

zBulk

Compartmentalization of Rust Applications with Zero Retrofitting

PLOS'26 · Session 3: Security and Safety · Prague, Czech Republic · September 29, 2026

Maxim Ritter von Onciul · Phillip Raffeck · Peter Wägemann · Rüdiger Kapitza

Friedrich-Alexander-Universität Erlangen-Nürnberg (Germany)



Friedrich-Alexander-Universität
Technische Fakultät



Lehrstuhl für Informatik 4
Systemsoftware

zBulk

Compartmentalization of Rust Applications with Zero Retrofitting

PLOS'26 · Session 3: Security and Safety · Prague, Czech Republic · September 29, 2026

Maxim Ritter von Onciul · Phillip Raffeck · Peter Wägemann · Rüdiger Kapitza

Friedrich-Alexander-Universität Erlangen-Nürnberg (Germany)

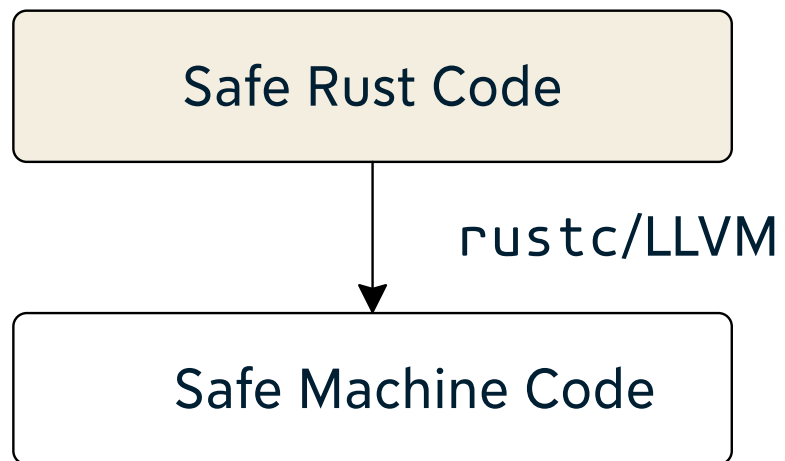


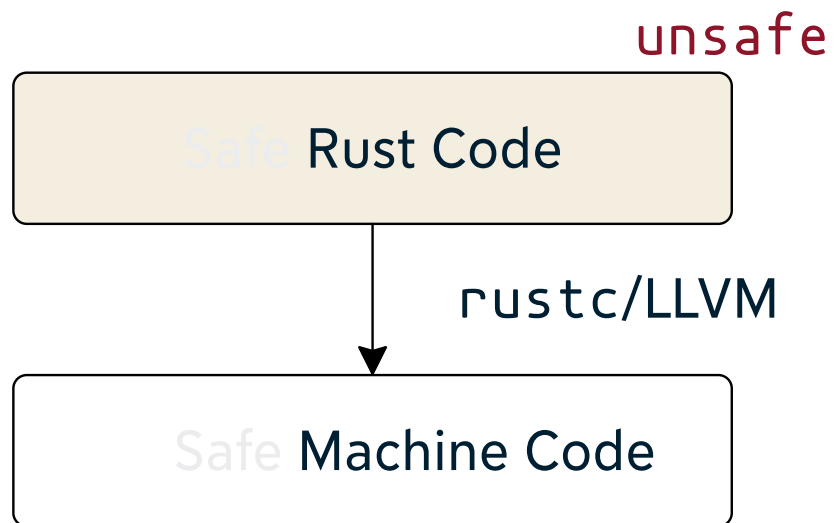
Friedrich-Alexander-Universität
Technische Fakultät



Lehrstuhl für Informatik 4
Systemsoftware

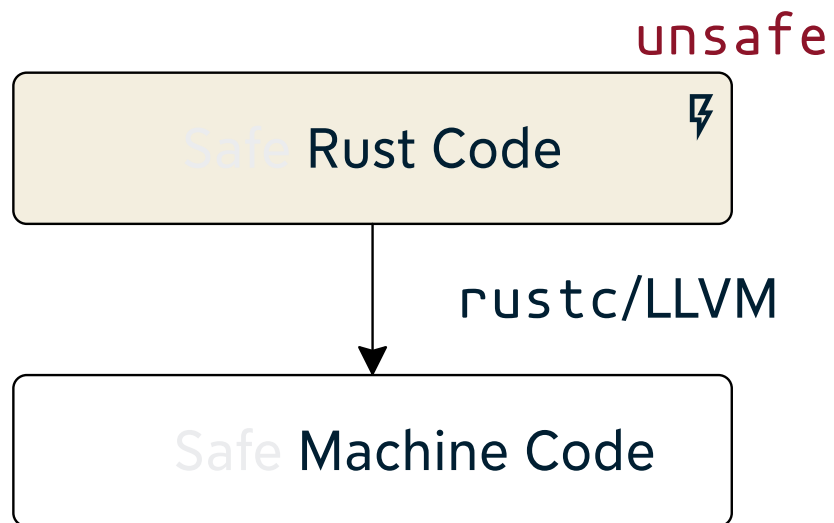
Memory Safety in Rust





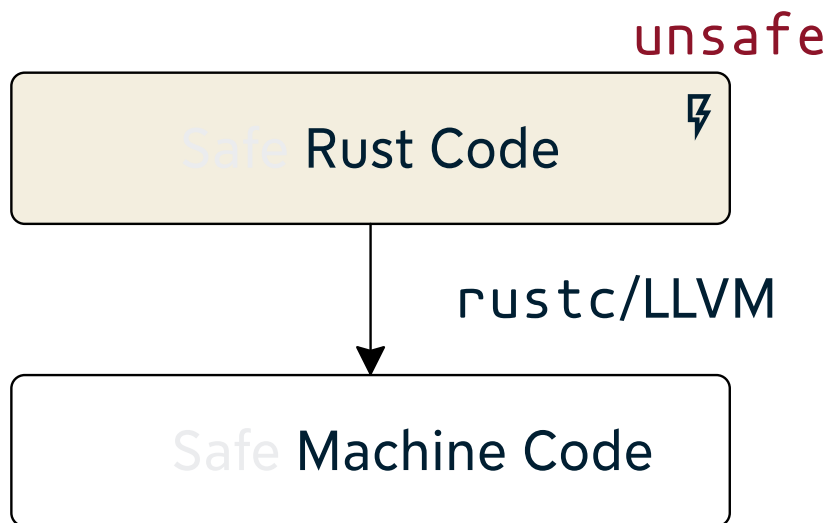
unsafe

- Direct Hardware Access
- Call Foreign Functions



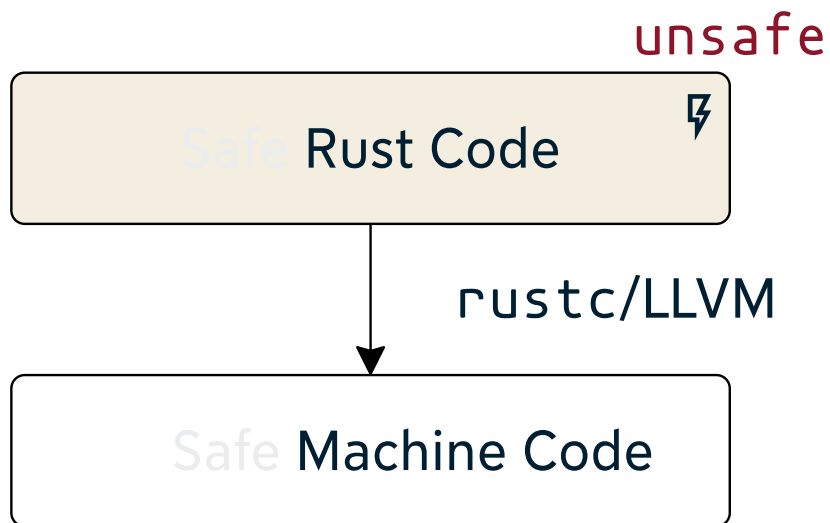
Actix Web Framework v0.7.19

```
1 let mut buffer = /* */;  
2  
3  
4  
5
```



Actix Web Framework v0.7.19

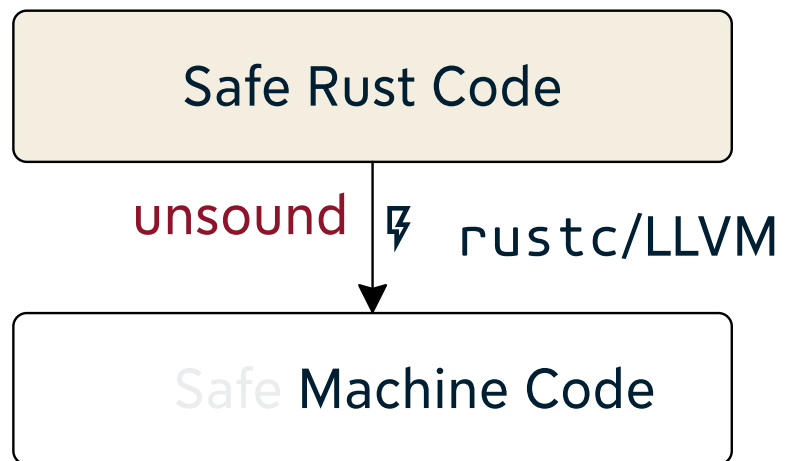
```
1 let mut buffer = /* */;  
2 let mut buf = unsafe {  
3   /* &*buffer */ };  
4  
5
```



Actix Web Framework v0.7.19

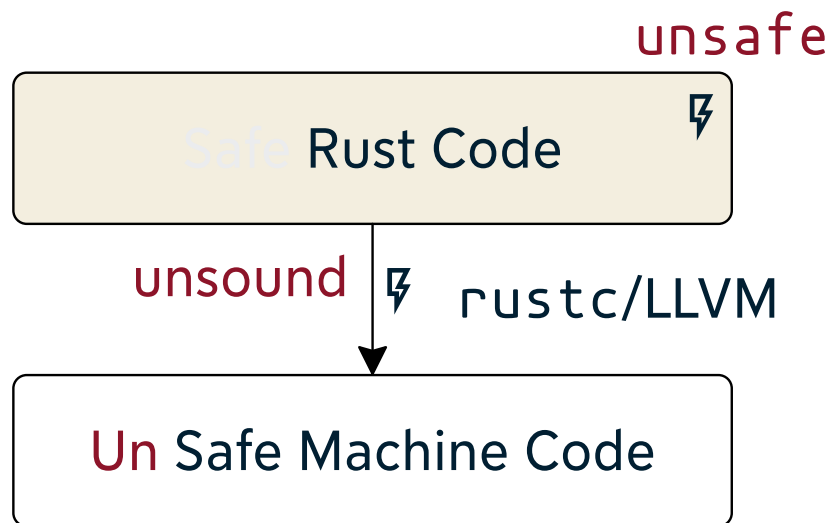
```
1 let mut buffer = /* */;  
2 let mut buf = unsafe {  
3   /* &*buffer */ };  
4 /* buf may turn dangling */  
5 buffer.reserve(len);
```

