Memory Safety is Merely Table Stakes

Safe Interactions with Foreign Languages through Omniglot

Leon Schuermann, Jack Toubes, Tyler Potyondy,
Pat Pannuto, Mae Milano, Amit Levy

PRINCETON UNIVERSITY

UC San Diego

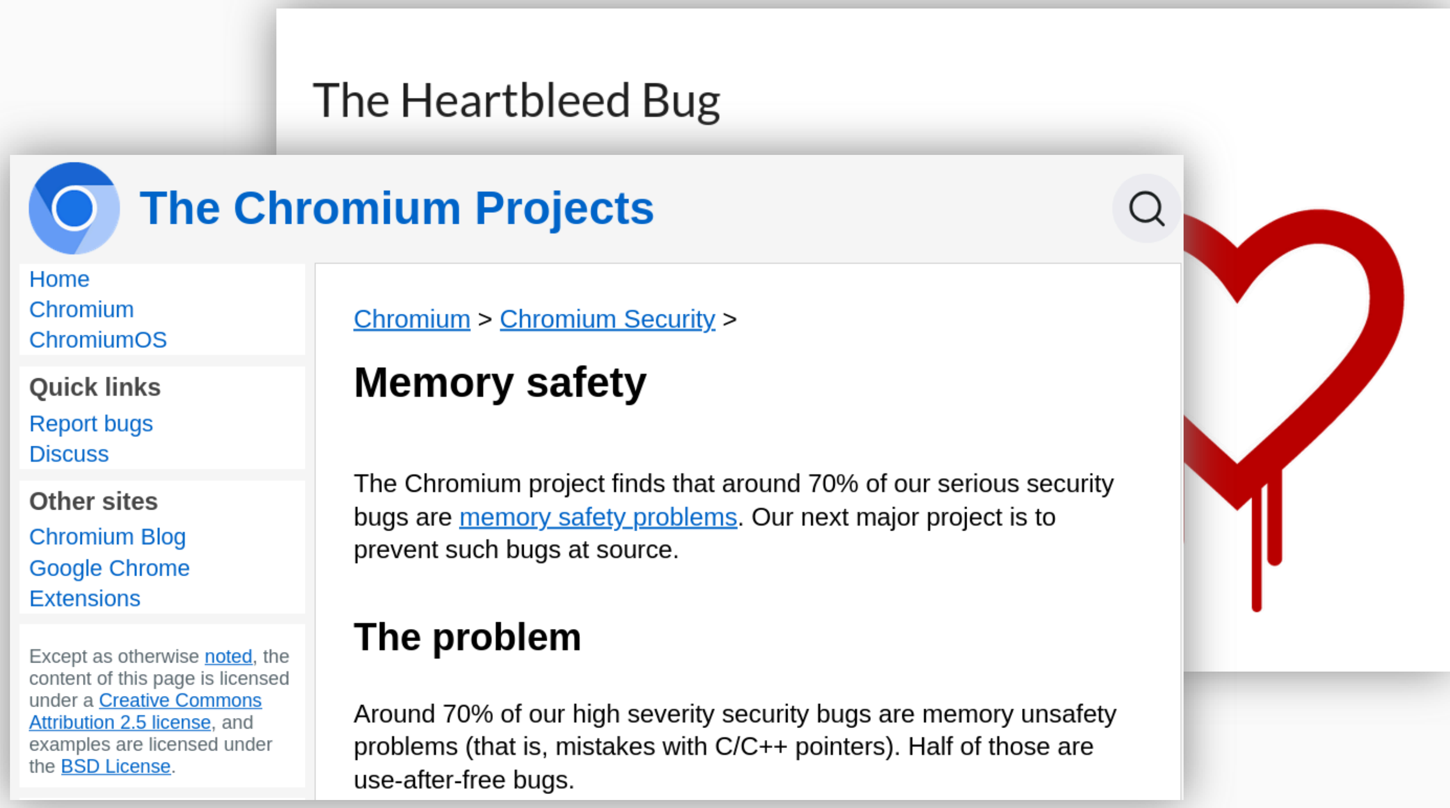
Safety Bugs Plague Software for Decades

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

Safety Bugs Plague Software for Decades



Safety Bugs Plague Software for Decades

The Heartbleed Bug

The Chromium Projects

of our serious security
najor project is to

s are memory unsafety
rs). Half of those are

SIGN IN / UP

The Register®

Boeing 787 software bug can shut down planes' generators IN FLIGHT

Have you turned it off and on again? That's the way to stop the plane becoming a brick

Simon Sharwood

Fri 1 May 2015 // 06:30 UTC

The US Federal Aviation Administration (FAA) has issued a new airworthiness directive (PDF) for Boeing's 787 because a software bug shuts down the plane's electricity generators every 248 days.

“We have been advised by Boeing of an issue identified during laboratory testing,” the directive says. That issue sees “The software counter internal to the generator control units (GCUs) will

Hi, I'm Leon!

- Ph.D. Candidate at Princeton

Hi, I'm Leon!

- Ph.D. Candidate at Princeton

Hi, I'm Leon!

- Ph.D. Candidate at Princeton
- Core Maintainer of Tock OS
- Care About Software Being Safe and Secure

