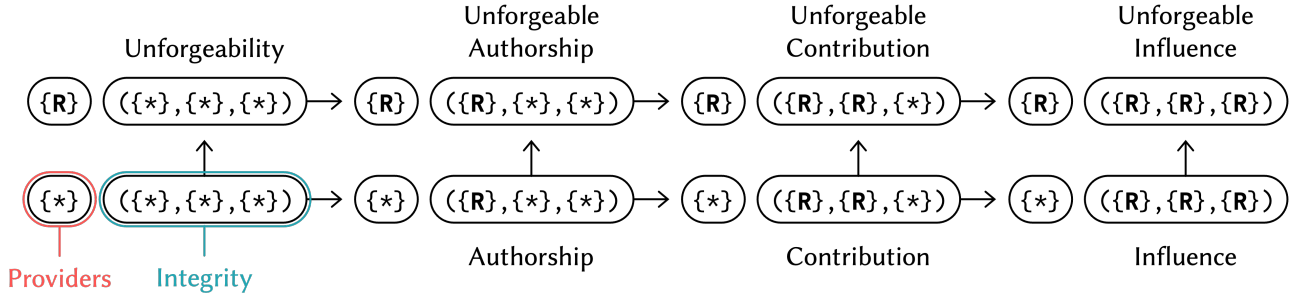


Figure 7. The capability design space. Each node represents a combination of provider and integrity annotations on the capability payload; edges represent implication (subtyping).

```

1 // Imagine this code appears in the standard library
2 impl Capability<PathBuf> {
3     #[cnbt::providers({R})]
4     pub fn join(self: &Self, other: &Self) → Self { ... }
5 }
6
7 // This code appears in the root crate
8 type PathCap = Capability<PathBuf>;
9
10 fn main() {
11     let join = |c1: &PathCap, c2: &PathCap| {
12         let result = c1.join(c2);
13
14         // Ensure 'parser' only touches CSV files
15         match result.extension().and_then(|e| e.to_str()) {
16             Some("csv") => result,
17             _ => panic!("Parser may only join CSV files"),
18         }
19     }
20
21     let c1 = PathCap::new("sheets");
22
23     for i in 0..10 {
24         let c2 = PathCap::new(format!("{i}.csv"));
25
26         // Give 'parser' the ability to join paths
27         parser::parse(join, c1, c2);
28     }
29 }

```

Figure 8. Higher-order capability. Trusted code can share the ability to create capabilities through higher-order functions like join.

```

1 let result = Command::new("touch")
2   .args(["output.txt"])
3   .output();

```

Figure 9. This command is harmless, but with a few changes it could do some real damage!

which origins may supply values at boundaries such as function arguments and record fields. In the evaluation rules for

```

1 impl Capability<Command> {
2     #[cnbt::providers({R})]
3     pub fn new(cmd: String) → Command { ... }
4 }

```

Figure 10. Command capability. Commands can be wrapped inside of capabilities just like paths.

the system, imported code is modeled using a concept we call *membranes*, written $(t)_{\circ}$, so that foreign origins are tracked explicitly.

Our formalization uses four metafunctions to represent common integrity operations concisely: *create* establishes the integrity triple for a newly constructed value, *adapt* defines the integrity triple of a variable, *compose* determines the integrity triple of a composite value, and *extract* returns the integrity triple of a value that was "extracted" from a composite value.

The system also supports set abstractions $\lambda w \subseteq O. t$, which let terms and types quantify over bounded origin sets and model polymorphic behavior over provenance. We refer to the bound variable w as an *origin set variable*. Rather than fixing a concrete origin set at definition time, it allows a single function to be type checked once and then instantiated with a different origin set at each call site. This is essential for typing generic library code whose provenance requirements depend on the call site rather than the library itself. For example, a standard library function such as `iter` cannot be given one fixed origin set annotation since it may be invoked from any crate. Instead, we type it using an origin set variable w that is instantiated with the origin set of the argument at each call site. We return to this example in Section 4.

To model Rust’s nominal struct types, we use the named record expression $N\{t\}$ where N is the type identifier and t is its field. Our system extends to records with any number of fields, but we use one for simplicity. We represent the capability data structure from Figure 3 as follows:

```
struct Capability { T({R},{R},{R}) | {R} } in <program>
```

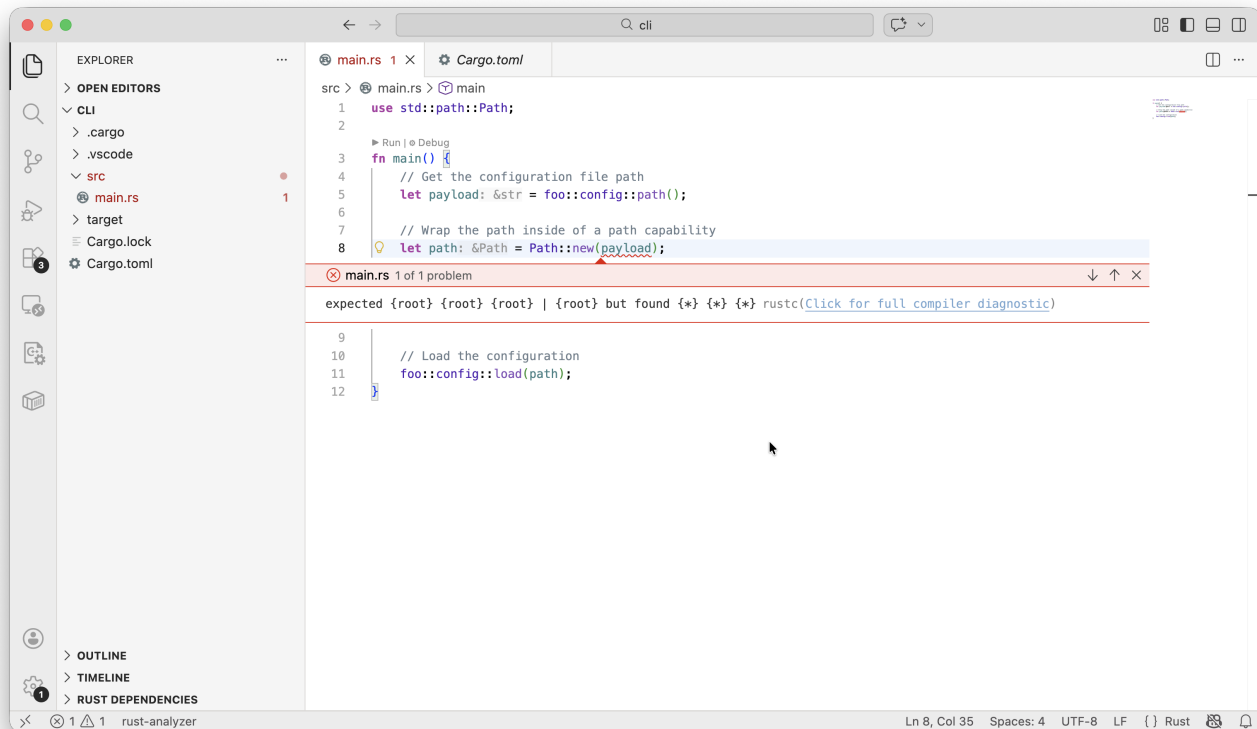


Figure 11. Example of an integrity error caught by Coenobita in VS Code.