RUSTSEC-2018-0019 /
CVE-2018-25025

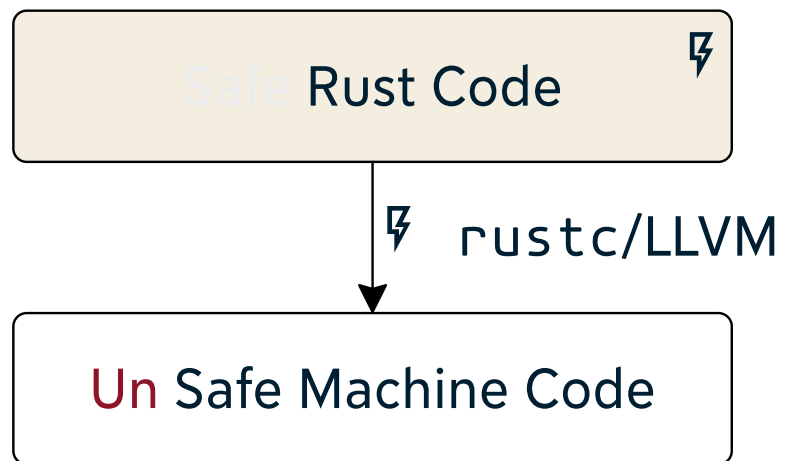


- Can break safety guarantees
- 135 open issues on soundness

Threats to Memory Safety in Rust

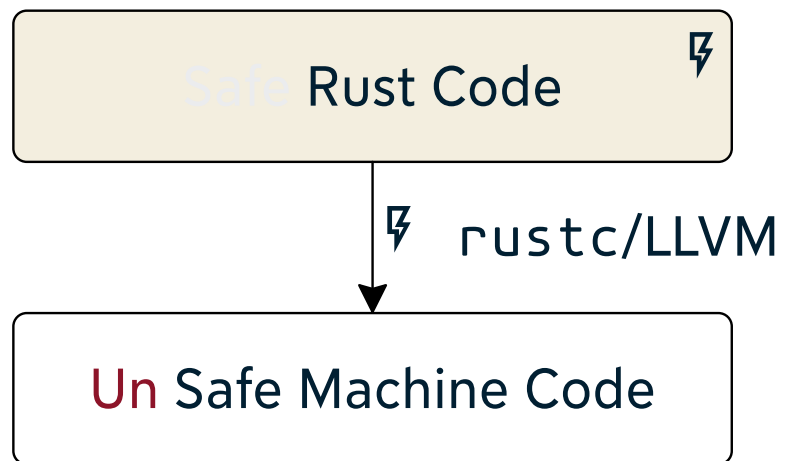


Threats to Memory Safety in Rust

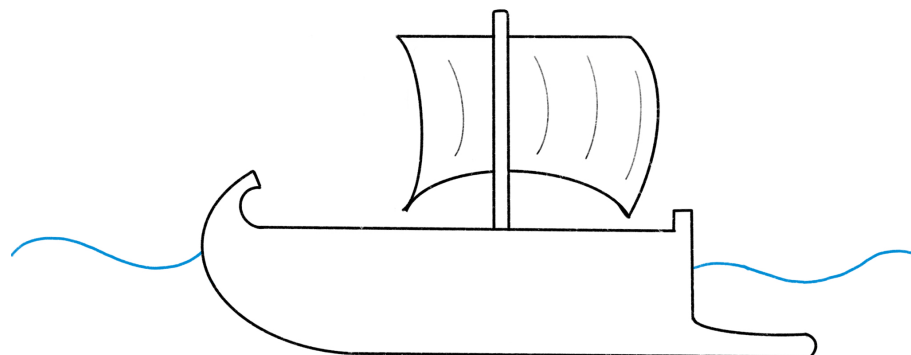


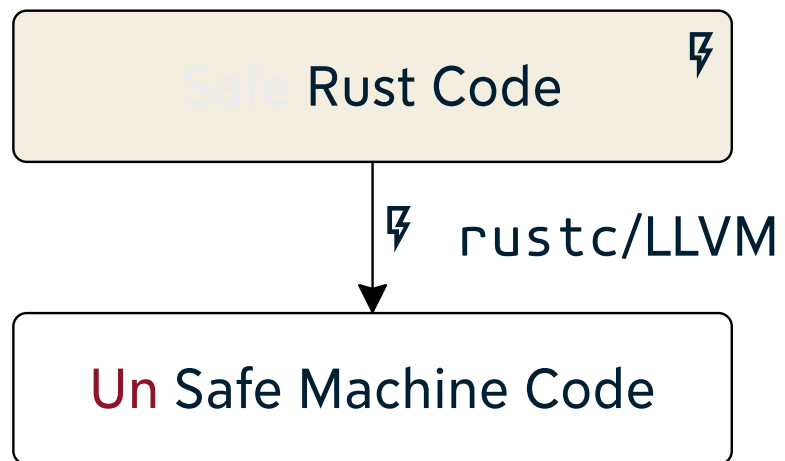
Threatened Security Goals

- Data Confidentiality
- Control-Flow Integrity

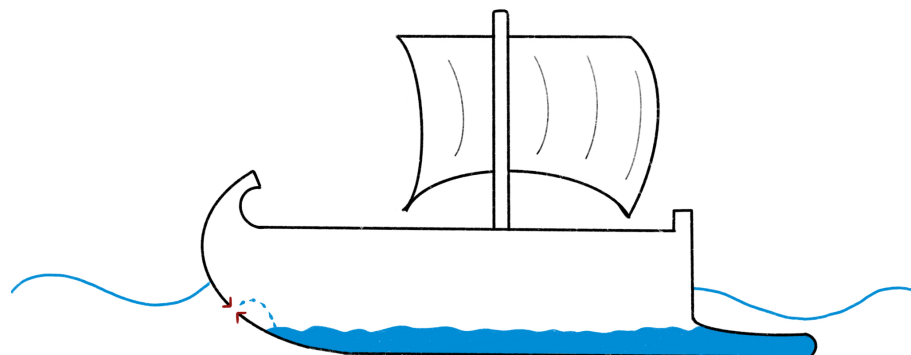


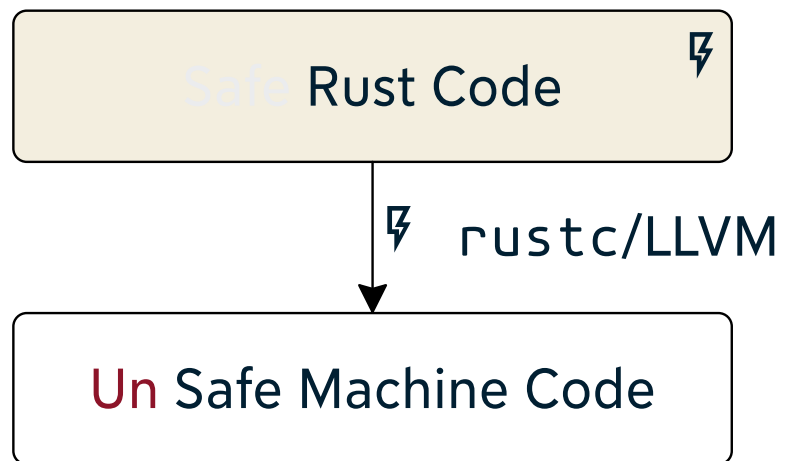
Compartmentalization



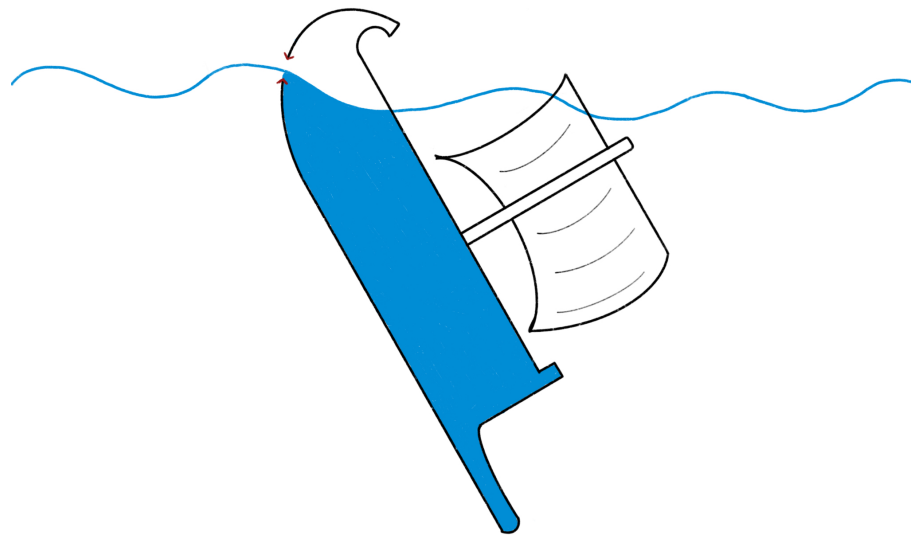


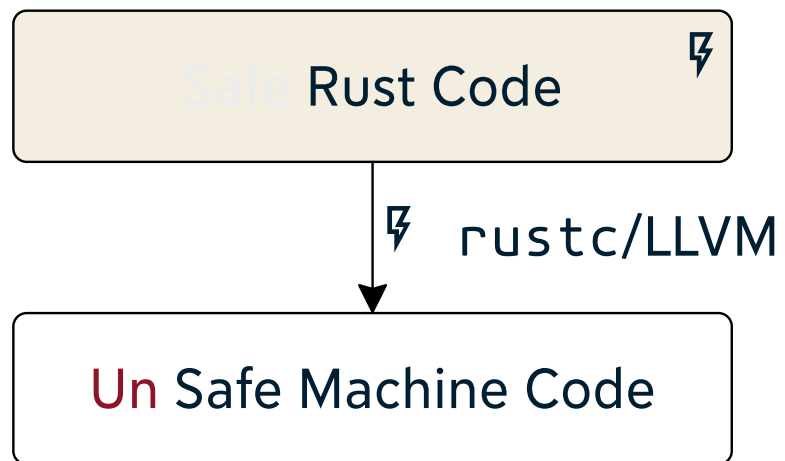
Compartmentalization



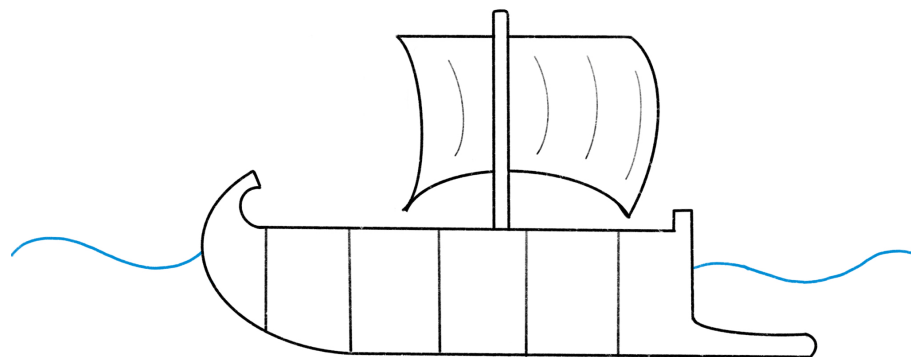


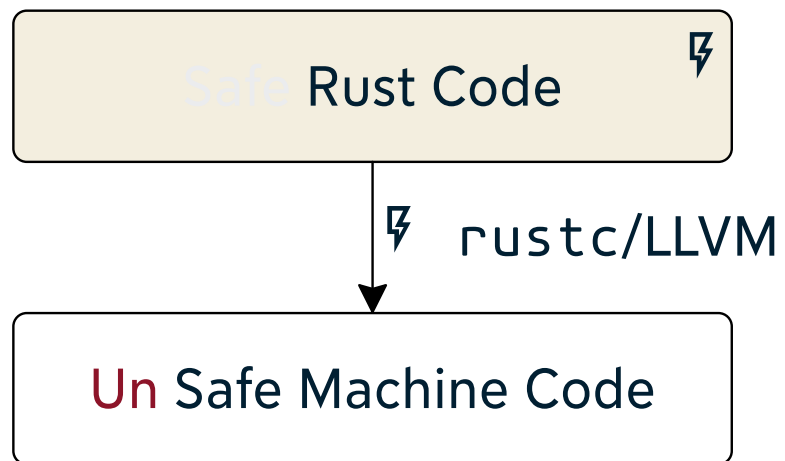
Compartmentalization



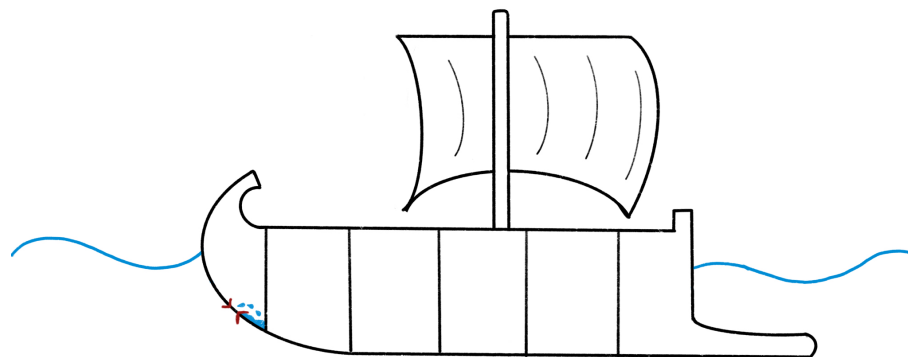


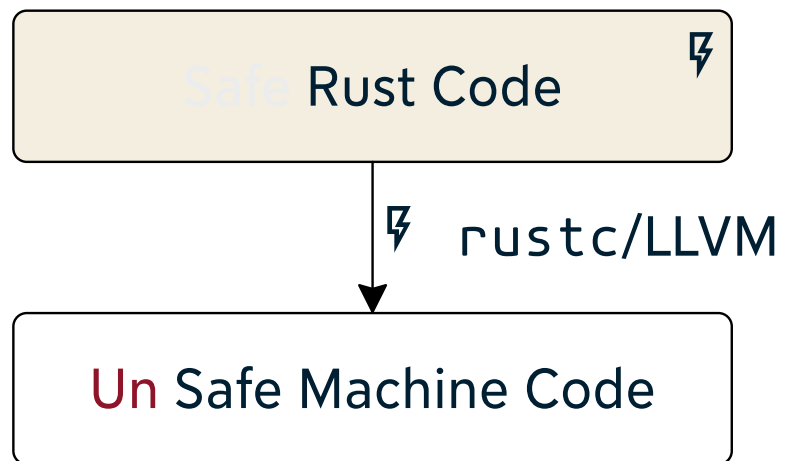
Compartmentalization



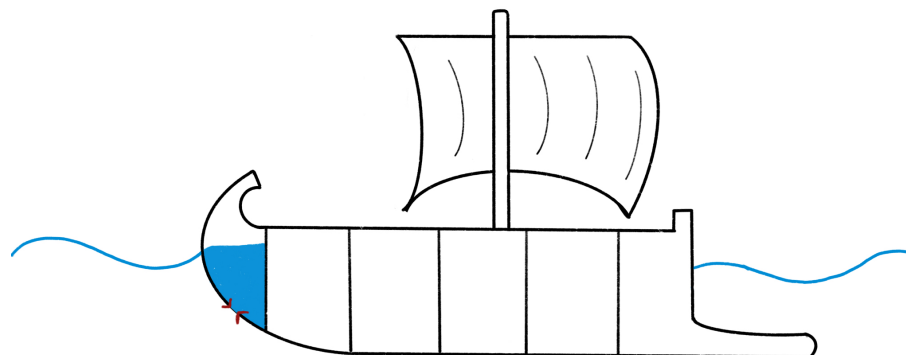


Compartmentalization





Compartmentalization



Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric

```
mod a {  
    struct SA { .. }  
    fn f1()  
    fn f2()  
  
    mod b {  
        fn f3()  
        fn f4()  
    }  
}
```

Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate

```
mod a {  
    struct SA { .. }  
    fn f1()  
    fn f2()  
  
    mod b {  
        fn f3()  
        fn f4()  
    }  
}
```

Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

```
mod a {  
    struct SA { .. }  
    fn f1()  
    fn f2()  
  
    mod b {  
        fn f3()  
        fn f4()  
    }  
}
```

Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

Problems in Practice

- Compartmentalization bloats code

```
mod a {  
    struct SA { .. }  
    fn f1()  
    fn f2()  
  
    mod b {  
        fn f3()  
        fn f4()  
    }  
}
```

Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

Problems in Practice

- Compartmentalization bloats code
 1. Partitioning

```
mod a {  
  struct SA { .. }  
  fn f1()  
  fn f2()  
}  
  
mod b {  
  fn f3()  
  fn f4()  
}
```

Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

Problems in Practice

- Compartmentalization bloats code
 1. Partitioning
 2. Isolation mechanism

```
mod a {  
  struct SA { .. }  
  fn f1() switch to B  
  fn f2() switch to B  
}  
  
mod b {  
  fn f3()  
  fn f4() switch to A  
}  
}
```

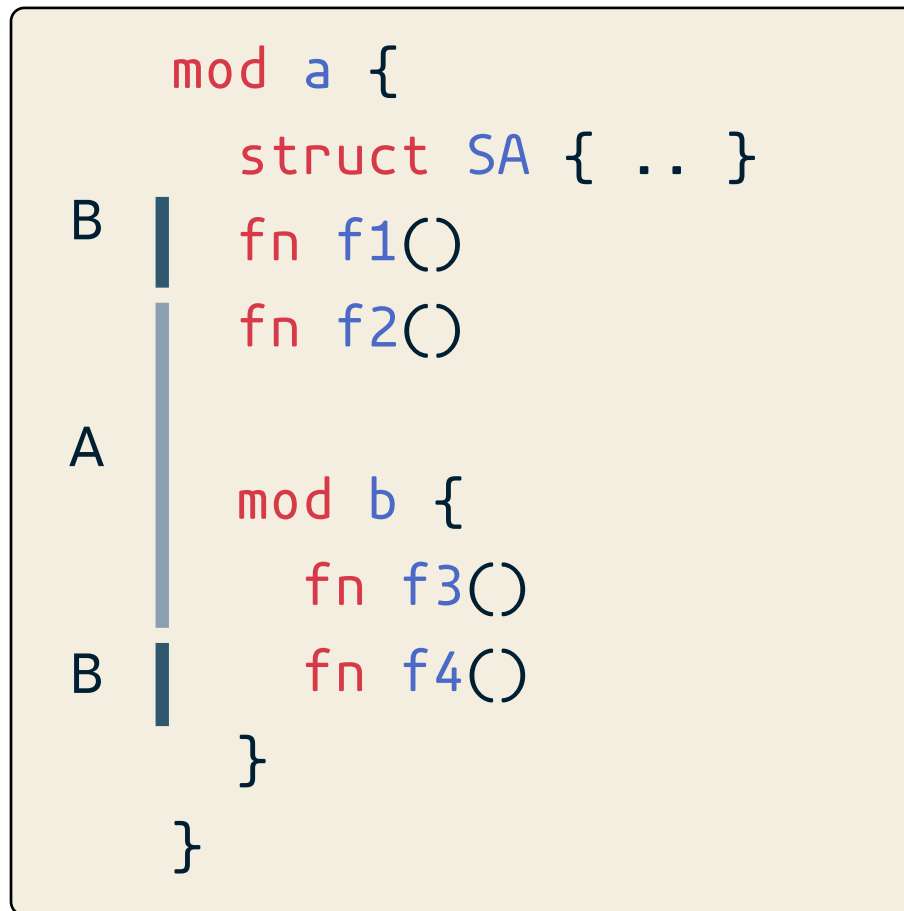
Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

Problems in Practice

- Compartmentalization bloats code
 1. Partitioning
 2. Isolation mechanism



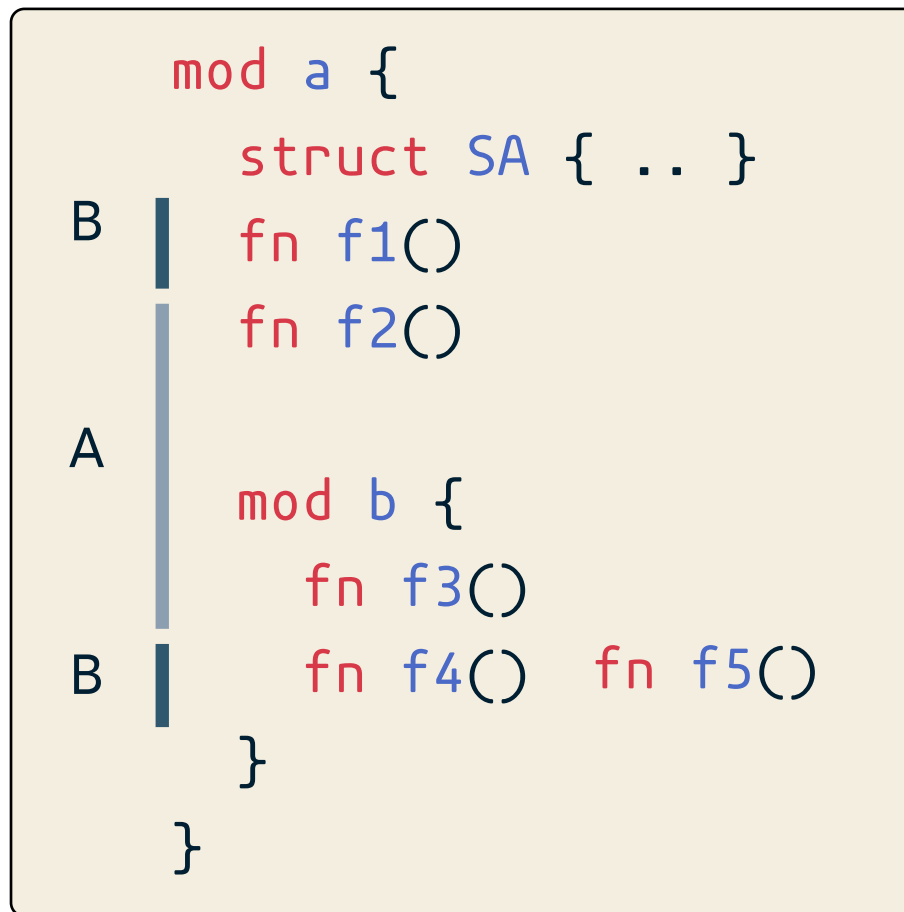
Problems with Manual Compartmentalization

Where the Analogy Breaks

- Code is not geometric
- No continuous unit to separate
- How to draw boundaries?

Problems in Practice

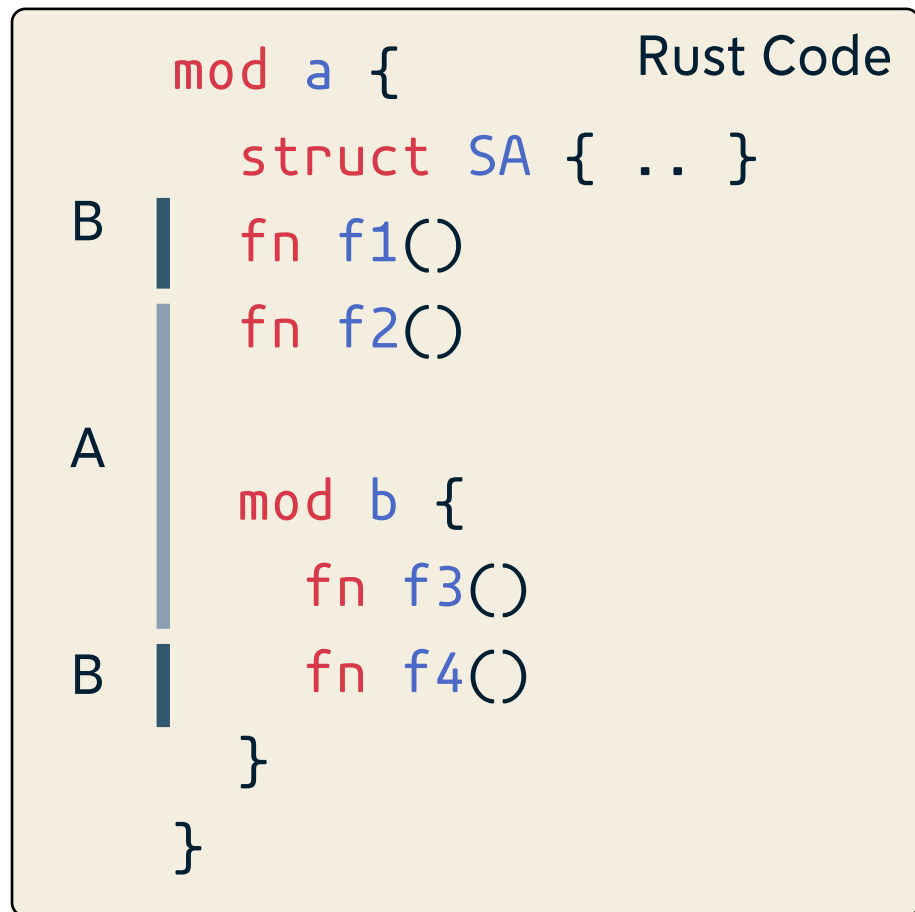
- Compartmentalization bloats code
 1. Partitioning
 2. Isolation mechanism



zBulk

**Source-To-Source Compiler to Make
Compartmentalization a
Build-Time Decision**

Make Compartmentalization a Build-Time Decision



Make Compartmentalization a Build-Time Decision

Rust Code

```
mod a {  
    struct SA { .. }  
    fn f1()  
    fn f2()  
    mod b {  
        fn f3()  
        fn f4()  
    }  
}
```

The diagram illustrates the compartmentalization of Rust code. A vertical bar on the left is divided into three sections: a top section labeled 'B' (dark blue), a middle section labeled 'A' (light blue), and a bottom section labeled 'B' (dark blue). The code is structured to match these compartments: the top 'B' section contains the first two function definitions (`fn f1()` and `fn f2()`); the 'A' section contains the definition of module `b`; and the bottom 'B' section contains the function definitions inside module `b` (`fn f3()` and `fn f4()`).

Placement Rule

Compartment A:

- `f2`
- `f3`

Compartment B:

- `f1`
- `f4`

Make Compartmentalization a Build-Time Decision

```
mod a {                                Rust Code
    struct SA { .. }
    fn f1()
    fn f2()

    mod b {
        fn f3()
        fn f4()
    }
}
```

Placement Rule

Compartment A:

- f2
- f3

Compartment B:

- f1
- f4

Make Compartmentalization a Build-Time Decision

```
mod a {                               Rust Code
    struct SA { .. }
    fn f1()
    fn f2()

    mod b {
        fn f3()
        fn f4()
    }
}
```

How?