Safe Programming Languages Eliminate Memory Safety Vulnerabilities

SIGN IN / UP The Register

Microsoft is busy rewriting core Windows code in memory-safe Rust

Now that's a C change we can back

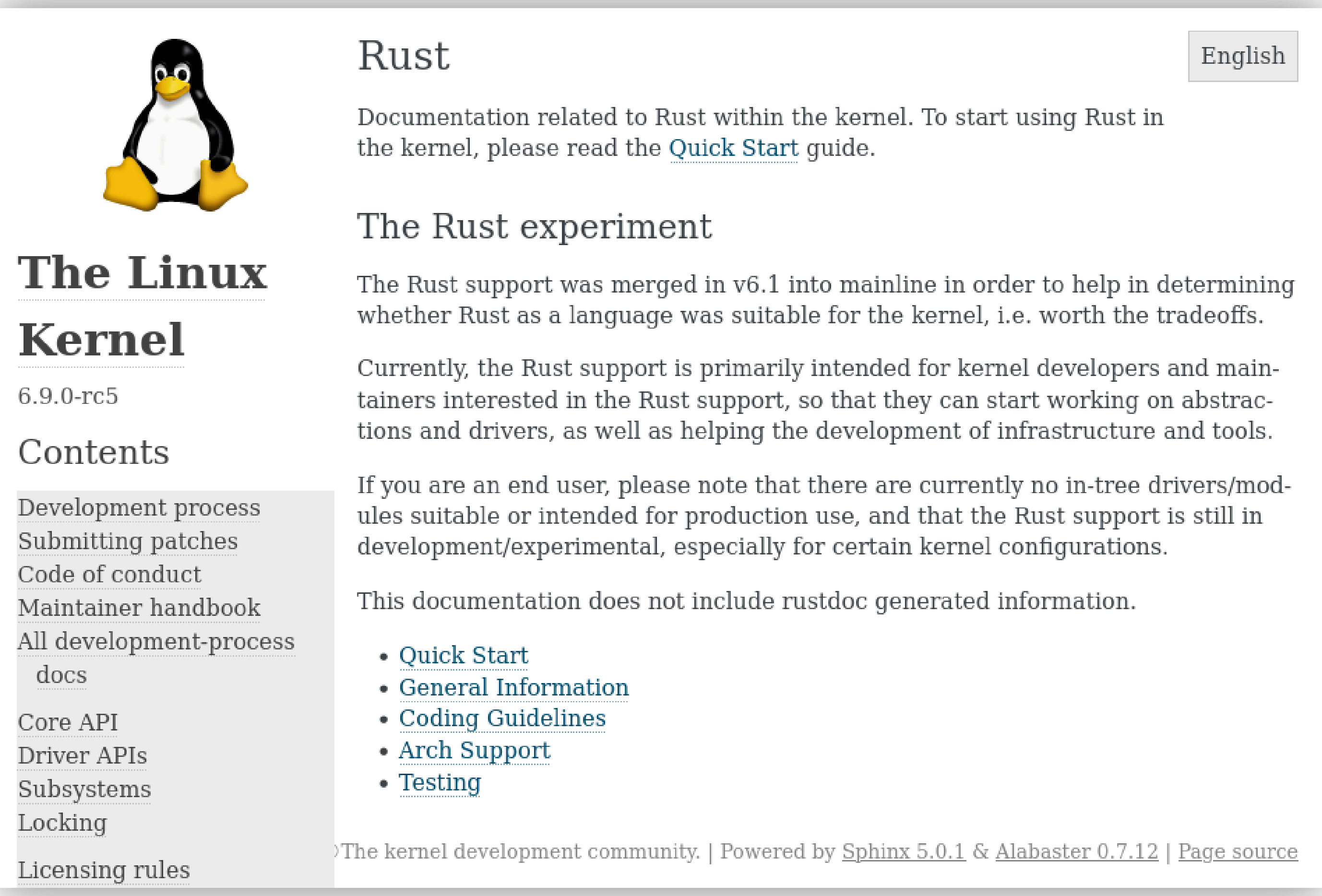
Thomas Claburn   Thu 27 Apr 2023 // 20:45 UTC

Microsoft is rewriting core Windows libraries in the Rust programming language, and the more memory-safe code is already reaching developers.

David "dwizzle" Weston, director of OS security for Windows, announced the arrival of Rust in the operating system's kernel at BlueHat IL 2023 in Tel Aviv, Israel, last month.

"You will actually have Windows booting with Rust in the kernel in probably the next several weeks or months, which is really cool," he said. "The basic goal here was to convert some of these internal C++ data types into their Rust equivalents."

Safe Programming Languages Eliminate Memory Safety Vulnerabilities



The Linux Kernel

6.9.0-rc5

Contents

- Development process
- Submitting patches
- Code of conduct
- Maintainer handbook
- All development-process docs
- Core API
- Driver APIs
- Subsystems
- Locking
- Licensing rules

English

Rust

Documentation related to Rust within the kernel. To start using Rust in the kernel, please read the Quick Start guide.

The Rust experiment

The Rust support was merged in v6.1 into mainline in order to help in determining whether Rust as a language was suitable for the kernel, i.e. worth the tradeoffs.

Currently, the Rust support is primarily intended for kernel developers and maintainers interested in the Rust support, so that they can start working on abstractions and drivers, as well as helping the development of infrastructure and tools.

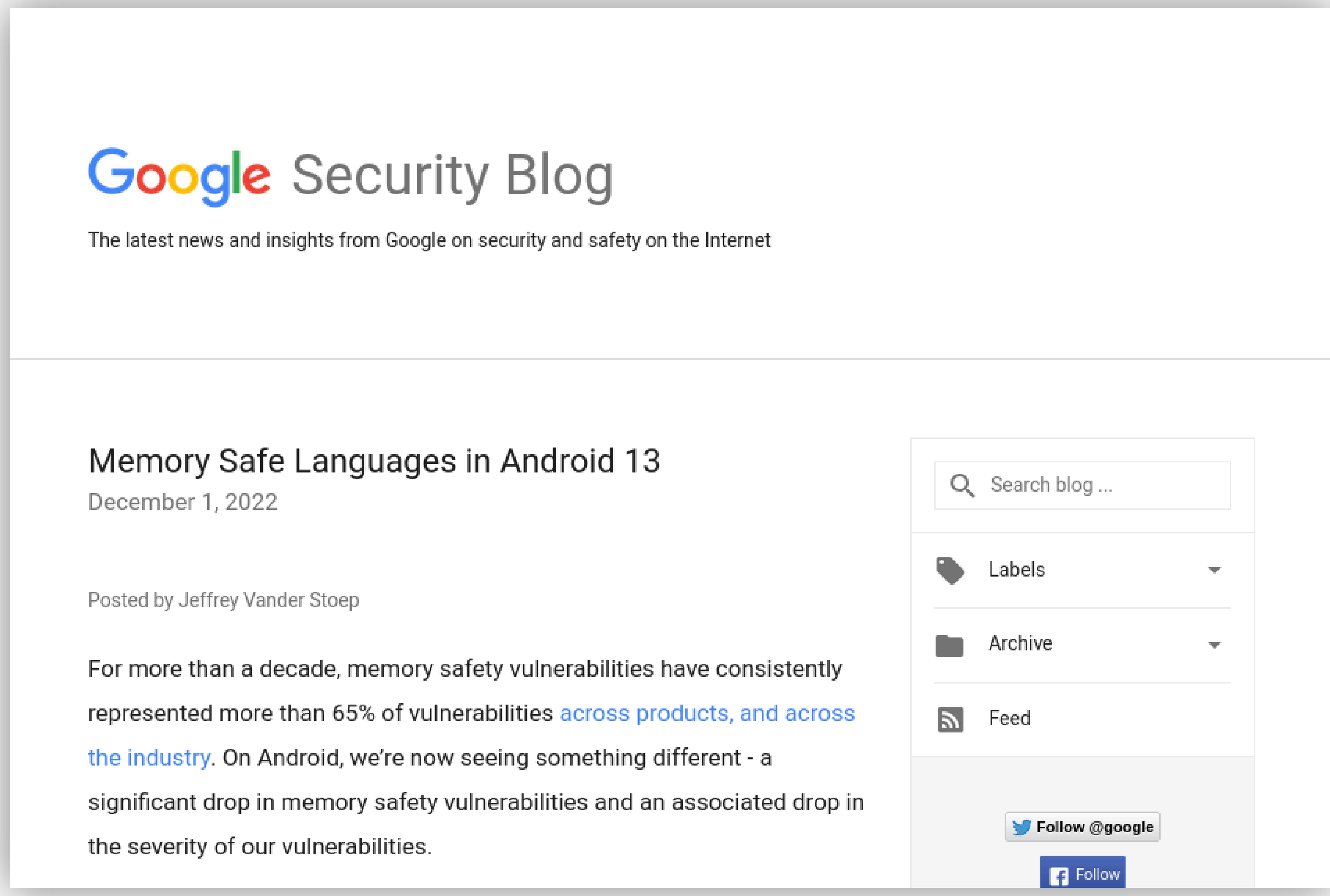
If you are an end user, please note that there are currently no in-tree drivers/modules suitable or intended for production use, and that the Rust support is still in development/experimental, especially for certain kernel configurations.