We are able to prove that the resulting system satisfies progress and preservation. We are in the process of proving relevant noninterference and safety properties to the author and provider sets; we leave this for future work.

4 Implementation

We built Coenobita as a wrapper around the Rust compiler. Developers use the tool by annotating language constructs with their expected provider sets and integrity triples using the providers and integrity attributes, as seen in Figure 3. During compilation, Coenobita computes the provider set and integrity triple inferred for each construct and checks that each inferred set is a subset of the corresponding set in the programmer-provided annotation. If the check fails, Coenobita emits an error such as the one seen in Figure 11.

Coenobita treats existing standard library types such as `Path` and `PathBuf` as capabilities by layering provider and integrity annotations over the standard library. These annotations are written in a separate configuration file and describe the expected flows for standard library methods and constructors. For example, the configuration entry in Figure 12 states that paths must be constructed using data provided and created by the root origin.

```
1 fn Path::new({R} {R} {R} | {R}) → {R} {R} {R};
```

Figure 12. Example configuration file entry layered over the Rust standard library.

We had to overcome several obstacles in our implementation. One such obstacle concerns loop desugaring. In Figure 13, a `for` loop is transformed into a more primitive loop that repeatedly calls the `next` method on an iterator. This transformation hides the original loop body behind calls to standard library iterator methods, so a checker that only sees the desugared program must recover how the integrity of the source collection flows through those calls. Suppose the range `0..5` has integrity $(\{x\}, \{x\}, \{x\})$. The expression `0..5.into_iter()` on the first line influences the scope that follows. We do not annotate the definition of `into_iter` with provider and integrity information as it may be called from any crate, so the implicit flow context inside the `for` loop ends up being $\{*\}$ instead of $\{x\}$ as we might expect. Origin set variables (introduced in Section 3) for polymorphism are designed for this scenario – they propagate the integrity of the collection through the standard library call and into the desugared loop body.

```

1 // Original
2 for i in 0..5 {
3     println!("Count {i}")
4 }
5
6 // Desugared
7 match (0..5).into_iter() {
8     mut r => loop {
9         match r.next() {
10             None => break,
11             Some(i) => {
12                 println!("Count {i}")
13             }
14         }
15     }
16 }

```

Figure 13. Rust desugars for loops into more primitive iterator operations, as shown here. This posed an implementation challenge for Coenobita – integrity from the original iterator expression must be preserved through the implicit `into_iter` and `next` calls into the loop body.

```

1 // Function expects any 'AsRef<Path>'
2 fn read_config<P: AsRef<Path>>(path: P) → String {
3     // ...
4 }
5
6 // This code appears in the 'foo' crate
7 pub fn load() → String {
8     // Attempt to bypass the need for a capability
9     read_config("../..../cargo/credentials.toml")
10 }

```

error: cannot accept type 'str' for 'AsRef<Path>' due to capability safety policy
 --> src/lib.rs:27:17

```

27 |     read_config("../..../cargo/credentials.toml")
    |                  ~~~~~
    = help: allowed types are 'Path', 'PathBuf'

```

Figure 14. When a function expects any `AsRef<Path>`, it must already be either a `Path` or a `PathBuf`.

Another subtle obstacle was caused by traits. Many standard library file system utilities accept arguments of any type that implements `AsRef<Path>` rather than requiring the concrete `Path` or `PathBuf` type. This means attackers can bypass our restrictions on capability construction by providing raw strings instead of path capabilities. To close this loophole, Coenobita’s analysis enforces a strict policy for such conversion – when a function argument is generic over a trait like `AsRef<T>` and `T` is a capability type, the supplied argument must already be of type `T`. See Figure 14 for an example.

Rust’s type safety and memory safety guarantees only apply to *safe* Rust, because the compiler disables certain checks inside unsafe blocks. Unsafe functions such as `transmute` are out of scope for our safety guarantees.

5 Evaluation

We evaluated Coenobita by modifying popular crates to use our capability-safe abstractions. We aim to answer the following research questions. For each question, we compare the modified crate to the original.

Q1 What is the runtime overhead?

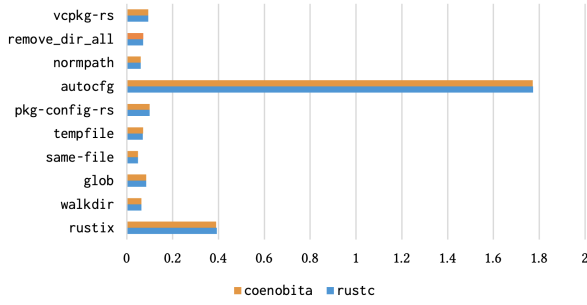
Q2 What is the programming overhead?

Q3 What is the compilation overhead?

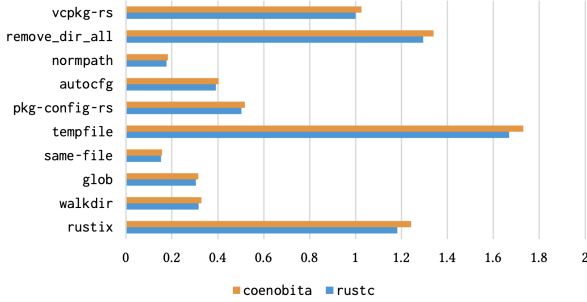
Experimental Setup. We ran our experiments on a 2021 MacBook Pro (Apple M1 Pro, 16 GB RAM) running macOS Monterey. For each crate and its modified variant, we ran a Python script to execute both of their test suites in batches of size m . We chose m so that a single batch would take roughly ten seconds, then estimated the average runtime of a single run by dividing the cumulative time by m , across 30 sample runs. We measured differences in compilation time using a similar method, comparing against `rustc`.

Chosen Crates. We chose a selection of crates that extensively interact with the file system. Crates `rustix`, `walkdir`, `glob`, and `remove_dir_all` appear among the top ten crates in the `filesystem` category on `crates.io` based on total downloads, while `normpath` and `tempfile` are dependencies of some of the above.