Placement Rule

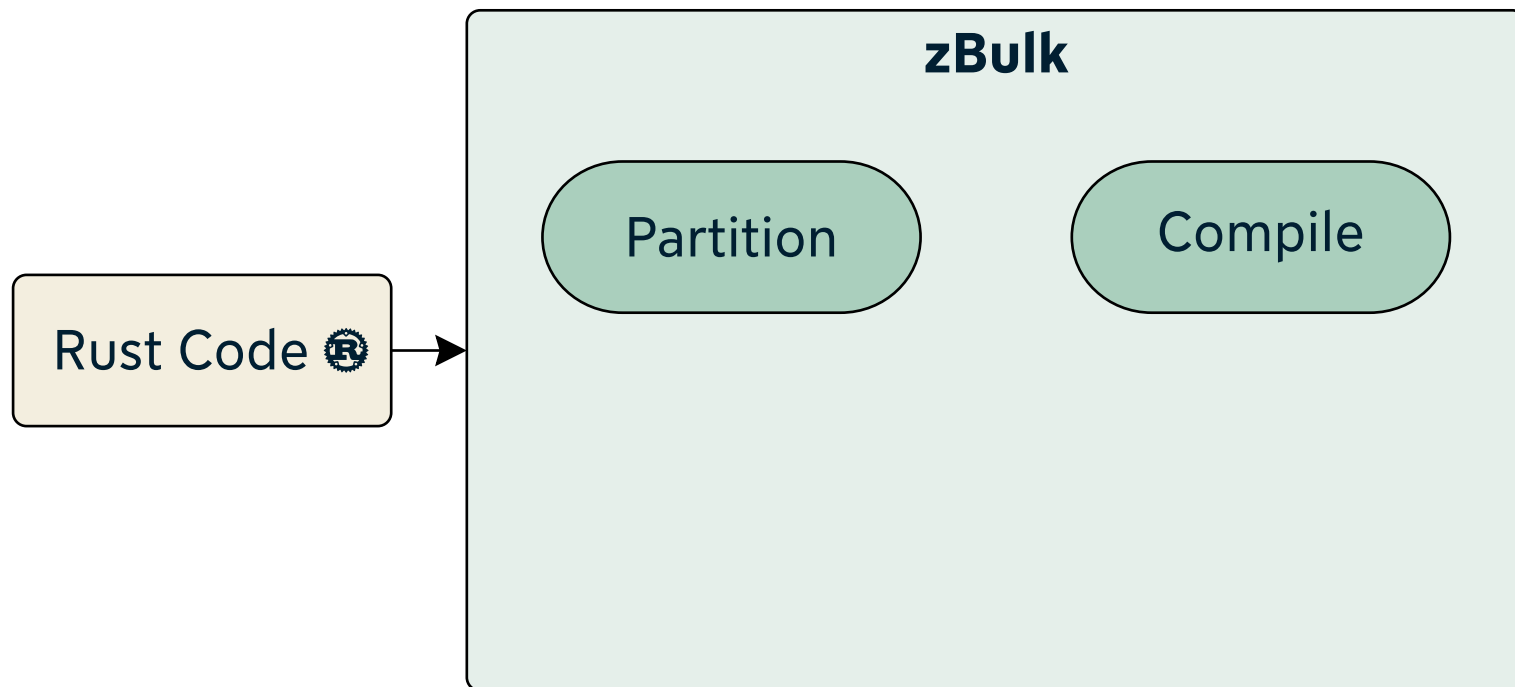
Compartment A:

- f2
- f3

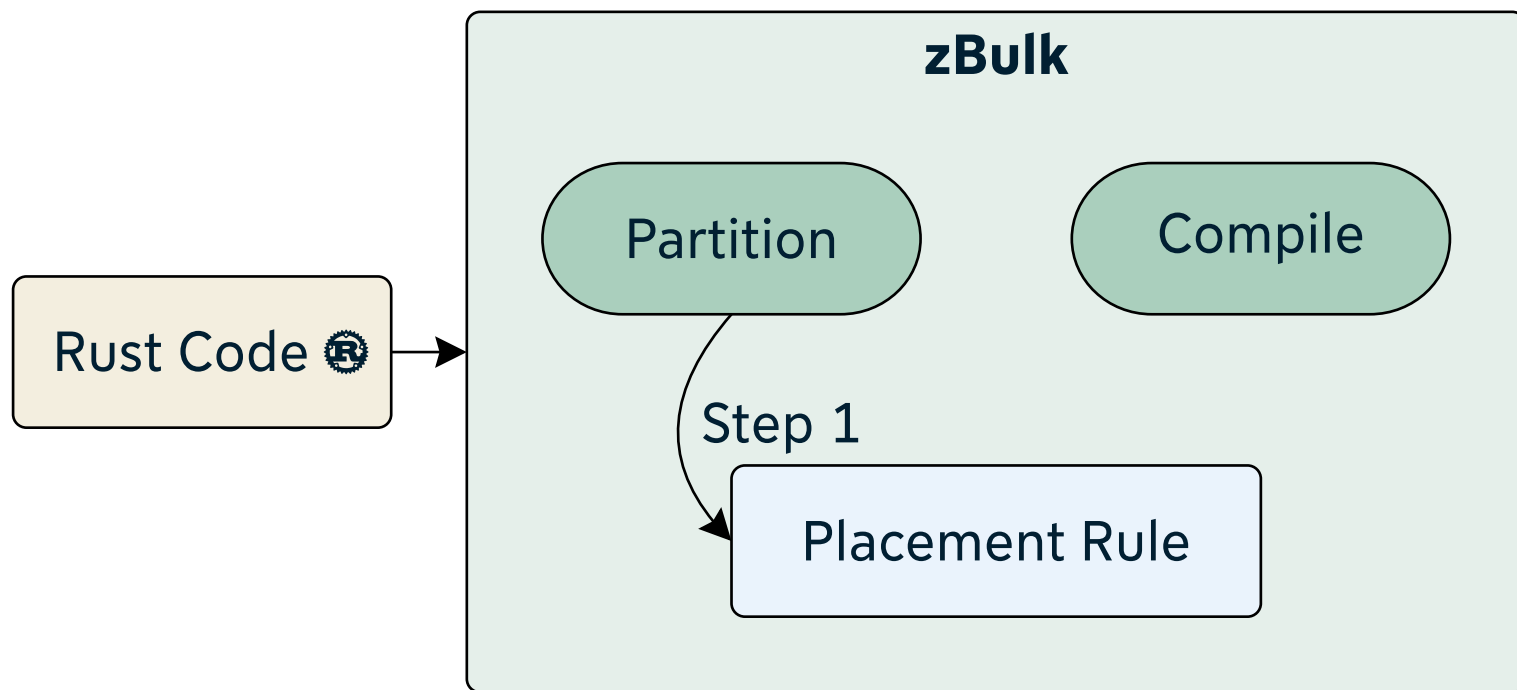
Compartment B:

- f1
- f4

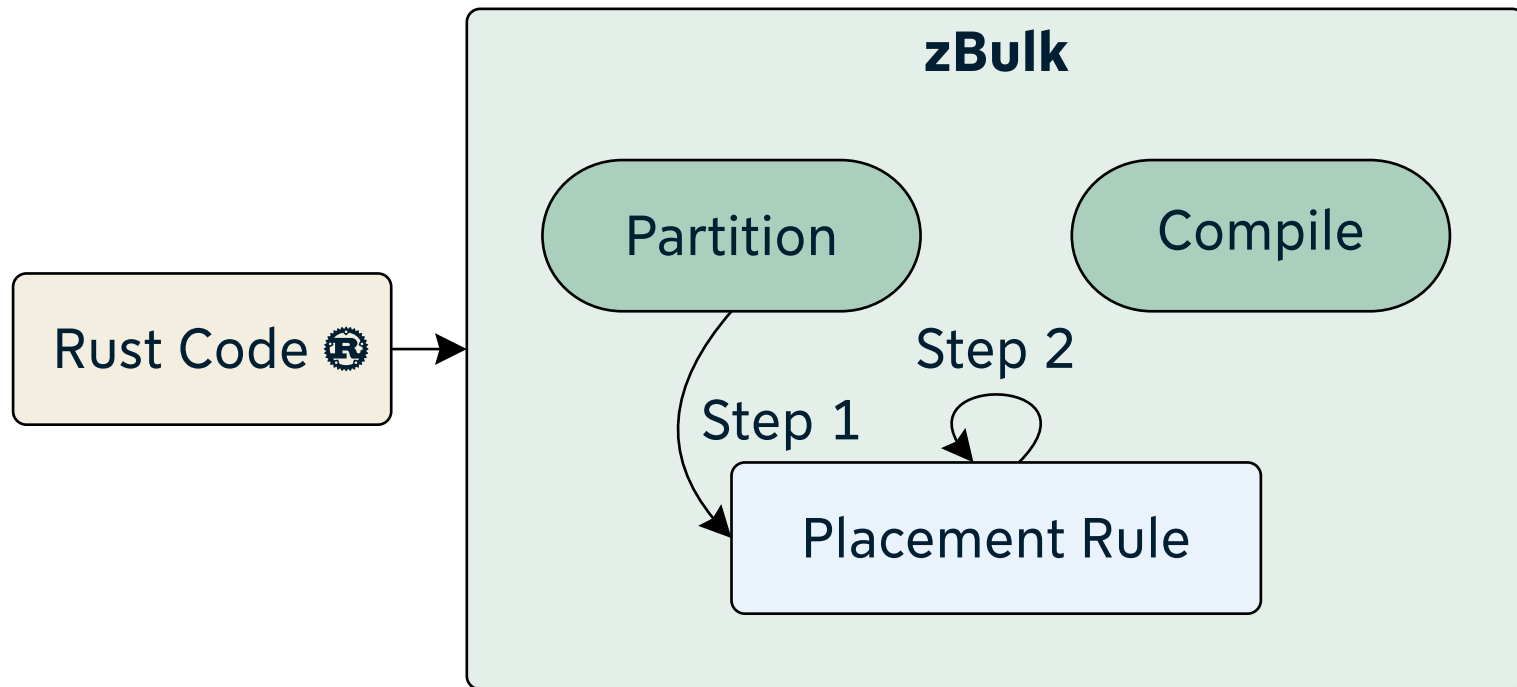
zBulk: Source-To-Source Rust Compiler



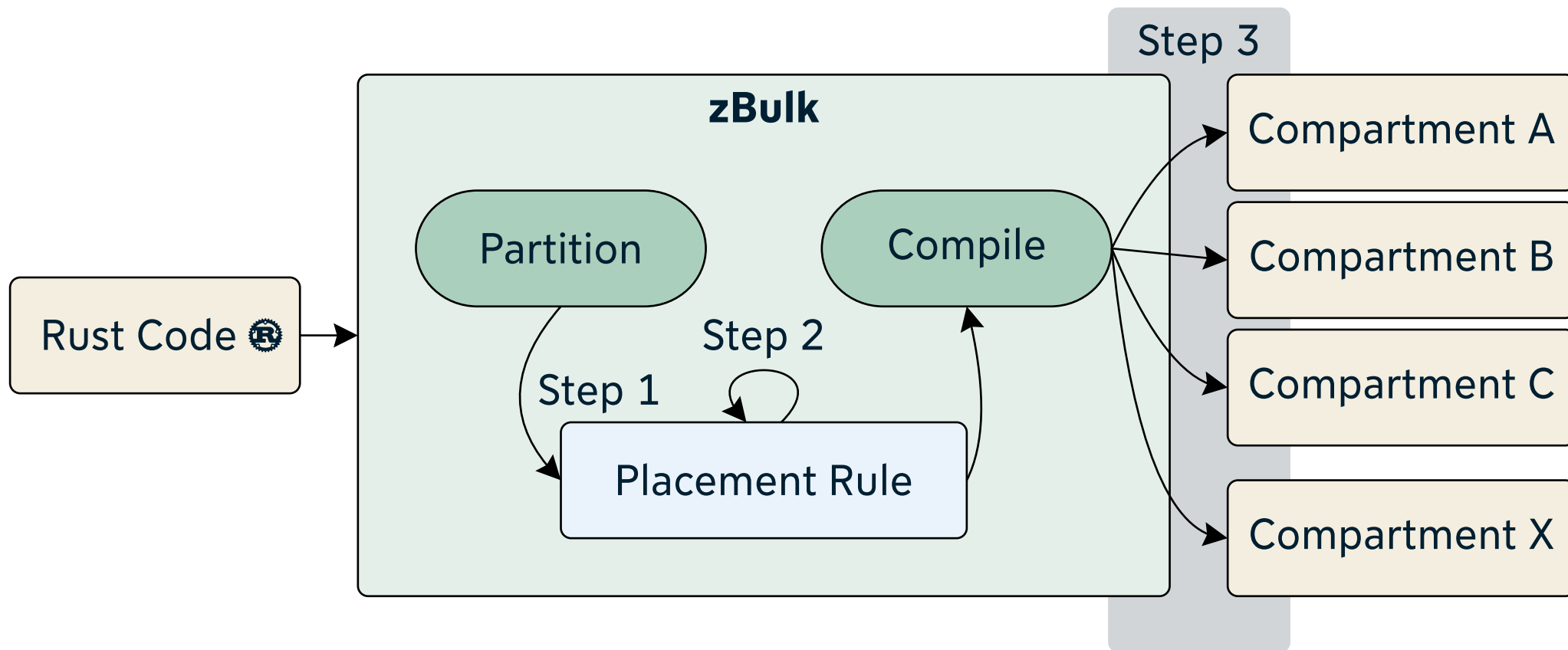
zBulk: Source-To-Source Rust Compiler



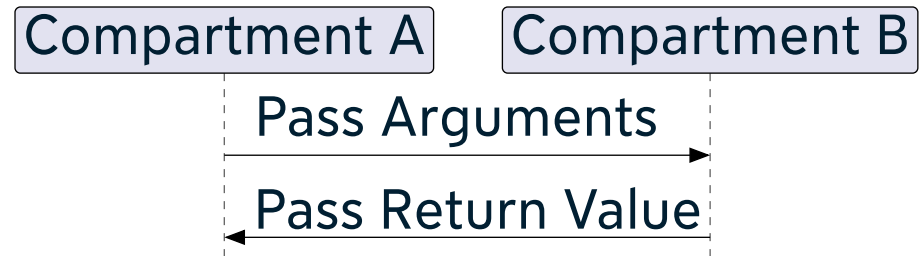
zBulk: Source-To-Source Rust Compiler



zBulk: Source-To-Source Rust Compiler



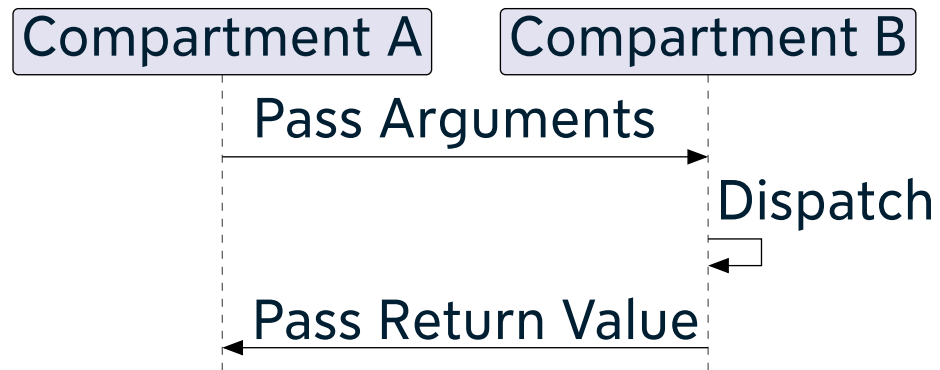
C1: Automatic Instrumentation



Research Challenges

C1: Automatic Instrumentation

C2: Compartment Entry Dispatch



C1: Automatic Instrumentation

C2: Compartment Entry Dispatch

C3: Performance-Security Tradeoff

Overhead

- What is the overhead?
- Where does it come from?
- What could we do to improve?

What **zBulk** Builds Upon

Compartment A

Unmodified foo

Compartment B

Unmodified bar

What **zBulk** Builds Upon

Compartment A

Unmodified foo



Stub

serialize args 

encode "bar" 

Compartment B

Unmodified bar

What **zBulk** Builds Upon

Compartment A

Unmodified foo



Stub

serialize args 

encode "bar" 