This documentation does not include rustdoc generated information.

- Quick Start
- General Information
- Coding Guidelines
- Arch Support
- Testing

©The kernel development community. | Powered by Sphinx 5.0.1 & Alabaster 0.7.12 | Page source

Safe Programming Languages Eliminate Memory Safety Vulnerabilities



Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Memory Safe Languages in Android 13

December 1, 2022

Posted by Jeffrey Vander Stoep

For more than a decade, memory safety vulnerabilities have consistently represented more than 65% of vulnerabilities across products, and across the industry. On Android, we're now seeing something different - a significant drop in memory safety vulnerabilities and an associated drop in the severity of our vulnerabilities.

Search blog ...

Labels

Archive

Feed

Follow @google

Follow

Safe Programming Languages Eliminate Memory Safety Vulnerabilities

Posted by Jeffrey Vander Stoep

For more than a decade, memory safety vulnerabilities have consistently represented more than 65% of vulnerabilities across products, and across the industry. On Android, we're now seeing something different - a significant drop in memory safety vulnerabilities and an associated drop in the severity of our vulnerabilities.

Looking at vulnerabilities reported in the Android security bulletin, which includes critical/high severity vulnerabilities reported through our vulnerability rewards program

We Cannot Rewrite all Code in Safe Programming Languages

Existing systems span millions of lines of code

We Cannot Rewrite all Code in Safe Programming Languages

Existing systems span millions of lines of code

Even new systems must integrate existing, proven, optimized, and tested libraries

We Cannot Rewrite all Code in Safe Programming Languages

Existing systems span millions of lines of code

Even new systems must integrate existing, proven, optimized, and tested libraries

Rewriting is impractical: cost, time, domain expertise, certification requirements, ...

We Cannot Rewrite all Code in Safe Programming Languages

libsodium

Existing systems span millions of lines of code

Even new systems must integrate existing, proven, optimized, and tested libraries

→ Safe languages like Rust must be able to *safely* interact with code written in foreign languages

1. What makes interactions with foreign languages *safe*?

1. What makes interactions with foreign languages *safe*?

---

2. Safely integrate arbitrary, untrusted foreign libraries with Omniglot

1. What makes interactions with foreign languages *safe*?

---

2. Safely integrate arbitrary, untrusted foreign libraries with Omniglot

Memory Safety is Merely Table Stakes

Key Property: Memory Safety

Memory Safety is Merely Table Stakes

What does Memory Safety mean?

Memory Safety is Merely Table Stakes

> *"Memory safety [describes] whether software or a programming language is designed to prevent memory bugs and vulnerabilities."*
>
> —Internet Society

"Memory safety [describes] whether software or a programming language is designed to prevent memory bugs and vulnerabilities."

—Internet Society

Relates to the absence of bugs like:

Buffer Overflows	

Memory Safety is Merely Table Stakes

> *"Memory safety [describes] whether software or a programming language is designed to prevent memory bugs and vulnerabilities."*
>
> —Internet Society

Relates to the absence of bugs like:

Buffer Overflows	Use-After-Free

"Memory safety [describes] whether software or a programming language is designed to prevent memory bugs and vulnerabilities."

—Internet Society

Relates to the <u>absence</u> of bugs like:

Buffer Overflows	Use-After-Free
Data Races	

"Memory safety [describes] whether software or a programming language is designed to prevent memory bugs and vulnerabilities."