To modify each crate, we iteratively ran Coenobita and manually performed a series of transformations until Coenobita no longer emitted errors. Two common classes of errors were as follows. First, Coenobita reported violations of its integrity and provider rules when library crates directly invoked methods that manipulate paths, such as `join`. We addressed these errors by replacing occurrences of `p1.join(p2)` with `join(p1, p2)`, where `join` is a closure passed into the crate explicitly by the root crate. An example of this pattern appears in Figure 16, where we change a function in `tempfile` to make it capability-safe. Similar changes were made to replace the `push`, `pop`, and `collect` methods. Second, Coenobita reported uses of standard library conversions, such as string arguments where types implementing `AsRef<Path>` were expected. We addressed these errors by changing the relevant signatures and call sites so that path arguments already have the required path type. While these modifications often propagate throughout the crate, requiring changes in many different locations, we found that they often follow predictable patterns and could be automated in principle.



(a) Compared test times.



(b) Compared build times.

Figure 15. Comparison of test times before and after retrofitting and build times under Coenobita versus rustc. The horizontal axis represents seconds.

Crate	Total LOC	Modified LOC
rustix*	44298	1
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

Table 1. Programming overhead results: total lines of code per crate vs. lines changed or added to make the crate capability-safe. Changes to satisfy the tests and examples are not included. For crates marked *, see explanation in the text.

5.1 (Q1) Runtime Overhead

The results of the first research question are shown in Figure 15a. The bars are statistically indistinguishable for all crates. Using a paired two-tailed t -test, we fail to reject the null hypothesis of no runtime difference since $p > 0.05$ for every crate. The full analysis appears in the Appendix, Table 2. We conclude that the **changes required to comply**

with our tool do not incur any measurable runtime performance overhead for this selection of Rust crates.

5.2 (Q2) Programming Overhead

The results of the second research question are shown in Table 1. We estimated programming overhead by counting the lines of code we modified in every crate to satisfy Coenobita. The table shows that most crates required few or no source changes, with the largest retrofit modifying 74 lines. We conclude that **programming overhead tends to be low** because most crate code can be reused unchanged and annotations remain localized.

Qualitatively, Coenobita integrates smoothly with rust-analyzer and VS Code. Provider and integrity errors appear in the IDE alongside regular compiler errors. Provider and integrity attributes only need to be added in the root crate and in the standard library.

Some crates (marked with an asterisk * in the table) require almost no modification because of their shape rather than the absence of risk. For instance, `rustix` is primarily a thin wrapper around raw system calls. Furthermore, it does not directly manipulate paths – it either routes them into safe standard library utilities or across the FFI boundary into C functions, which is unsafe and out of scope. The `walkdir` crate requires no changes because it acts as an intermediary – it simply passes paths from the caller to the standard library and does not directly manipulate them.

We did not need to use the `trust` (endorsement) macro described in Section 2.2 in any of these crates because we evaluate each one against its own test suite in isolation. Therefore, there is no independent “root” consumer crate with a reason to accept tainted data. A real application that depends on these crates may need to endorse data using the `trust` macro. For example, it may need to indicate that a path returned from the `glob` crate is safe. Measuring how often this occurs in practice requires evaluating entire applications rather than standalone crates, which we leave to future work.

5.3 (Q3) Compilation Overhead

The results of the third research question are shown in Figure 15b. The figure compares the build time of each modified crate under Coenobita against the build time of the same crate under rustc alone. The full measurements appear in the Appendix, Table 3. Compiling the modified variant of each crate using Coenobita adds between 2.6% and 5.1% to the build time (3.6% on average). We conclude that Coenobita **has a small compilation overhead**.

6 Expressiveness

In this section, we study the expressiveness of Coenobita qualitatively by comparing it on some example programs to three existing systems: Shill, Joe-E, and cap-std.

```

1 pub fn create(dir: &Path) → io::Result<File> {
2     create_unix(dir)
3 }

```

(a) Original `unix::create`.

```

1 pub fn create<J>(dir: &Path, join: J) → io::Result<File>
2 where
3     J: Fn(&OsStr, &OsStr) → PathBuf
4 {
5     create_unix(dir, join)
6 }

```

(b) Capability-safe `unix::create`.

Figure 16. Example of the transformation required to replace illegal calls to `join` in the `tempfile` crate, showing the original function (a) and the modified version (b). Since Coenobita treats `Path` and `PathBuf` as capabilities, the revised version accepts a `join` closure from the root crate and threads it through to call sites that require it, instead of invoking `Path::join` directly.

<pre> copy : { from : dir(contents, lookup, read), to : dir(create-file, create-dir, lookup) } → any fn copy(from: dir!("contents", "lookup", "read"), to: dir!("create-file", "create-dir", "lookup")) → Any </pre>	Shill Coenobita
---	-------------------------------

Figure 17. A Shill contract and its Coenobita counterpart. Both require the caller to provide directory capabilities with the permissions needed to copy files.

Shill is a scripting language that enforces the *principle of least privilege* [30]. Scripts written using Shill access system resources via capabilities provided by the caller, and the Shill sandbox prevents them from violating their stated contracts. Coenobita can model these programs, as shown in Figure 17 with a simplified contract for a script that copies files and its equivalent implementation in Coenobita. One important difference is that Shill capabilities carry permissions as runtime values, whereas Coenobita can encode these permissions statically by representing a set of privileges as a tuple of `Grant` and `Deny` types, as shown in Figure 18. This encoding allows Coenobita to mirror the permission checks Shill performs at compile time while also enforcing stronger properties, such as the integrity of the path stored inside a directory capability.

Joe-E is a subset of Java that observes the object-capability model [27]. Code written using Joe-E can be statically checked and safely used within existing Java applications.

Definitions

```

// Permission markers
struct Grant;
struct Deny;

// Permission traits
trait CreateFile {}
trait ReadFile {}
...

impl<R, W, D> CreateFile for (Grant, R, W, D) {}
impl<C, W, D> ReadFile for (C, Grant, W, D) {}
...

// Represents the directory at 'inner'
struct Dir<P> {
  #[cnbt::providers({R})]
  inner: PathBuf,
  _phantom: PhantomData<P>
}

impl<P: CreateFile> Dir<P> {
  fn create_file(&self, name: PathBuf) → File {
    // Create a file inside this directory...
  }
}

```

Usage

```

let dir1: Dir<(Grant, Deny, Deny, Deny)>
  = Dir::new("output");
let dir2: Dir<(Deny, Grant, Deny, Deny)>
  = Dir::new("output");

// OK, first position is 'Grant'
dir1.create_file("log.txt");

// ERROR, first position is 'Deny'
dir2.create_file("log.txt");

```

Figure 18. A static encoding of directory permissions in Coenobita. The `create_file` method is available only when the parameter `P` implements the corresponding trait.