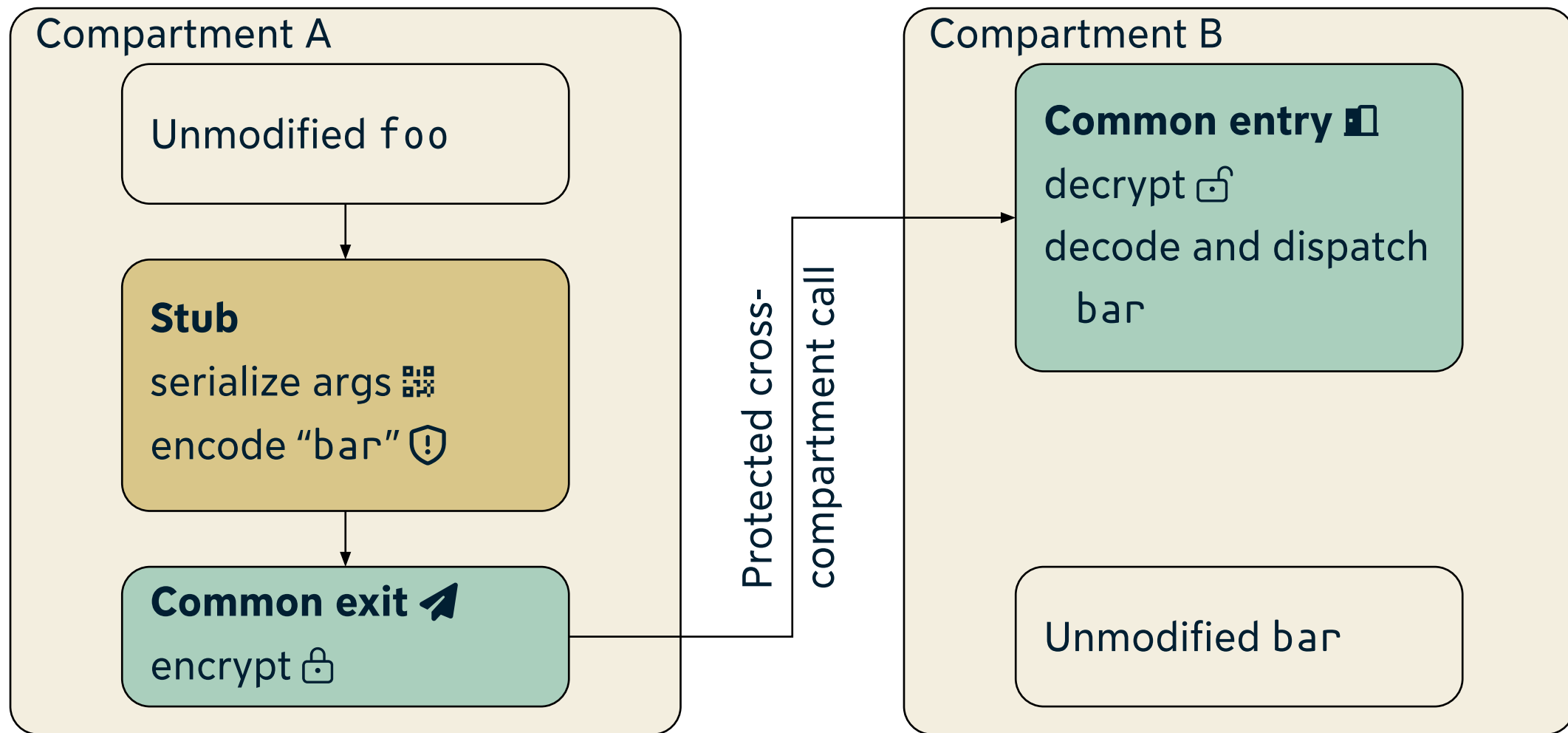
Common exit 

encrypt 

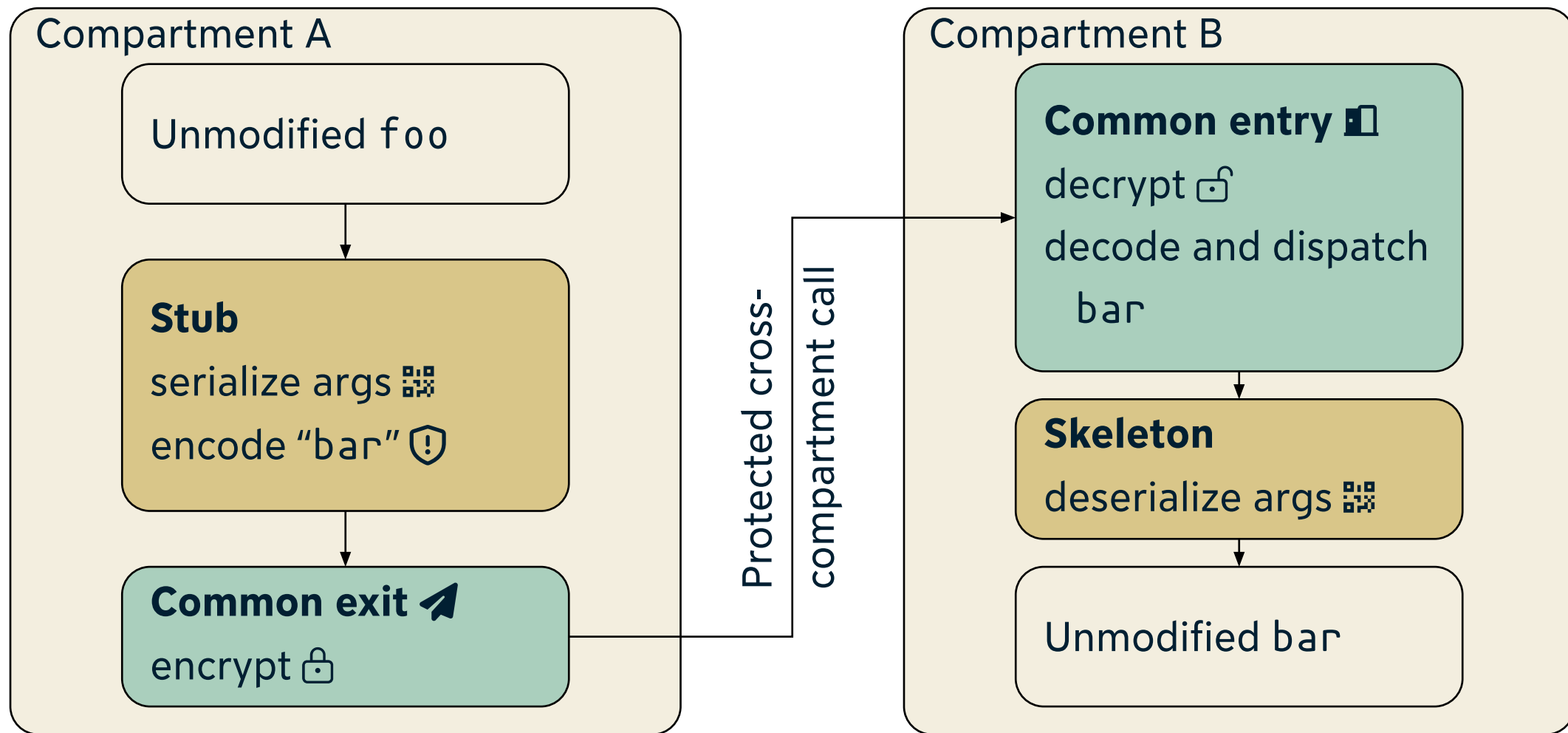
Compartment B

Unmodified bar

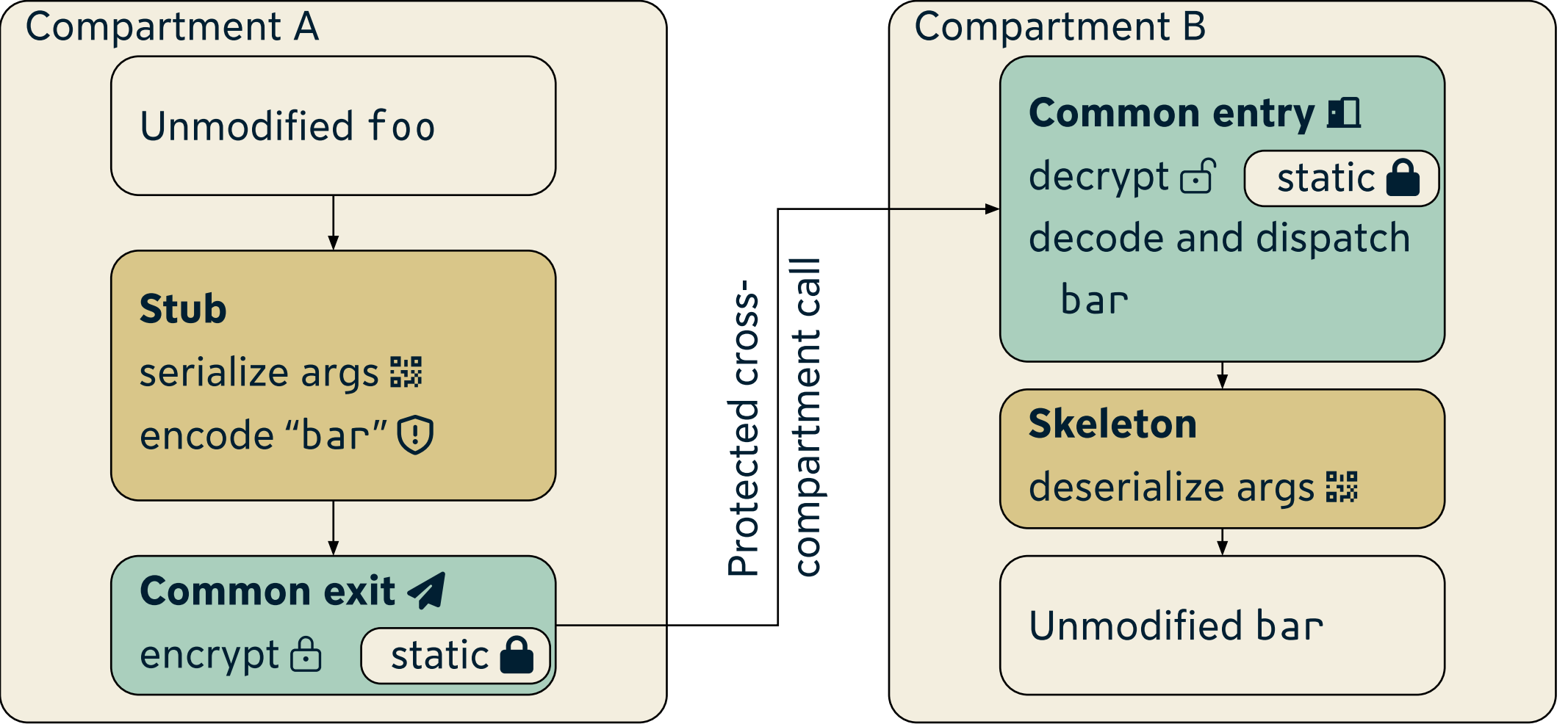
What **zBulk** Builds Upon

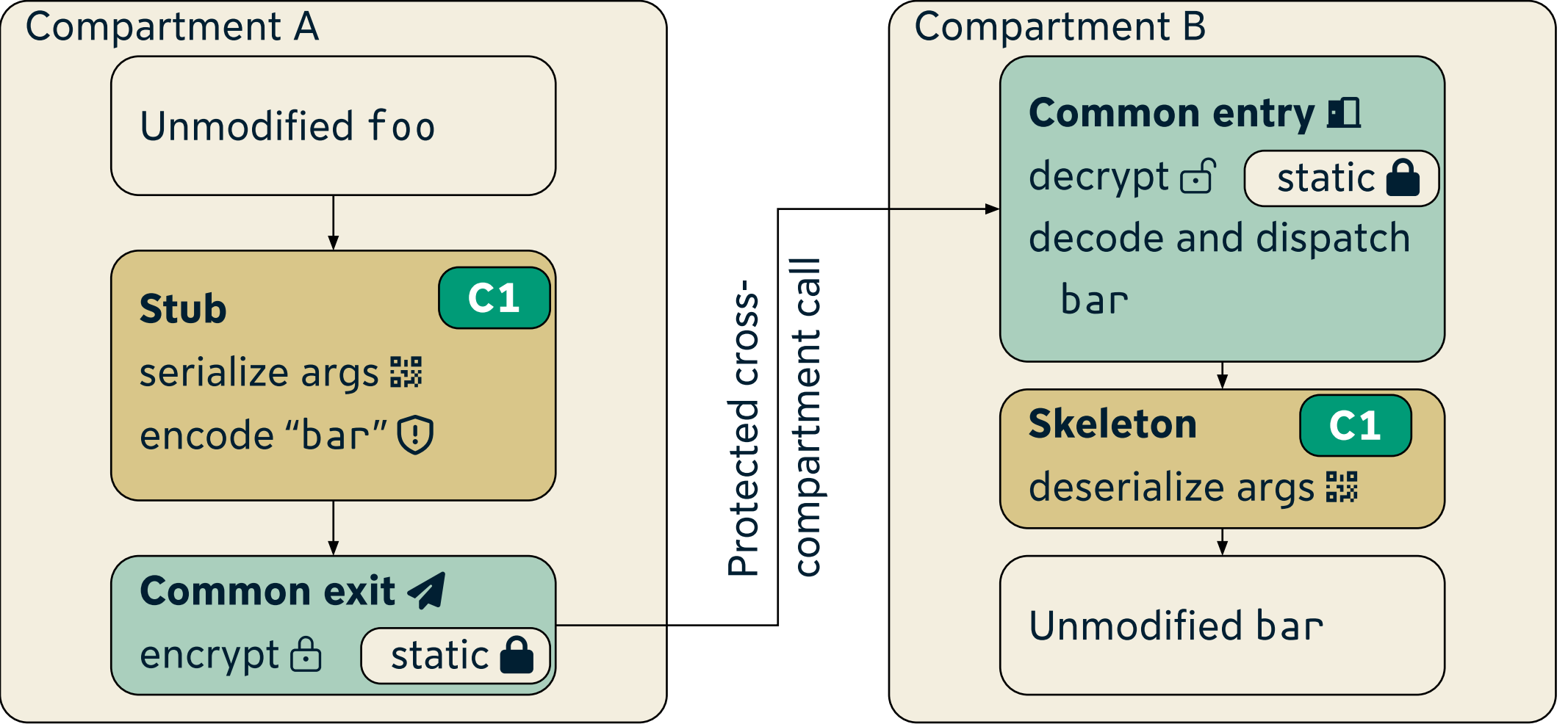


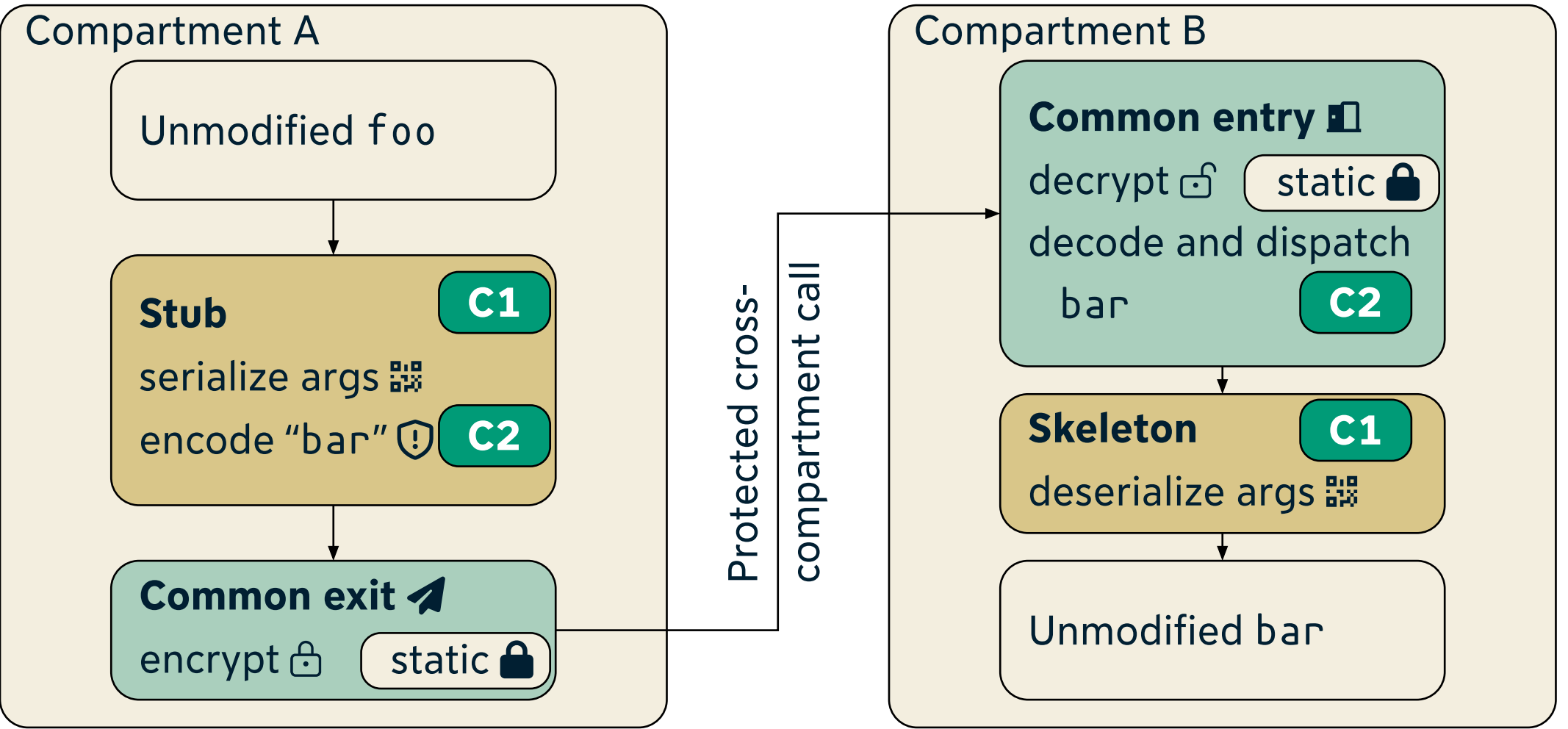
What **zBulk** Builds Upon



What zBulk Builds Upon







C1: Automatic Instrumentation

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3
4
5
6
7
8
9
10
11
```

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3 // stub
4 fn bar(arg0: T0, arg1: T1) -> T2 {
5     let mut args = /* allocate memory */;
6
7
8
9
10
11
```

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3 // stub
4 fn bar(arg0: T0, arg1: T1) -> T2 {
5     let mut args = /* allocate memory */;
6     // for each arg
7     args.append(serialize(arg0));
8     args.append(serialize(arg1));
9
10
11
```

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3 // stub
4 fn bar(arg0: T0, arg1: T1) -> T2 {
5     let mut args = /* allocate memory */;
6     // for each arg
7     args.append(serialize(arg0));
8     args.append(serialize(arg1));
9     let result = cross_call(/* solve identification */, args);
10
11
```

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3 // stub
4 fn bar(arg0: T0, arg1: T1) -> T2 {
5     let mut args = /* allocate memory */;
6     // for each arg
7     args.append(serialize(arg0));
8     args.append(serialize(arg1));
9     let result = cross_call(/* solve identification */, args);
10    deserialize::<T2>(result)
11 }
```

```
1 fn foo() { let x = bar(arg0, arg1); }
2 fn bar(arg0: T0, arg1: T1) -> T2;
3 // stub
4 fn bar(arg0: T0, arg1: T1) -> T2 {
5     let mut args = /* allocate memory */;
6     // for each arg
7     args.append(serialize(arg0));
8     args.append(serialize(arg1));
9     let result = cross_call(/* solve identification */, args);
10    deserialize::<T2>(result)
11 }
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {  
2     generic_member: T, // transform <T: Serialize + Deserialize>  
3  
4  
5  
6  
7  
8  
9  
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {  
2     generic_member: T, // transform <T: Serialize + Deserialize>  
3     anonymous_type: impl TraitStatic, // similar  
4  
5  
6  
7  
8  
9  
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {  
2     generic_member: T, // transform <T: Serialize + Deserialize>  
3     anonymous_type: impl TraitStatic, // similar  
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?  
5 }  
6  
7  
8  
9  
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {  
2     generic_member: T, // transform <T: Serialize + Deserialize>  
3     anonymous_type: impl TraitStatic, // similar  
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?  
5 }  
6 extern "C" { fn call_foreign(); } // supported  
7  
8  
9  
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {  
2     generic_member: T, // transform <T: Serialize + Deserialize>  
3     anonymous_type: impl TraitStatic, // similar  
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?  
5 }  
6 extern "C" { fn call_foreign(); } // supported  
7 fn references<'a, T1, T2, T3, T4>(  
8  
9  
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {
2     generic_member: T, // transform <T: Serialize + Deserialize>
3     anonymous_type: impl TraitStatic, // similar
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?
5 }
6 extern "C" { fn call_foreign(); } // supported
7 fn references<'a, T1, T2, T3, T4>(
8     arg0: &T1, arg1: &mut T2, // supported
9
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {
2     generic_member: T, // transform <T: Serialize + Deserialize>
3     anonymous_type: impl TraitStatic, // similar
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?
5 }
6 extern "C" { fn call_foreign(); } // supported
7 fn references<'a, T1, T2, T3, T4>(
8     arg0: &T1, arg1: &mut T2,           // supported
9     arg2: T3<'a>,                       // not supported
10
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {
2     generic_member: T, // transform <T: Serialize + Deserialize>
3     anonymous_type: impl TraitStatic, // similar
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?
5 }
6 extern "C" { fn call_foreign(); } // supported
7 fn references<'a, T1, T2, T3, T4>(
8     arg0: &T1, arg1: &mut T2,           // supported
9     arg2: T3<'a>,                       // not supported
10 ) -> &'a T4           // Research question: is this even possible?
```

Assumptions Made Explicit