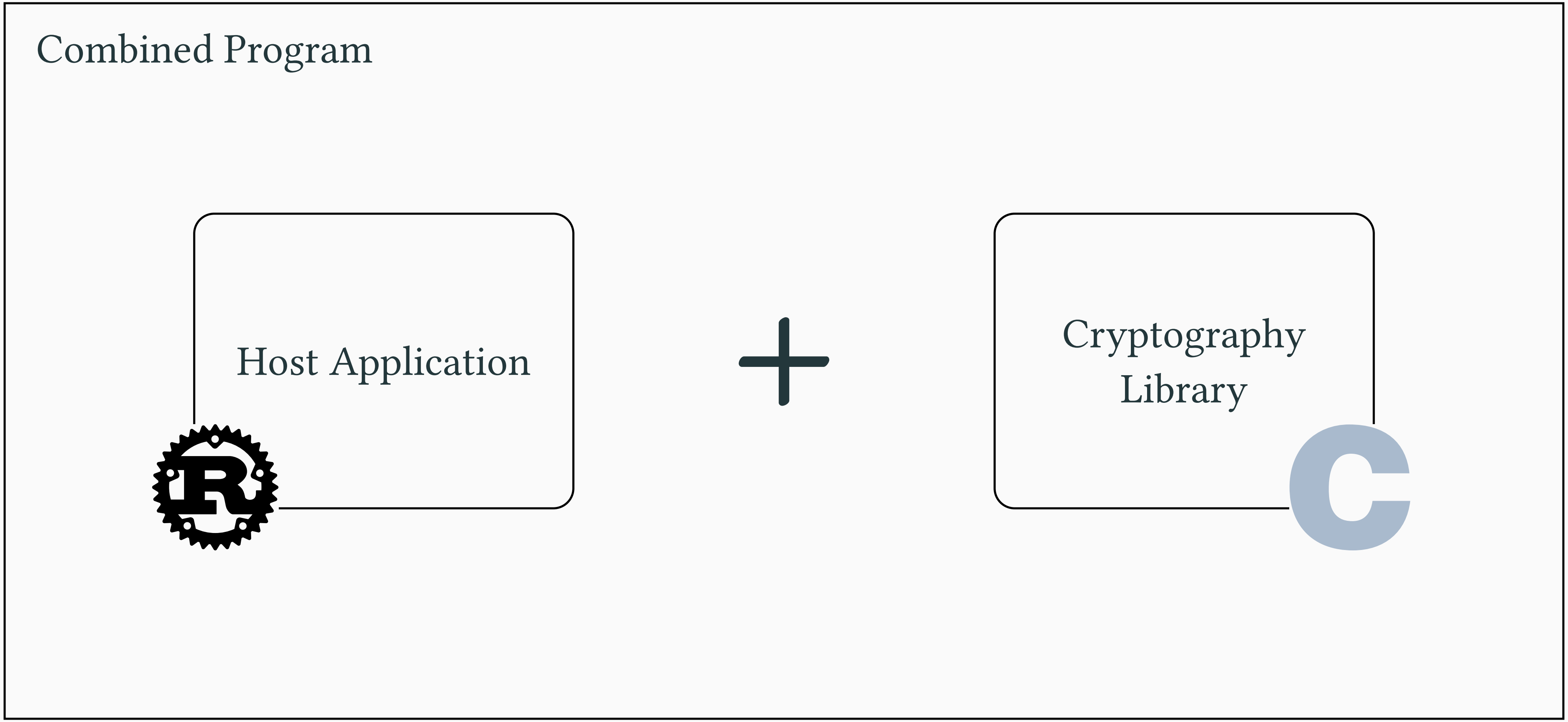
—Internet Society

Relates to the absence of bugs like:

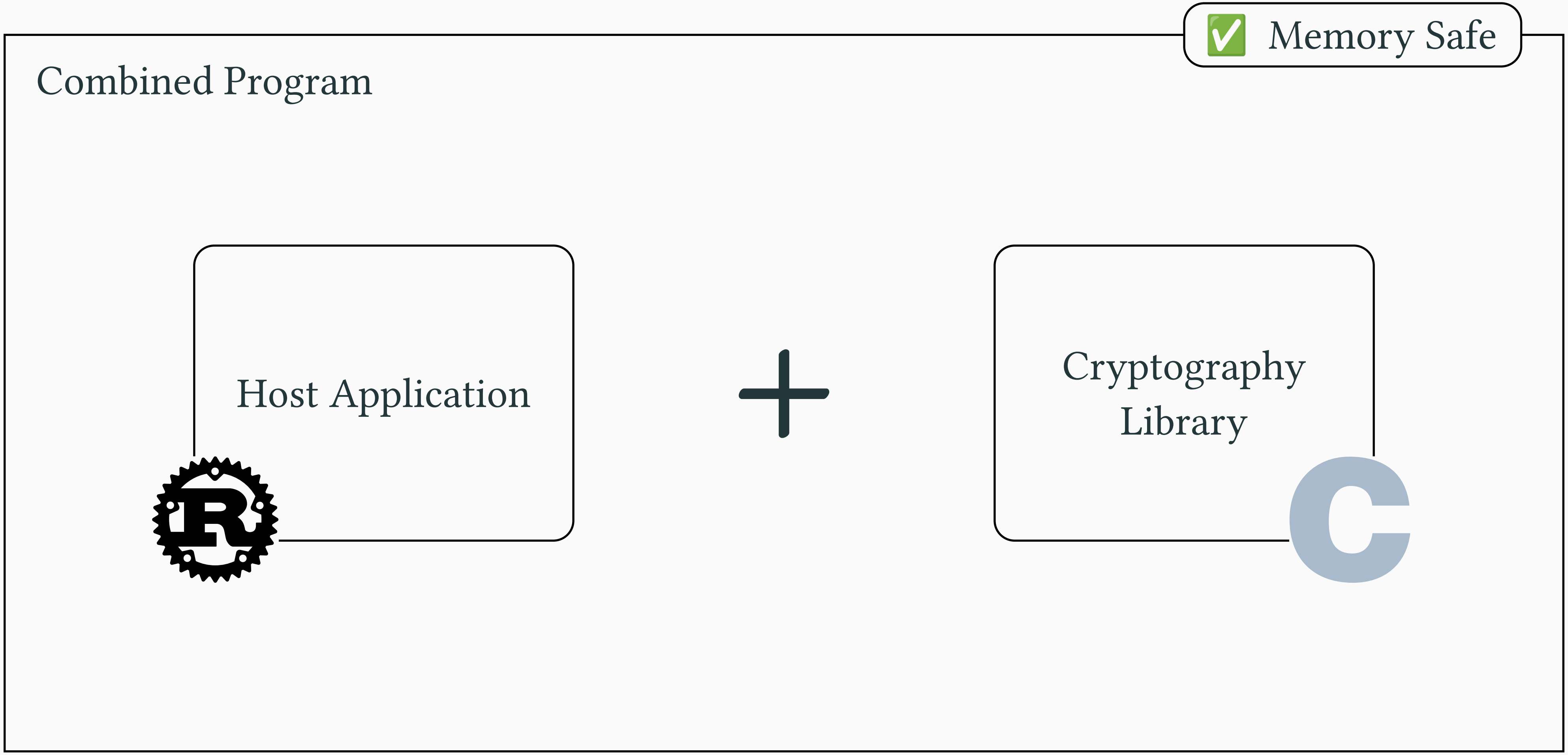
Buffer Overflows	Use-After-Free
Data Races	Uninitialized Accesses