Figure 19 shows a Joe-E decoder interface and a corresponding Rust trait. As long as the struct implementing `Decoder` does not contain file capabilities and unsafe Rust is not used, we know that multiple invocations of `decode` will be independent, and that no state will leak between invocations.

`cap-std` is a capability-based version of the Rust standard library maintained by the Bytecode Alliance [1]. It protects file system resources using coarse-grained capabilities [1]. Crates using `cap-std` cannot directly access files. Instead, every file must be accessed through a capability for its parent directory. Directory capabilities can only be generated using several privileged methods that accept an `AmbientAuthority` as an argument. Unlike Coenobita, their system is enforced dynamically, and interactions with the file

```

Joe-E
public interface Decoder extends Immutable {
    byte[][][] decode(byte[] imagedata);
}

Coenobita
pub trait Decoder {
    fn decode(&self, imagedata: &[i8]) → Bitmap;
}

```

Figure 19. A Joe-E immutable decoder interface and the corresponding Rust trait checked by Coenobita.

```

// Suppose this appears in a crate called 'math'
pub fn add(a: i32, b: i32) → i32 {
    let a = ambient_authority();
    let dir = Dir::open_ambient_dir("../foo", a).unwrap();

    dir.create("baz")
        .and_then(|mut file| file.write(b"This is unsafe!"));

    return a + b;
}

```

Figure 20. A cap-std function that obtains ambient authority internally. The file system effect is hidden from the function signature.

system may fail to appear in the public interface of a third-party crate if it misuses ambient authority. The example in Figure 20 is impossible to write in Coenobita.

Overall, these comparisons show that Coenobita can express common capability patterns while adding additional integrity guarantees. Like Shill, it can require callers to provide the capabilities needed for an operation. Like Joe-E, it can rule out hidden authority. Like cap-std, it supports capability-based file system access in Rust.

7 Related Work

7.1 Capabilities

Dennis and Van Horn first proposed capabilities in 1966 as a security mechanism for multiprogrammed machines [11]. Levy provides a comprehensive treatment of capability-based systems developed in the following decades [24]. Hydra was a pioneering capability-based operating system that introduced the object-capability model for fine-grained access control [46]. It was followed by KeyKOS and EROS, which refined capability-based microkernel designs with strong compartmentalization [16, 37]. Amoeba brought capabilities to distributed systems [39]. CHERI’s memory protection and software compartmentalization mechanisms for RISC architectures are based on capabilities [44].

At the language level, the E programming language was among the first explicitly designed around capabilities [28]. Building on E’s foundations, Caja brought capability-based

security to JavaScript [29] by combining static verification and runtime checks to transform arbitrary JavaScript into a capability-safe subset with embedded third-party code. Mettler’s Joe-E, like E, enforces capability safety by restricting how programs obtain references to objects [27, 28]. Wyvern [8, 22, 36] adopts a capability-safe language design, in particular in its module system to manage resource access [26]. Coenobita shares a similar goal of making authority explicit, but its capability abstraction is designed to protect resources as individual data, rather than by providing specific effectful API(s) or wrappers over them to modules. Our higher-order capabilities bear resemblance to Wyvern’s design. Shill [30] use capabilities to protect shell scripts, and cap-std [1] implement a capability system for the filesystem in Rust; we compare with these works in Section 6. Information flow (integrity constraints) are not supported by these systems.

Other systems use capabilities to control particular classes of effects. For example, LaCasa uses object capabilities in Scala to support safe concurrency [15], and Pony uses reference capabilities to prevent data races [7].

7.2 Information Flow Control

Information flow control originated with Denning’s lattice model [10] and appeared in early systems such as ADEPT-50 [45] and MITRE [3]. The programming languages community later adapted these ideas at the language level. JFlow [31] and Jif [34] statically track information flow in Java using decentralized security labels [32, 33]. SLam presents a lambda calculus model that tracks both direct and indirect influence on values [19] (our first cut to formalize the type system was based on SLam). Myers, Sabelfeld, and Zdancewicz [35] study robustness for programs with declassification. This is relevant to the *confused deputy* example in our system in Section 2.

LIO uses Haskell’s monadic structure to implement dynamic information flow control [38]. Cocoon enforces static confidentiality in Rust using macros rather than relying on compiler modifications [23]. Coenobita also targets Rust, but enforces integrity rather than confidentiality and combines information flow control with capability safety. Another recent work, Flowistry, takes advantage of ownership and mutability annotations to perform modular information flow analysis in Rust [9]. It would be interesting to investigate if similar techniques (using ownership and mutability information) could improve our analysis.

7.3 Capabilities + Information Flow Control

Ours is not the first system to combine capabilities and information flow. Dimoulas et al. [13] show that capabilities alone cannot secure programs that mix trusted and untrusted code, and propose a system that combines capabilities, access control, and integrity tracking. Unlike in Coenobita, their access control is enforced dynamically through contracts

at use time. Gouni et al. [14] present a substructural type system in which capabilities, quantitative information flow, and privilege changing protocols arise as fragments of a single dependency tracking framework. The goals of this work are more general than Coenobita, which is intended to be more specialized to Rust. Birgisson et al. [4] connect capabilities to information flow by inlining a dynamic monitor that hands code segments only the write capabilities their guards and data permit, which enforces noninterference for confidentiality in a simple imperative language.