```
1 struct ProblemType<T> {
2     generic_member: T, // transform <T: Serialize + Deserialize>
3     anonymous_type: impl TraitStatic, // similar
4     dynamic_dispatch: dyn TraitDyn, // Research question: how?
5 }
6 extern "C" { fn call_foreign(); } // supported
7 fn references<'a, T1, T2, T3, T4>(
8     arg0: &T1, arg1: &mut T2,           // supported
9     arg2: T3<'a>,                       // not supported
10 ) -> &'a T4 // Research question: is this even possible?
```

Instrumentation-Aware Placement Rule

Compartment A: [f1]

Compartment B: [f2]

Compartment C: [f3]

Instrumentation-Aware Placement Rule

Compartment A: [f1]

Compartment B: [f2]

Compartment C: [f3]

Include where required:

- f4 # dynamically dispatches
- f5 # mono instance not serializable
- f6 # ...

Instrumentation-Aware Placement Rule

Compartment A: [f1]

Compartment B: [f2]

Compartment C: [f3]

Include where required:

- f4 # dynamically dispatches
- f5 # mono instance not serializable
- f6 # ...

Research Challenges

C1: Automatic Instrumentation (✓)

C2: Compartment Entry Dispatch

C3: Performance-Security Tradeoff

C2: Compartment Entry Dispatch

C2: How **zBulk** Identifies Functions

1. Hook into rustc:

```
struct Id {  
    item_id: u64,                // from DefId
```

C2: How **zBulk** Identifies Functions

1. Hook into rustc:

```
struct Id {  
    item_id: u64,           // from DefId  
    crate_id: u64,         // from StableCrateId  
}
```

C2: How **zBulk** Identifies Functions

1. Hook into rustc:

```
struct Id {  
    item_id: u64,           // from DefId  
    crate_id: u64,         // from StableCrateId  
    mono_info: MonoInfo,    // from MIR  
}
```

C2: How **zBulk** Identifies Functions

1. Hook into rustc:

```
struct Id {  
    item_id: u64,           // from DefId  
    crate_id: u64,         // from StableCrateId  
    mono_info: MonoInfo,   // from MIR  
}
```

2. Dispatch routine:

```
match id { Id { of_foo } => foo_skel(args), _ => panic!(), }
```

Research Challenges

C1: Automatic Instrumentation (✓)

C2: Compartment Entry Dispatch ✓

C3: Performance-Security Tradeoff

C3: Performance-Security Tradeoff

Benchmark

- representative systems application
- reads from `stdin` and computes
- uses an external crate: `base64`

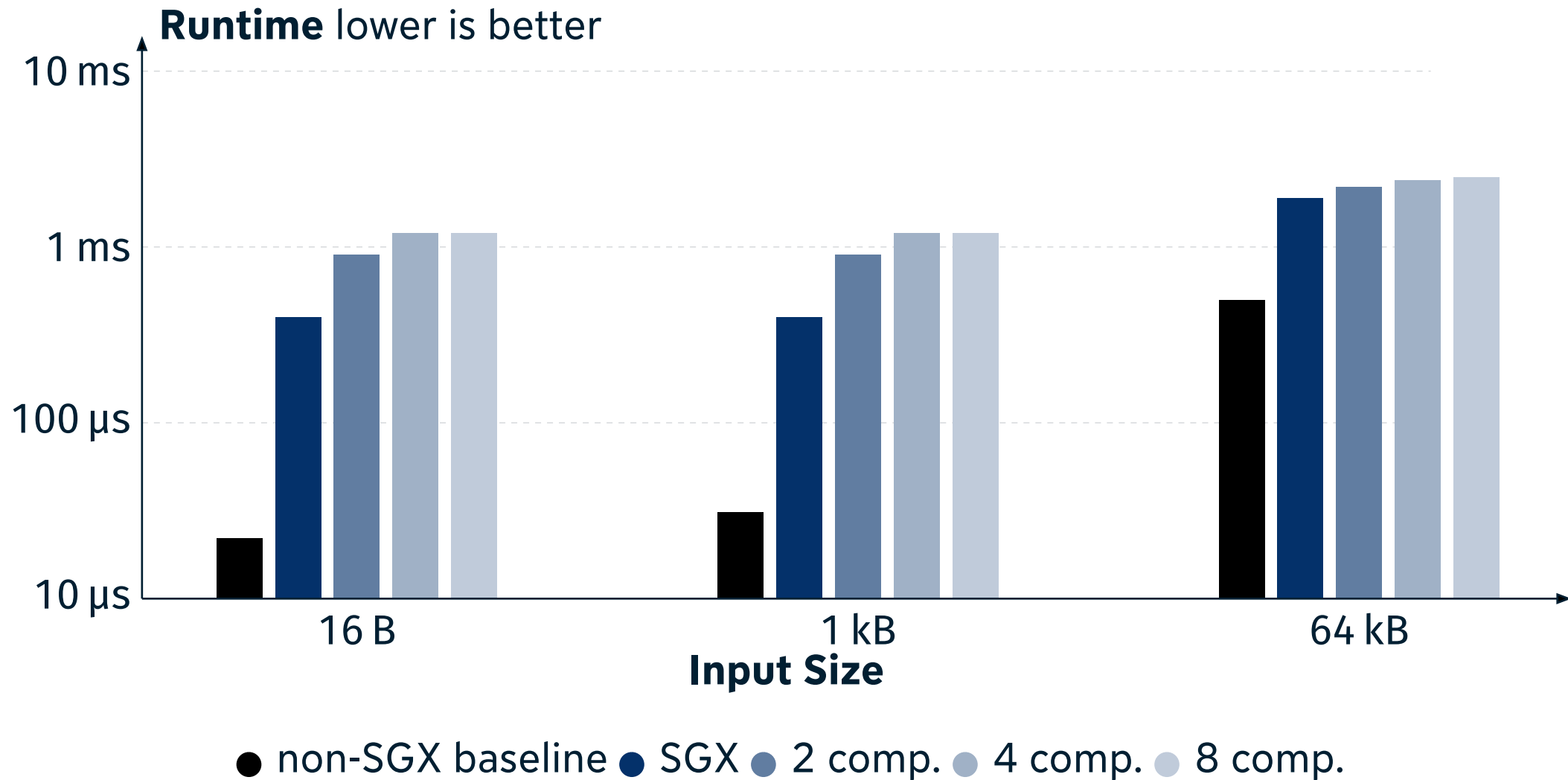
Benchmark

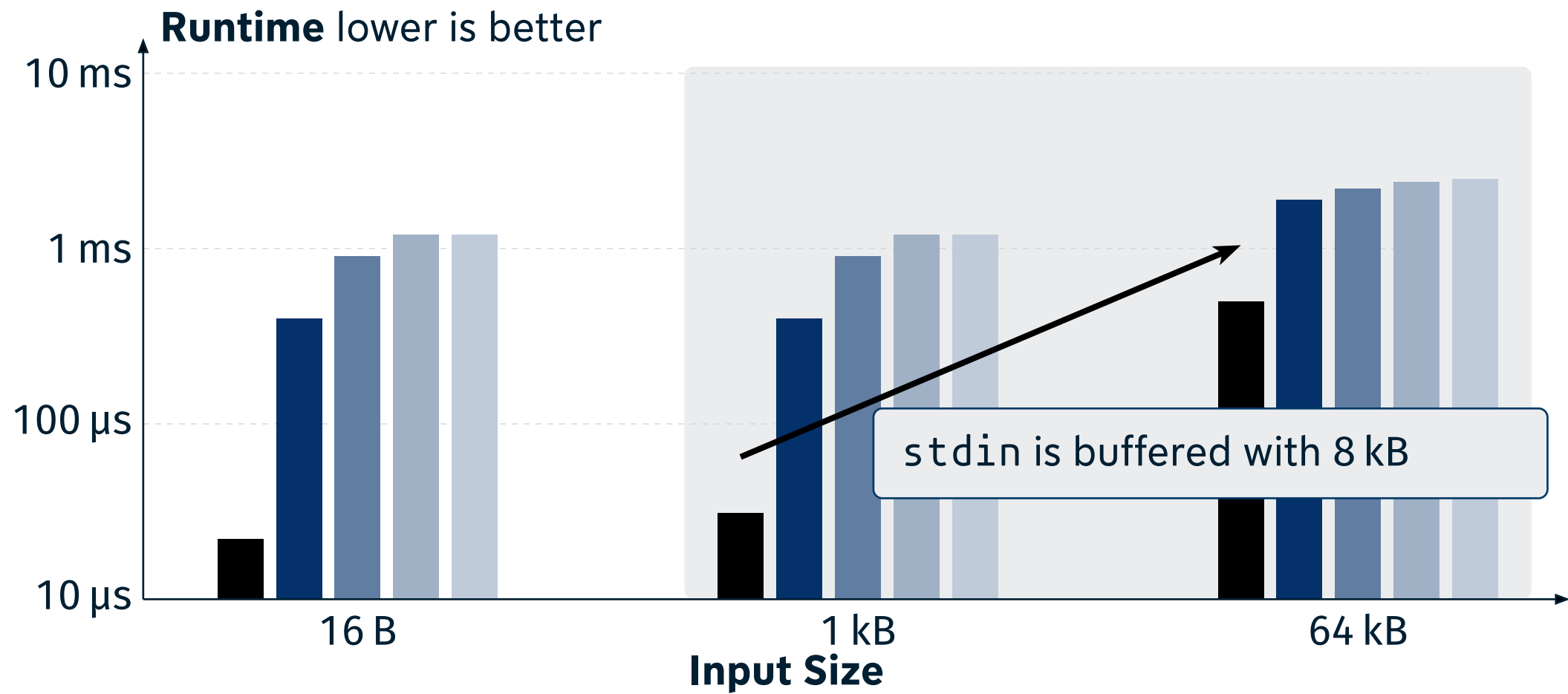
- representative systems application
- reads from `stdin` and computes
- uses an external crate: `base64`

Isolation

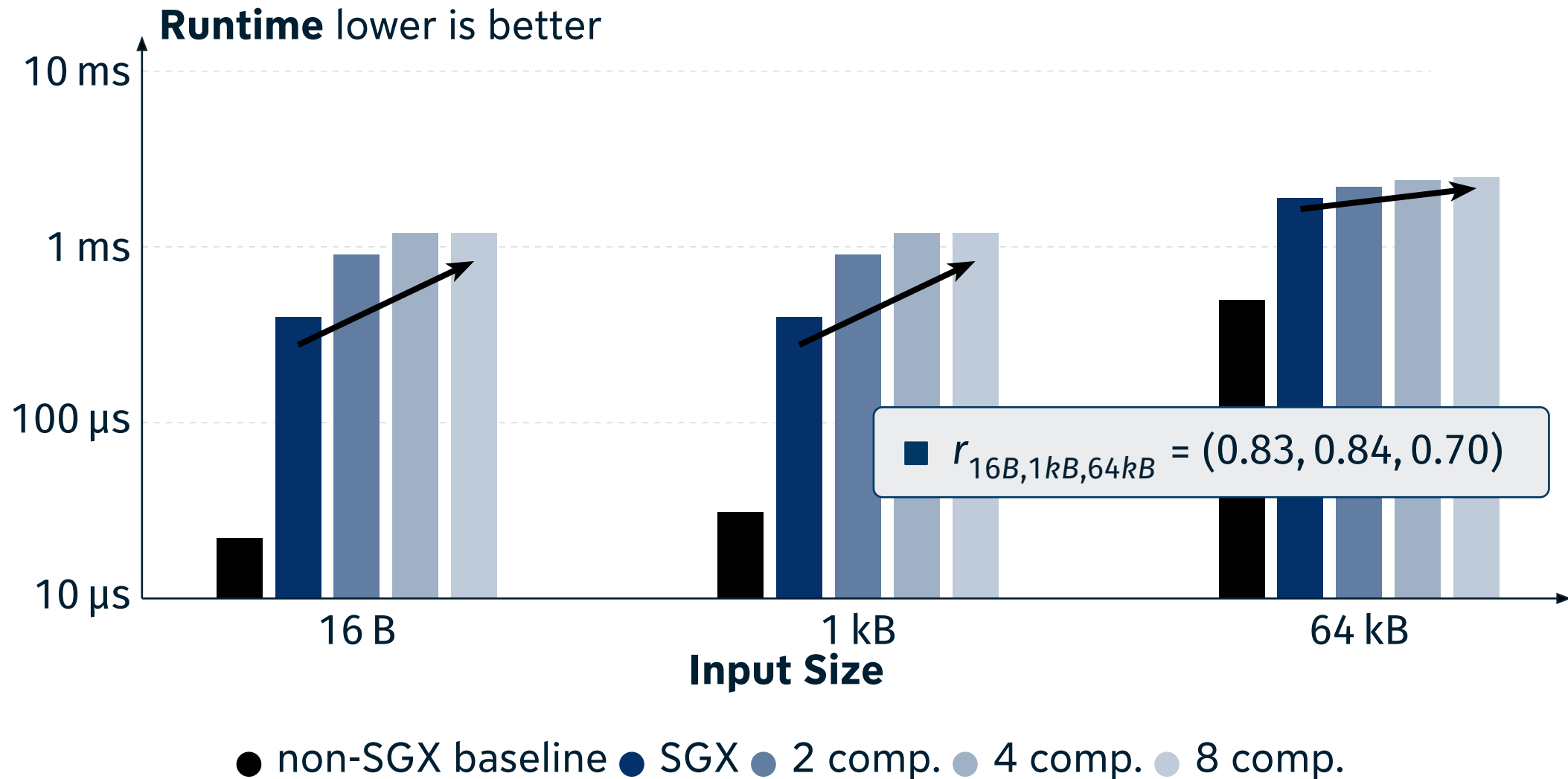
- SGX as isolation mechanism

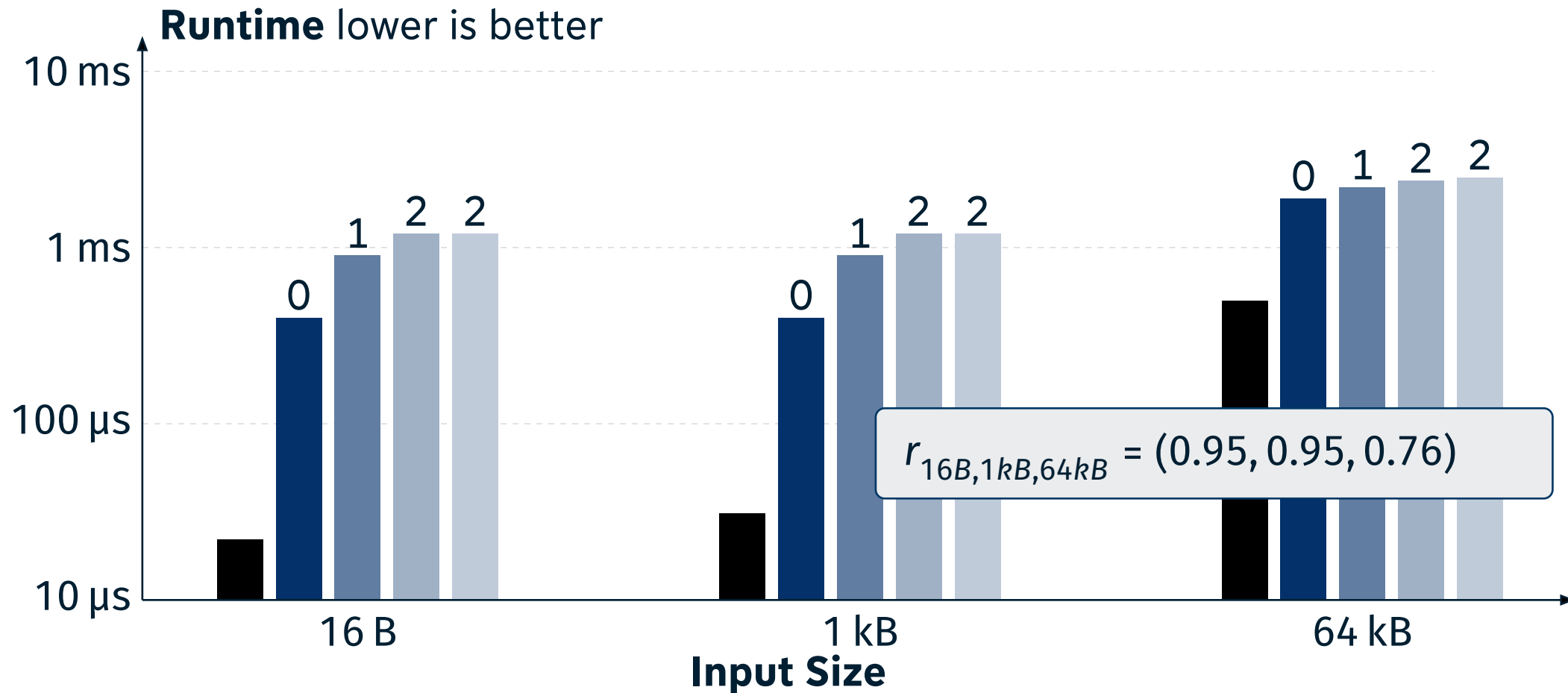
Measured Runtime





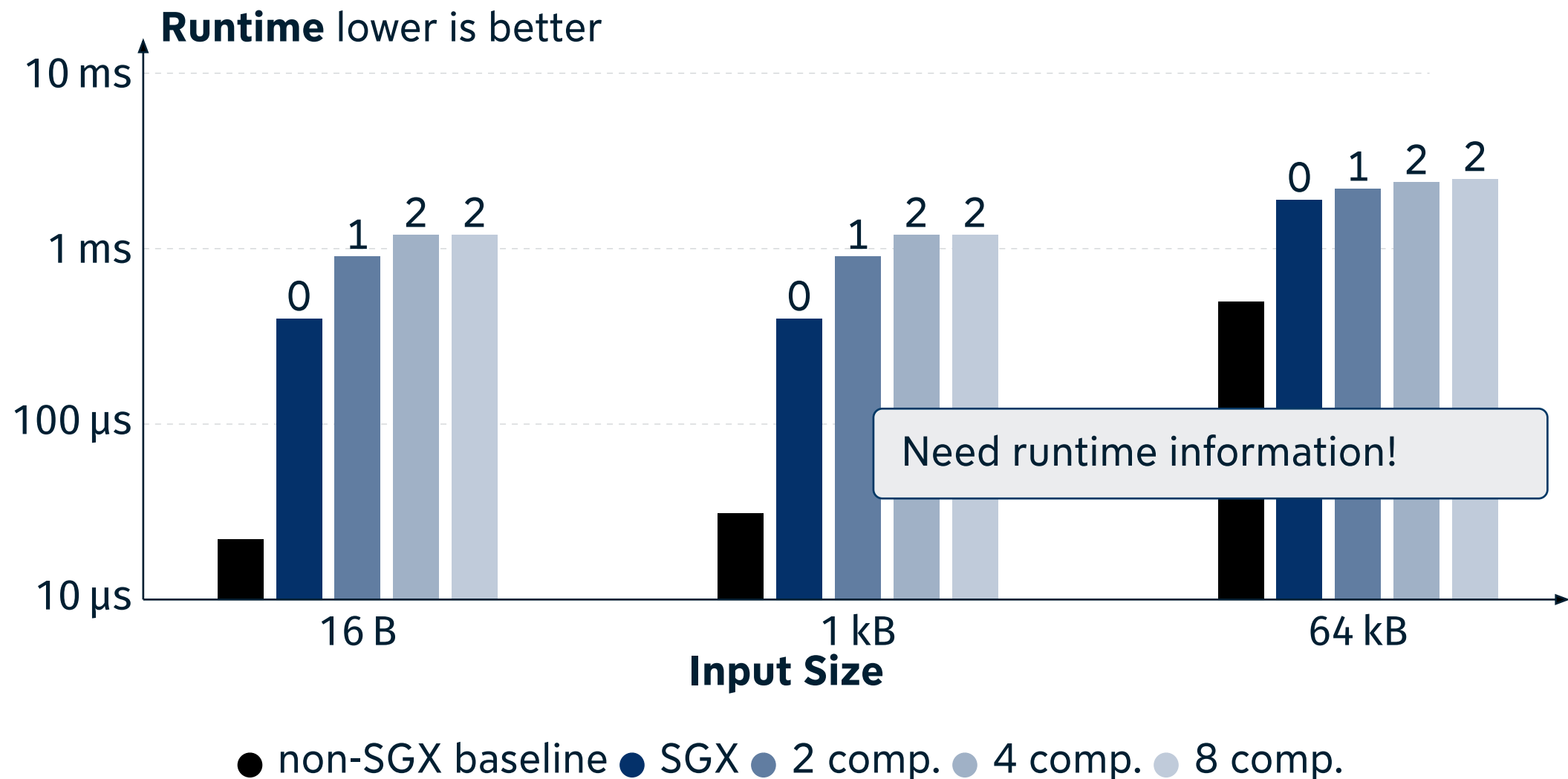
● non-SGX baseline ● SGX ● 2 comp. ● 4 comp. ● 8 comp.





● non-SGX baseline ● SGX ● 2 comp. ● 4 comp. ● 8 comp.

Measured Runtime



Towards Automatic Compartmentalization

zBulk: a Source-to-Source compiler

C1: Automatic Instrumentation (✓)

C2: Compartment Entry Dispatch ✓

C3: Performance-Security Tradeoff (✓)

Towards Automatic Compartmentalization

zBulk: a Source-to-Source compiler

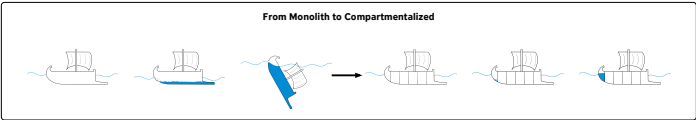
C1: Automatic Instrumentation (✓)

C2: Compartment Entry Dispatch ✓

C3: Performance-Security Tradeoff (✓)

zBulk: Towards Automatic Compartmentalization of Existing Rust Applications with Zero Retrofitting

Maxim Ritter von Onciul, Phillip Raffeck, Peter Wägemann, Rüdiger Kapitza



Problems with Manual Compartmentalization

Code Snippet	Comment	Change is
<pre>pub mod a { mod b { fn foo() { fn bar() { // ... } } } }</pre>	Is this good?	Expensive!
<pre>pub mod a { mod b { fn foo() { switch to B } } }</pre>	Change is	Expensive!

Challenges

- Function Instrumentation**
 - How do we instrument functions into compartment interfaces?
 - How can we stay isolation-mechanism agnostic?
- Compartment Entry Dispatch**
 - How do we correctly identify functions across compartments?
 - There is no dynamic runtime reflection in Rust.
- Performance-Security Tradeoff**
 - Where does overhead occur?
 - How can we reduce overhead?
 - How can we maintain security expectations?

Solution: Automatic Compartmentalization

zBulk Architecture

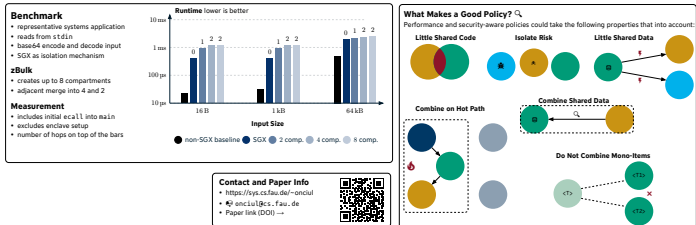
- zCargo**: Rust compiler interface, HRB pass + MIR, Construct Placement
- zBulk**: HRB pass + MIR, Main Transformation
- zBulk**: HRB pass + MIR, Main Transformation
- zBulk**: HRB pass + MIR, Main Transformation

Instrumentation

```
// original code  
fn foo() { let x = bar(Cargo, arg1); }  
fn bar(Cargo: T0, arg1: T1) -> T2 {  
  // ...  
}
```

Non-Trivial Types

```
struct ProblemType {  
  // requires of Serialize + Deserialize  
  generic: T;  
  // change impl Trait + Serialize + Deserialize  
  generic: impl Trait;  
  // how to support generic dispatch  
  dynamic_dispatch: dyn Trait;  
}
```



Towards Automatic Compartmentalization

zBulk: a Source-to-Source compiler

C1: Automatic Instrumentation (✓)

C2: Compartment Entry Dispatch

C3: Performance-Security Tradeoff (✓)

Questions?

- Poster session
- onciul@cs.fau.de
- Paper link (DOI) →



zBulk: Towards Automatic Compartmentalization of Existing Rust Applications with Zero Retrofitting

Maxim Ritter von Onciul, Phillip Raffeck, Peter Wägemann, Rüdiger Kapitza