Memory Safety is Merely Table Stakes

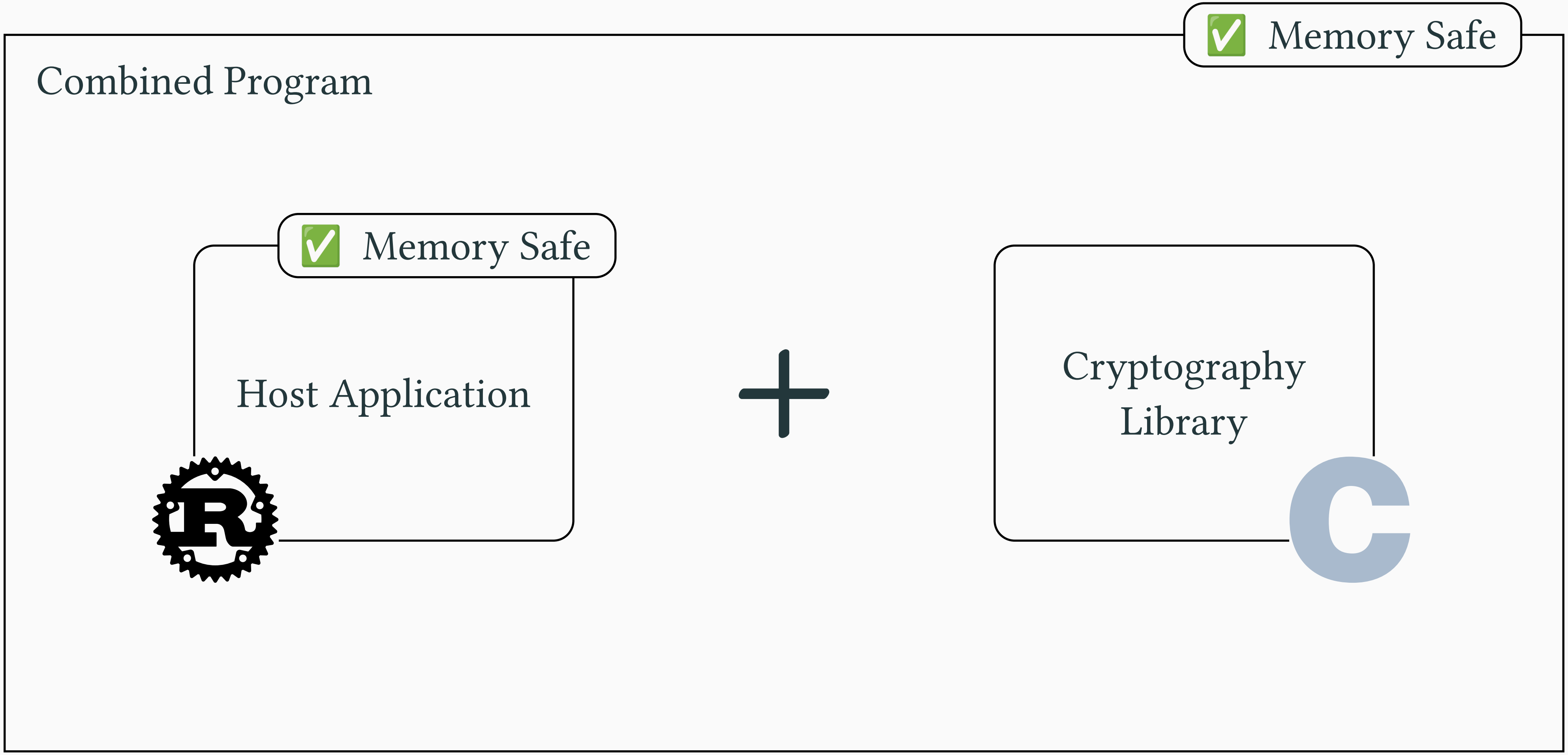
Memory Safety is Merely Table Stakes



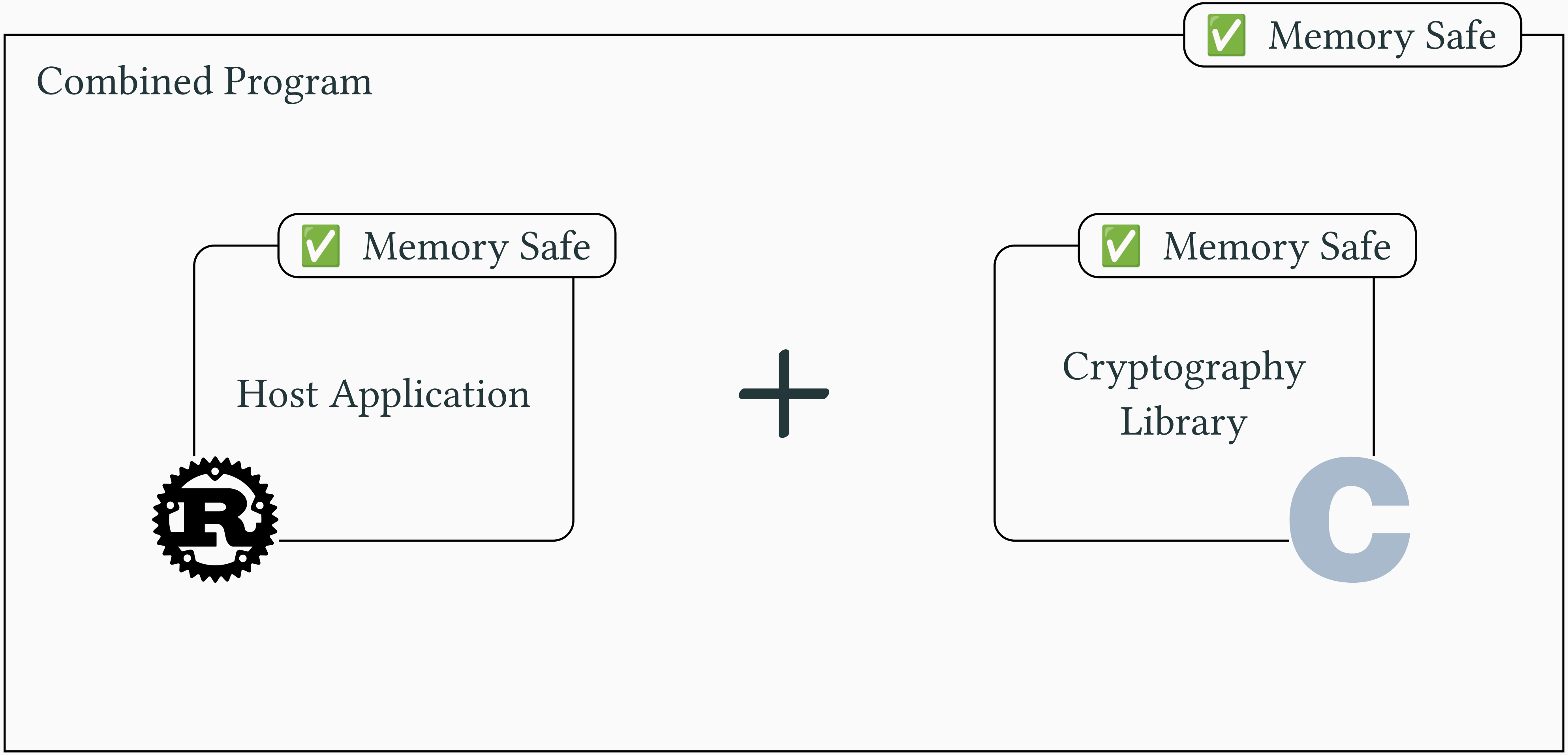
Memory Safety is Merely Table Stakes



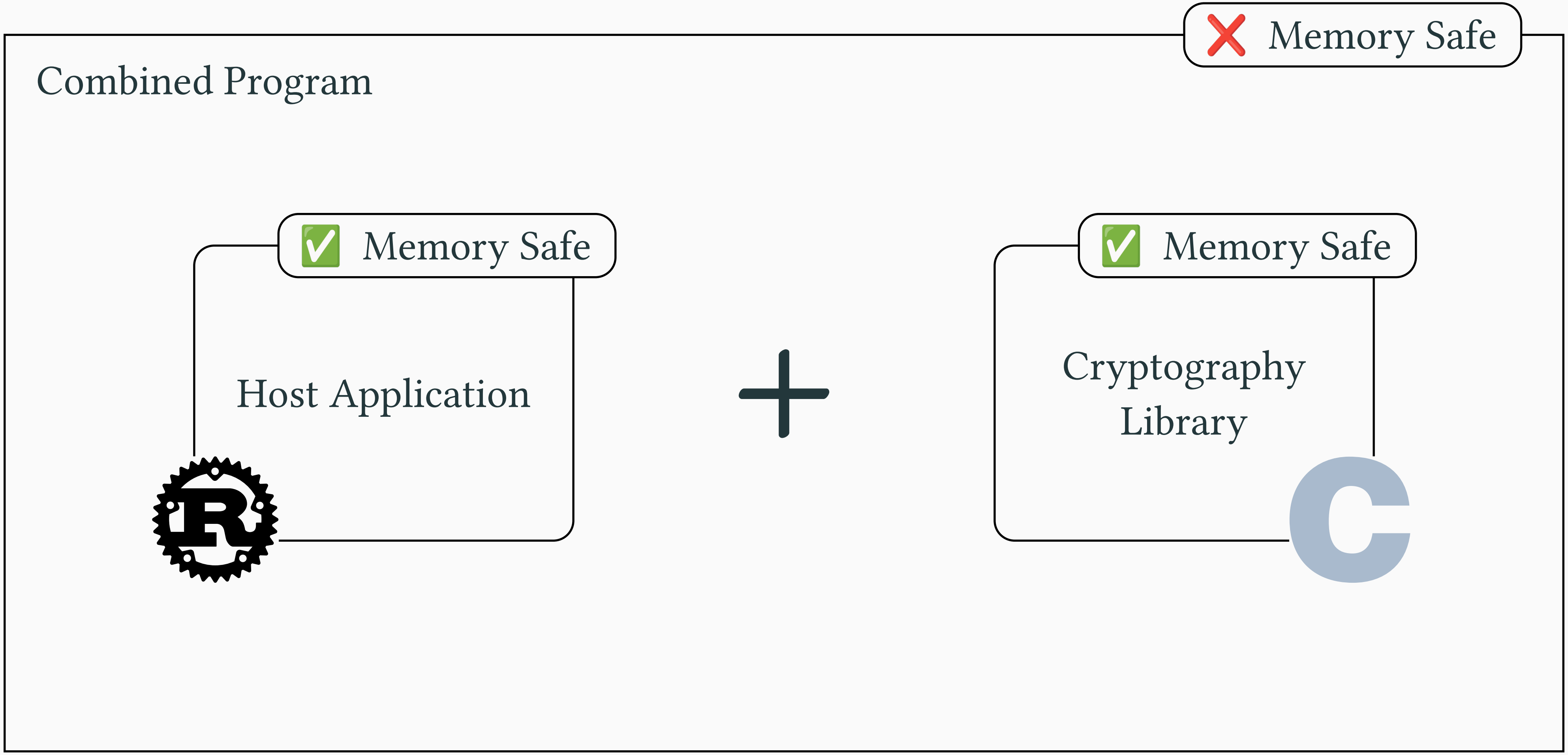
Memory Safety is Merely Table Stakes



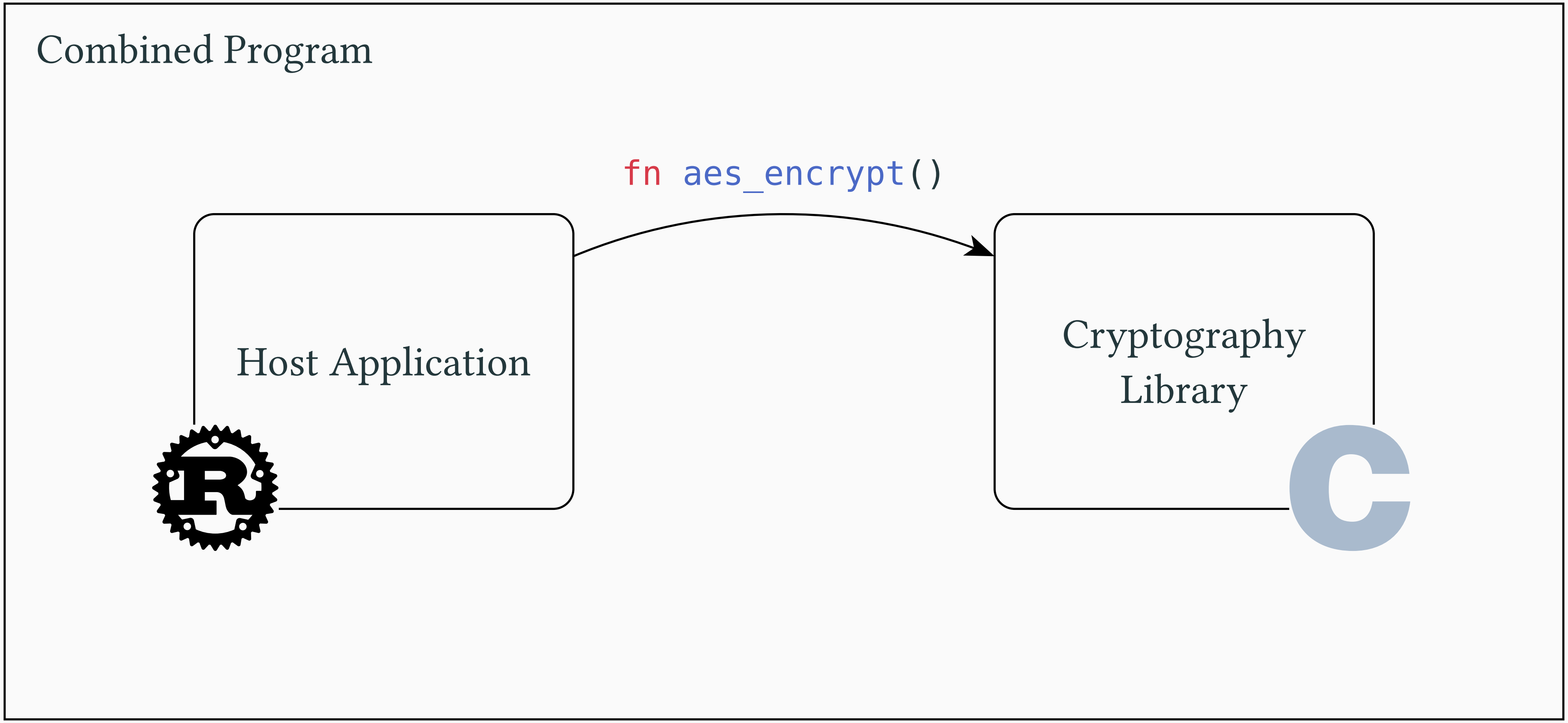
Memory Safety is Merely Table Stakes



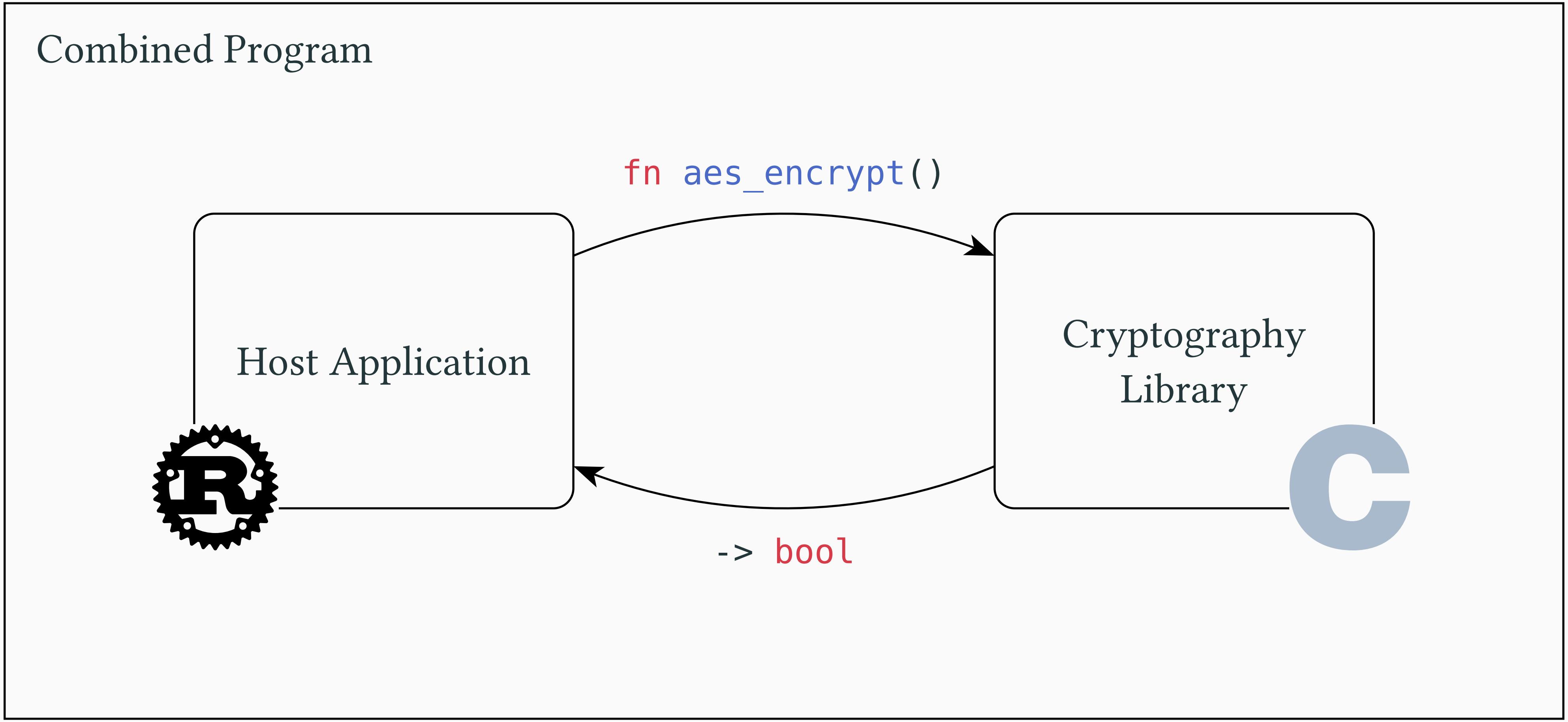
Memory Safety is Merely Table Stakes



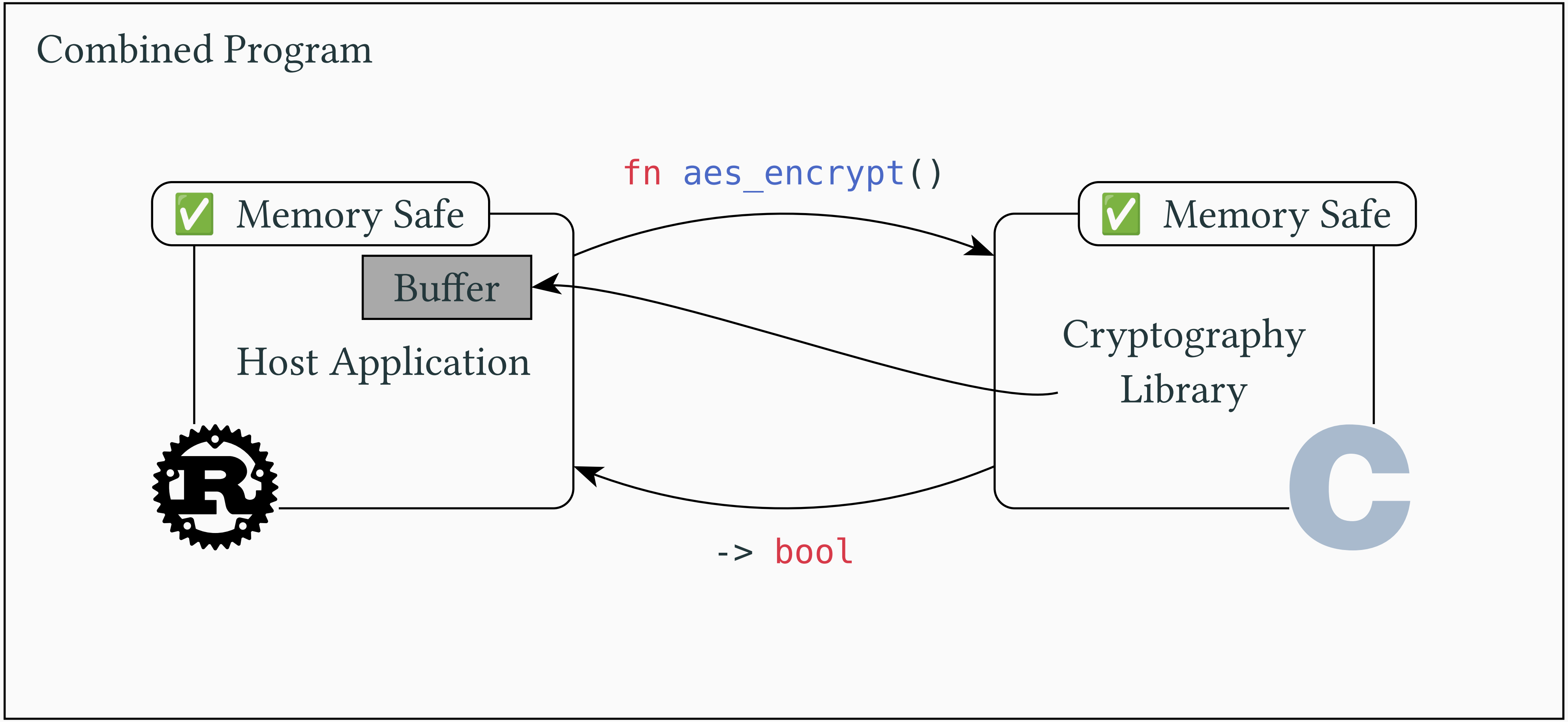
Memory Safety is Merely Table Stakes



Memory Safety is Merely Table Stakes



Memory Safety is Merely Table Stakes



A Type Safety Violation Breaks Memory Safety

✅ Memory Safe