References

- [1] Bytecode Alliance. 2025. cap-std: Capability-based version of Rust’s standard library. <https://github.com/bytecodealliance/cap-std>. Accessed: 2025-03-10.
- [2] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 270 (Oct. 2023), 31 pages. doi:10.1145/3622846
- [3] David E. Bell and Leonard J. LaPadula. 1973. *Secure Computer Systems: Mathematical Foundations and Model*. Technical Report M74-244. The MITRE Corporation, Bedford, MA.
- [4] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. 2011. Capabilities for information flow. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security* (San Jose, California) (PLAS ’11). Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. doi:10.1145/2166956.2166961
- [5] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 21 (Nov. 2023), 52 pages. doi:10.1145/3618003
- [6] Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering purity with comonads and capabilities. *Proc. ACM Program. Lang.* 4, ICFP, Article 111 (Aug. 2020), 28 pages. doi:10.1145/3408993
- [7] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Pittsburgh, PA, USA) (AGERE! 2015). Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2824815.2824816
- [8] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for free. In *International Conference on Formal Engineering Methods*. Springer, 231–247.
- [9] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular information flow through ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3519939.3523445
- [10] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976), 236–243. doi:10.1145/360051.360056
- [11] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (March 1966), 143–155. doi:10.1145/365230.365252
- [12] Rust Project Developers. 2026. RUSTSEC-2026-0260: arrayref: ‘arrayref’ 0.3.10 was removed from crates.io due to a malicious dependency | RustSec Advisory Database. <https://rustsec.org/advisories/{R}{U}{S}{T}{S}{E}{C}-2026-0260>. [Accessed 06-10-2026].
- [13] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *2014 IEEE 27th Computer Security Foundations Symposium*. 3–17. doi:10.1109/CSF.2014.9
- [14] Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2026. Security Reasoning via Substructural Dependency Tracking. *Proc. ACM Program. Lang.* 10, POPL, Article 27 (Jan. 2026), 29 pages. doi:10.1145/3776669
- [15] Philipp Haller and Alex Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) (OOPSLA 2016). Association for Computing Machinery, New York, NY, USA, 272–291. doi:10.1145/2983990.2984042
- [16] Norman Hardy. 1985. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.* 19, 4 (Oct. 1985), 8–25. doi:10.1145/858336.858337
- [17] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (Oct. 1988), 36–38. doi:10.1145/54289.871709
- [18] Muhammad Hassnain and Caleb Stanford. 2024. Counterexamples in Safe Rust. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 128–135.
- [19] Nevin Heintze and Jon G. Riecke. 1998. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) (POPL ’98). Association for Computing Machinery, New York, NY, USA, 365–377. doi:10.1145/268946.268976
- [20] Graydon Hoare. 2010. Project Servo: Technology from the past came to save the future from itself – Mozilla Annual Summit. <http://venge.net/graydon/talks/intro-talk-2.pdf>. [PDF, accessed 10-07-2025. Archive link: <https://archive.ph/20211226213836/http://venge.net/graydon/talks/intro-talk-2.pdf>].
- [21] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.
- [22] Darya Kurilova, Alex Potanin, and Jonathan Aldrich. 2014. Wyvern: Impacting software security via programming language design. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. 57–58.
- [23] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. 2024. Cocoon: Static Information Flow Control in Rust. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 100 (April 2024), 28 pages. doi:10.1145/3649817
- [24] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, USA.
- [25] Rust Security Team Manish Goregaokar. 2026. Supply chain attack on arrayref | Rust Blog. <https://blog.rust-lang.org/2026/08/20/supply-chain-attack-on-arrayref>. [Accessed 06-10-2026].
- [26] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:27. doi:10.4230/LIPIcs.ECOOP.2017.20
- [27] Adrian Mettler, David A Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java.. In *NDSS*, Vol. 10. 357–374.
- [28] Mark Samuel Miller. 2006. *Robust composition: towards a unified approach to access control and concurrency control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA. Advisor(s) Shapiro, Jonathan S.
- [29] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in sanitized JavaScript. Google, Inc., Technical Report. Accessed at <https://google-code-archive-downloads.storage.googleapis.com/v2/code.google.com/google-caja/caja-spec-2008-06-06.pdf>.
- [30] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: A Secure Shell Scripting Language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX

- Association, Broomfield, CO, 183–199. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/moore>
- [31] Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) (POPL '99). Association for Computing Machinery, New York, NY, USA, 228–241. doi:10.1145/292540.292561
- [32] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) (SOSP '97). Association for Computing Machinery, New York, NY, USA, 129–142. doi:10.1145/268998.266669
- [33] Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.* 9, 4 (Oct. 2000), 410–442. doi:10.1145/363516.363526
- [34] Andrew C. Myers, Nathan Nystrom, Lantian Zheng, and Steve Zdancewic. 2001. Jif: Java Information Flow, Software Release. Online. Available: <http://www.cs.cornell.edu/jif>.
- [35] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. 2006. Enforcing robust declassification and qualified robustness. *Journal of Computer Security* 14, 2 (2006), 157–196.
- [36] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance*. 9–16.
- [37] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) (SOSP '99). Association for Computing Machinery, New York, NY, USA, 170–185. doi:10.1145/319151.319163
- [38] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. *SIGPLAN Not.* 46, 12 (Sept. 2011), 95–106. doi:10.1145/2096148.2034688
- [39] Andrew S Tanenbaum, Sape J Mullender, and Robbert Van Renesse. 1986. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*. IEEE, 558–563.
- [40] Rust Security Team. 2022. RUSTSEC-2022-0042: malicious crate 'rust-decimal'. RustSec Advisory Database – <https://rustsec.org/advisories/RUSTSEC-2022-0042.html>. [Accessed 21-07-2026].
- [41] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*. 137–148.
- [42] Speykious (GitHub user). 2024. cve-rs: Blazingly fast memory vulnerabilities, written in 100% safe Rust. — [github.com](https://github.com/Speykious/cve-rs). <https://github.com/Speykious/cve-rs>. [Accessed 11-11-2024].
- [43] David Walker, Karl Cray, and Greg Morrisett. 2000. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (July 2000), 701–771. doi:10.1145/363911.363923
- [44] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. doi:10.1109/SP.2015.9
- [45] C. Weissman. 1969. Security controls in the ADEPT-50 time-sharing system. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference* (Las Vegas, Nevada) (AFIPS '69 (Fall)). Association for Computing Machinery, New York, NY, USA, 119–133. doi:10.1145/1478559.1478574
- [46] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: the kernel of a multiprocessor operating system. *Commun. ACM* 17, 6 (June 1974), 337–345. doi:10.1145/355616.364017

- [47] Lydia Zoghbi, David Thien, Ranjit Jhala, Deian Stefan, and Caleb Stanford. 2024. Auditing Rust crates effectively. Preprint at <https://web.cs.ucdavis.edu/~cdstanford/doc/2024/CargoScan-draft.pdf>.

A Formalization

A.1 Syntax

We extend the syntax of the simply typed lambda calculus with origin sets, named records, and other constructs.

Term $t ::= x \mid v \mid t t \mid t O \mid t + t \mid N \{t\}$
 $\mid \text{proj } t \mid \text{struct } N \{S^P\} \text{ in } t$
 $\mid \text{if } t \text{ then } t \text{ else } t$
 $\mid \text{let } x = t \text{ in } t$
 $\mid \langle t \rangle_o$

Value $v ::= n \mid \lambda x : S^P. t \mid \lambda w \subseteq O. t$
 $\mid N \{v\} \mid \langle t \rangle_o$

Type $T ::= \text{Int} \mid \text{Unit} \mid S^P \rightarrow S \mid N \{S^P\}$

Secure Type $S ::= T^\kappa \mid \forall w \subseteq O. S$

Origin Set $O ::= \{o_1 \dots o_m\} \mid O \cup O \mid w$

Provider Set $p ::= \{o_1 \dots o_m\}$

Type Context $\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, N \{S^P\}$
 $\mid \Gamma, w \subseteq O$

Influence $I ::= \emptyset \mid I, O$

A.2 Typing Rules

In this section, we sketch out the type system that guided our implementation of Coenobita. We distinguish between *types* and *secure types*. A type, such as Int , simply describes the shape of a value. A secure type like Int^κ is more informative, annotating the type with an integrity triple $\kappa = (a, c, i)$ that captures the provenance of the value. These triples track author (a), contributor (c), and influencer (i) origins, which are crates in our case. We will see that these sets must satisfy the invariant $a \subseteq c \subseteq i$.

Furthermore, our system uses *provider sets* to restrict which origins are permitted to supply values to functions and records. For instance, a function of type $S^P \rightarrow^\kappa S$ can only be called by an origin in the set p . This mechanism is fundamental to guaranteeing that capabilities cannot be forged.

The provider set is a meaningful annotation only where values cross boundaries between origins. Consider a function defined in crate a that is called by crate b . The function serves as a "portal" through which values from b can pass into a , and the provider annotation on the function argument type controls this portal.

Records also act as portals between crates. When crate *b* constructs a record, it provides the value within. Suppose crate *b* then passes the record to crate *c*, which passes the record to crate *a*, which extracts the value from the record. That means crate *b* provided *a* with data without communicating directly! If crate *a* wishes to avoid this scenario, it can annotate the field of the record it accepts with a provider set of $\{c\}$. Now, crate *a* can only receive records containing values provided by crate *c*.

Notational Conventions. We introduce some additional notation to keep our rules short and simple. We use κ to describe an integrity triple (a, c, i) and define $\kappa \subseteq \kappa'$ as the pointwise extension of subset inclusion.

$$\begin{aligned}\kappa &\triangleq (a, c, i) \\ \kappa \subseteq \kappa' &\triangleq \begin{aligned} &\kappa.a \subseteq \kappa'.a \wedge \\ &\kappa.c \subseteq \kappa'.c \wedge \\ &\kappa.i \subseteq \kappa'.i \end{aligned}\end{aligned}$$

We also present four metafunctions to represent common integrity triple operations concisely. Function `create` establishes the integrity triple for a newly constructed value while `adapt` defines the integrity triple of a variable. Function `compose` determines the integrity triple of a composite value, and `extract` returns the integrity triple of a value that was "extracted" from a composite value.

$$\begin{aligned}\text{create}(I, o) &\triangleq (\{o\}, \{o\}, \{o\} \cup I) \\ \text{compose}(o, \kappa_{j=1}^n) &\triangleq (\{o\}, \{o\} \cup (\bigcup_{j=1}^n \kappa_j.c), \{o\} \cup (\bigcup_{j=1}^n \kappa_j.i)) \\ \text{extract}(\kappa, \kappa') &\triangleq (\kappa.a, \kappa.c, \kappa.i \cup \kappa'.i) \\ \text{adapt}(\kappa, I) &\triangleq (\kappa.a, \kappa.c, \kappa.i \cup I)\end{aligned}$$

Rules for primitive values like integers (T-INT) and units (T-UNIT) use `create`, establishing that the current origin is the author, contributor, and influencer. Values composed of other terms, such as records (T-RCD) and arithmetic sums (T-ADD) are annotated using `compose` to reflect that the contributors and influencers of their subterms are inherited by the overall value. When a component is projected from a composite value (T-PROJ) or a function is applied (T-APP), its integrity is updated using `extract` to account for the origins that influenced the composite structure that contained it.

The final metafunction, `adapt`, handles values as they cross contexts. It updates an integrity triple by adding the current influence *I* to the influencer set of a term. The T-VAR rule applies `adapt` whenever a variable is used, ensuring the type reflects local influences. More critically, this function allows the system to handle external code safely. The T-MEMB rule

uses `adapt` to type a special membrane term, written $(\llbracket t \rrbracket)_o$, which indicates that the term *t* comes from a foreign origin *o*. To ensure the origin of all external code is tracked, the system treats an import statement as syntactic sugar for a let binding that wraps the imported value in a membrane. This rigorous provenance tracking is essential for preventing supply chain attacks.

$$\text{from } o \text{ import } x = t_1 \text{ in } t_2 \triangleq \text{let } x = (\llbracket t_1 \rrbracket)_o \text{ in } t_2$$

The set abstraction rule T-SETABS is formulated so that the invariant $a \subseteq c \subseteq i \subseteq \top$ always holds regardless of whether any one or more of these origin sets are actually variables. If an origin is an author of a value, then it is also a contributor and an influencer by extension, since contribution and influence are transitive forms of authorship. In other words, authorship is a stronger property than contribution, which is in turn a stronger property than influence. To avoid writing three separate rules for set abstraction, we introduce another metafunction `bound` that, given a set variable *w* and the origin set *O* that serves as its upper bound, returns an integrity triple with *w* in its proper location for the subset relationship to hold.

$$\text{bound}(w, O) = \begin{cases} (\underline{w}, c, i) & \text{if } O \subseteq c \\ (a, \underline{w}, i) & \text{if } O \subseteq i \\ (a, c, \underline{w}) & \text{if } O \subseteq \top \end{cases}$$

Some rules have side conditions to ensure that only permitted origins can supply a construct with values. For instance, rule T-APP requires the current origin *o* to be a member of the provider set *p* that annotates the type of the input value. This constraint ensures that only origins defined in *p* can apply the abstraction.

$$\boxed{I; \Gamma \vdash_o t : S}$$

$$I; \Gamma \vdash_o n : \text{Int}^{\text{create}(I, o)} \quad (\text{T-INT})$$

$$I; \Gamma \vdash_o () : \text{Unit}^{\text{create}(I, o)} \quad (\text{T-UNIT})$$

$$\frac{I; \Gamma \vdash_o t_1 : S_1 \quad I; \Gamma, x : S_1 \vdash_o t_2 : S_2}{I; \Gamma \vdash_o \text{let } x = t_1 \text{ in } t_2 : S_2} \quad (\text{T-LET})$$

$$\frac{I; \Gamma, x : S \vdash_o t : T^\kappa}{I; \Gamma \vdash_o \lambda x : S^p. t : S^p \rightarrow^{\text{compose}(o, \kappa)} T^\kappa} \quad (\text{T-ABS})$$

$$\frac{x : T^\kappa \in \Gamma}{I; \Gamma \vdash_o x : T^{\text{adapt}(\kappa, I)}} \quad (\text{T-VAR})$$

$$o \in p \quad \frac{I; \Gamma \vdash_o t_1 : S^p \rightarrow^{\kappa_1} T^\kappa \quad I; \Gamma \vdash_o t_2 : S}{I; \Gamma \vdash_o t_1 t_2 : T^{\text{extract}(\kappa, \kappa_1)}} \quad (\text{T-APP})$$

$$\frac{I; \Gamma, w \subseteq O \vdash_o t : T^{\text{bound}(w, O)}}{I; \Gamma \vdash_o \lambda w \subseteq O. t : \forall w \subseteq O. T^{\text{bound}(w, O)}} \text{ (T-SETABS)}$$

$$\frac{I; \Gamma \vdash_o t : \forall w \subseteq O_1. S \quad \Gamma \vdash_o O \subseteq O_1}{[w \mapsto O] I; \Gamma \vdash_o t O : [w \mapsto O] S} \text{ (T-SETAPP)}$$

$$\frac{I; \Gamma \vdash_o t_1 : \text{Int}^{\kappa_1} \quad I; \Gamma \vdash_o t_2 : \text{Int}^{\kappa_2}}{I; \Gamma \vdash_o t_1 + t_2 : \text{Int}^{\text{compose}(o, \kappa_1, \kappa_2)}} \text{ (T-ADD)}$$

$$\frac{I; \Gamma, N \{S_1^p\} \vdash_o t : S_2}{I; \Gamma \vdash_o \text{struct } N \{S_1^p\} \text{ in } t : S_2} \text{ (T-STRUCT)}$$

$$o \in p \quad \frac{I; \Gamma \vdash_o t : T^\kappa \quad N \{T^\kappa \mid p\} \in \Gamma}{I; \Gamma \vdash_o N \{t\} : N \{T^\kappa \mid p\}^{\text{compose}(o, \kappa)}} \text{ (T-RCD)}$$

$$\frac{I; \Gamma \vdash_o t : N \{T^\kappa \mid p\}^{\kappa_1}}{I; \Gamma \vdash_o \text{proj } t : T^{\text{extract}(\kappa, \kappa_1)}} \text{ (T-PROJ)}$$

$$\frac{I; \Gamma \vdash_o t_1 : \text{Int}^{(a, c, i)} \quad I, i; \Gamma \vdash_o t_2 : S \quad I, i; \Gamma \vdash_o t_3 : S}{I; \Gamma \vdash_o \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : S} \text{ (T-IF)}$$

$$\frac{I; \Gamma \vdash_o t : S_1 \quad \Gamma \vdash_o S_1 <:_S S_2}{I; \Gamma \vdash_o t : S_2} \text{ (T-SUB)}$$

$$\frac{\{o\}; \emptyset \vdash_o t : T^\kappa}{I; \Gamma \vdash_{o_1} \llbracket t \rrbracket_o : T^{\text{adapt}(\kappa, I)}} \text{ (T-MEMB)}$$

The first subtyping judgment, $\Gamma \vdash_o T_1 <:_T T_2$, relates ordinary types without considering integrity annotations. It is reflexive and transitive. Function types follow the standard rule – argument types are contravariant while result types are covariant. Record subtyping requires the same record name and provider set, and the contained secure types must be mutually compatible.

$$\boxed{\Gamma \vdash_o T <:_T T}$$

$$\Gamma \vdash_o T <:_T T \text{ (SUBT-REFL)}$$

$$\frac{\Gamma \vdash_o T_1 <:_T T_2 \quad \Gamma \vdash_o T_2 <:_T T_3}{\Gamma \vdash_o T_1 <:_T T_3} \text{ (SUBT-TRANS)}$$

$$\frac{\Gamma \vdash_o S_2 <:_S S_1 \quad \Gamma \vdash_o S_3 <:_S S_4}{\Gamma \vdash_o S_1^p \rightarrow S_3 <:_T S_2^p \rightarrow S_4} \text{ (SUBT-ARR)}$$

$$\frac{\Gamma \vdash_o S_1 <:_S S_2 \quad \Gamma \vdash_o S_2 <:_S S_1}{\Gamma \vdash_o N \{S_1^p\} <:_T N \{S_2^p\}} \text{ (SUBT-RCD)}$$

The secure subtyping judgment, $\Gamma \vdash_o S_1 <:_S S_2$, lifts subtyping to secure types. Universally quantified origin set types are compared under the same bound. For concrete secure types, the underlying ordinary types must be in the subtype relation, and the integrity triple on the left must be pointwise contained in the one on the right.

$$\boxed{\Gamma \vdash_o S <:_S S}$$

$$\frac{\Gamma, w \subseteq O \vdash_o S_1 <:_S S_2}{\Gamma \vdash_o \forall w \subseteq O. S_1 <:_S \forall w \subseteq O. S_2} \text{ (SUBS-SETALL)}$$

$$\frac{\Gamma \vdash_o T_1 <:_T T_2 \quad \Gamma \vdash_o \kappa_1 \subseteq \kappa_2}{\Gamma \vdash_o T_1^{\kappa_1} <:_S T_2^{\kappa_2}} \text{ (SUBS-INTEGRITY)}$$

A.3 Evaluation Rules

We model program execution using the eager small-step operational semantics defined below. Our rules are based on those of the simply typed lambda calculus. Our evaluation relation has the form $t \rightarrow_o t'$, which means the term t can take a single step of computation in origin o to t' .

$$\boxed{t \rightarrow_o t'}$$

$$\frac{t_1 \rightarrow_o t'_1}{t_1 t_2 \rightarrow_o t'_1 t_2} \text{ (E-APP}_1\text{)} \quad \frac{t_2 \rightarrow_o t'_2}{v_1 t_2 \rightarrow_o v_1 t'_2} \text{ (E-APP}_2\text{)}$$

$$(\lambda x : T. t) v \rightarrow_o [x \mapsto v]t \text{ (E-APP}_3\text{)}$$

$$\frac{t \rightarrow_o t'}{t O \rightarrow_o t' O} \text{ (E-SETAPP}_1\text{)}$$

$$(\lambda w \subseteq O_1. t) O \rightarrow_o [w \mapsto O]t \text{ (E-SETAPP}_2\text{)}$$

$$\frac{t_1 \rightarrow_o t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow_o \text{let } x = t'_1 \text{ in } t_2} \text{ (E-LET}_1\text{)}$$

$$\text{let } x = v \text{ in } t \rightarrow_o [x \mapsto v]t \text{ (E-LET}_2\text{)}$$

$$\frac{t \rightarrow_o t'}{N \{t\} \rightarrow_o N \{t'\}} \text{ (E-RCD)}$$

$$\frac{t \rightarrow_o t'}{\text{proj } t \rightarrow_o \text{proj } t'} \text{ (E-PROJ}_1\text{)}$$

$$\text{proj } N \{v\} \rightarrow_o v \text{ (E-PROJ}_2\text{)}$$

$$\frac{t \rightarrow_o t'}{\text{struct } N \{T\} \text{ in } t \rightarrow_o \text{struct } N \{T\} \text{ in } t'} \text{ (E-STRUCT}_1\text{)}$$

$$\text{struct } N \{T\} \text{ in } v \rightarrow_o v \text{ (E-STRUCT}_2\text{)}$$

$$\begin{array}{c}
\frac{t_1 \rightarrow_o t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow_o \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \text{ (E-If)} \\
\\
\frac{n \neq 0}{\text{if } n \text{ then } t_2 \text{ else } t_3 \rightarrow_o t_2} \text{ (E-IfTrue)} \\
\\
\frac{n = 0}{\text{if } n \text{ then } t_2 \text{ else } t_3 \rightarrow_o t_3} \text{ (E-IfFalse)} \\
\\
\frac{t_1 \rightarrow_o t'_1}{t_1 + t_2 \rightarrow_o t'_1 + t_2} \text{ (E-ADD}_1\text{)} \quad \frac{t_2 \rightarrow_o t'_2}{v_1 + t_2 \rightarrow_o v_1 + t'_2} \text{ (E-ADD}_2\text{)} \\
\\
n_1 + n_2 \rightarrow_o n_3 \quad \text{where } n_3 = n_1 + n_2 \text{ (E-ADD}_3\text{)}
\end{array}$$

One important departure from the otherwise standard rules concerns values crossing between different origins. To track this, our semantics introduces the concept of an origin membrane. When a value v is passed to a foreign origin o_1 from origin o , it is wrapped in a membrane, which we notate as $\langle v \rangle_{o_1}$. This wrapping mechanism is crucial for maintaining type preservation. It ensures that the type of a value from one crate is interpreted correctly when used within another. For instance, without this explicit tracking, applying a foreign function to an integer might cause the type system to lose track of the integer's true origin, leading to a failure of type soundness. The evaluation rules below extensively involve origin membranes.

$$\begin{array}{c}
\frac{t \rightarrow_{o_1} t'}{\langle t \rangle_{o_1} \rightarrow_o \langle t' \rangle_{o_1}} \text{ (E-MEMB}_1\text{)} \\
\\
\langle \langle v \rangle_{o_1} \rangle_{o_2} \rightarrow_o \langle v \rangle_{o_1} \text{ (E-MEMB}_2\text{)} \\
\\
\langle \lambda x : T^p. t \rangle_{o_1} v \rightarrow_o \langle [x \mapsto \langle v \rangle_{o_1}] t \rangle_{o_1} \text{ (E-APPFRN)} \\
\\
\langle \lambda w \subseteq O_1. t \rangle_{o_1} O \rightarrow_o \langle [w \mapsto O] t \rangle_{o_1} \text{ (E-SETAPPFRN)} \\
\\
\langle n_1 \rangle_{o_1} + n_2 \rightarrow_o n_1 + n_2 \text{ (E-ADDFRN}_1\text{)} \\
\\
n_1 + \langle n_2 \rangle_{o_1} \rightarrow_o n_1 + n_2 \text{ (E-ADDFRN}_2\text{)} \\
\\
\frac{n \neq 0}{\text{if } \langle n \rangle_{o_1} \text{ then } t_2 \text{ else } t_3 \rightarrow_o t_2} \text{ (E-IfTrueFRN)} \\
\\
\frac{n = 0}{\text{if } \langle n \rangle_{o_1} \text{ then } t_2 \text{ else } t_3 \rightarrow_o t_3} \text{ (E-IfFalseFRN)}
\end{array}$$

B Evaluation

The full performance results of the evaluation are in Table 2 (for running times) and in Table 3 (for compile times). As

shown in Table 2, the 95% confidence interval for the difference is always narrow and always includes zero. After

Crate	Base (ms)	Δ (ms)	Δ (%)	95% CI (ms)	p
rustix	392.52	-2.52	-0.64	[-9.39, +4.35]	0.46
walkdir	62.64	+0.00	+0.01	[-0.18, +0.19]	0.97
glob	84.93	+0.12	+0.15	[-0.28, +0.53]	0.53
same-file	47.73	+0.06	+0.12	[-0.03, +0.14]	0.17
tempfile	70.01	+0.10	+0.15	[-0.14, +0.35]	0.39
pkg-config-rs	99.78	-0.01	-0.01	[-0.13, +0.11]	0.86
autocfg	1772.73	-0.42	-0.02	[-3.83, +3.00]	0.80
normpath	60.28	+0.07	+0.12	[-0.01, +0.16]	0.09
remove_dir_all	71.10	+0.15	+0.21	[-0.01, +0.31]	0.07
vcpkg-rs	93.70	-0.09	-0.10	[-0.29, +0.11]	0.36

Table 2. Test suite runtime before and after retrofitting. Each row summarizes thirty samples, where a single sample is the average execution time of the crate test suite. Because execution times are very short, we run the test suite in batches.

Crate	rustc (ms)	cnbt (ms)	Δ (ms)	Δ (%)	95% CI (%)
rustix	1182.4	1241.3	+58.8	+5.07	[+3.64, +6.50]
walkdir	316.4	329.1	+12.7	+4.02	[+3.87, +4.17]
glob	304.3	314.4	+10.1	+3.32	[+3.15, +3.48]
same-file	152.0	157.7	+5.7	+3.74	[+3.53, +3.96]
tempfile	1668.8	1731.1	+62.3	+3.73	[+2.97, +4.50]
pkg-config-rs	502.2	518.2	+16.1	+3.20	[+3.09, +3.31]
autocfg	392.1	403.7	+11.6	+2.96	[+2.83, +3.09]
normpath	176.5	183.2	+6.7	+3.81	[+3.62, +3.99]
remove_dir_all	1294.8	1340.0	+45.2	+3.51	[+2.90, +4.12]
vcpkg-rs	999.7	1025.8	+26.2	+2.62	[+2.49, +2.74]

Table 3. Build time of each (modified) crate under rustc alone versus under Coenobita (cnbt). Each row displays results from thirty samples, where a single sample is the build time of the (modified) crate across several uncached builds (enough to make the cumulative time over 10 seconds). Same as before, we build the crate in batches and report the average for a single build.

conducting a paired two-tailed t -test with null hypothesis $H_0 : a = a'$ (where a represents the runtime before transformations and a' represents the runtime after), we fail to reject the null hypothesis (all p -values > 0.05). We conclude that the changes required to comply with our tool do not incur any measurable runtime performance overhead on these crates. As shown in Table 3, the 95% confidence intervals for the compile time overhead are in the range 2.5%–6.5% on these crates (2.5%–4.5% with the exception of rustix). The average overhead for these 10 crates is 3.6%.