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

A Type Safety Violation Breaks Memory Safety

✅ Memory Safe

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

A Type Safety Violation Breaks Memory Safety

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```rust
enum bool {
  false = 0,
  true = 1,
}
```

A Type Safety Violation Breaks Memory Safety

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:

0	bool (0 or 1)	Vec Pointer	Vec Capacity	Vec Length

`Message::Unencrypted`:

1	CString Pointer			

A Type Safety Violation Breaks Memory Safety

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```rust
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:

0	bool (0 or 1)	Vec Pointer	Vec Capacity	Vec Length

`Message::Unencrypted`:

1	CString Pointer			

A Type Safety Violation Breaks Memory Safety

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```
enum bool {
  false = 0,
  true = 1,
}
```

Message::Encrypted:

0	bool (0 or 1)	Vec Pointer	Vec Capacity	Vec Length

Message::Unencrypted:

1	CString Pointer			

A Type Safety Violation Breaks Memory Safety

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```
enum bool {
  false = 0,
  true = 1,
}
```

Message::Encrypted:

0	bool (0 or 1)	Vec Pointer	Vec Capacity	Vec Length

Message::Unencrypted:

1	CString Pointer			

A Type Safety Violation Breaks Memory Safety

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```
enum bool {
  false = 0,
  true = 1,
}
```

Message::Encrypted:

0	bool (0 or 1)	Vec Pointer	Vec Capacity	Vec Length

Message::Unencrypted:

1	CString Pointer			

A Type Safety Violation Breaks Memory Safety

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```rust
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:

`bool` (`0` or `1`)	`Vec` Pointer	`Vec` Capacity	`Vec` Length

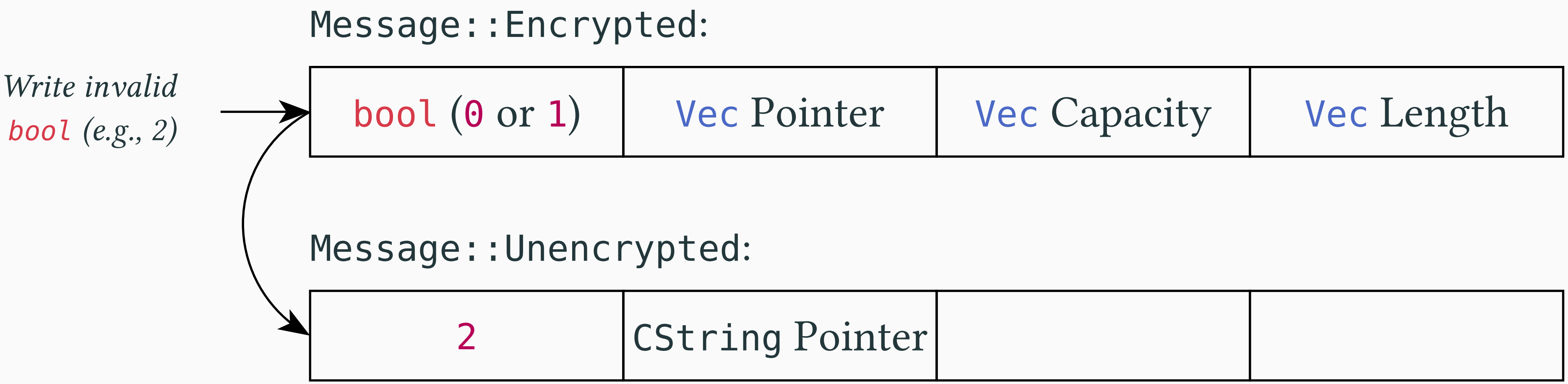
`Message::Unencrypted`:

`2`	`CString` Pointer		

A Type Safety Violation Breaks Memory Safety

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

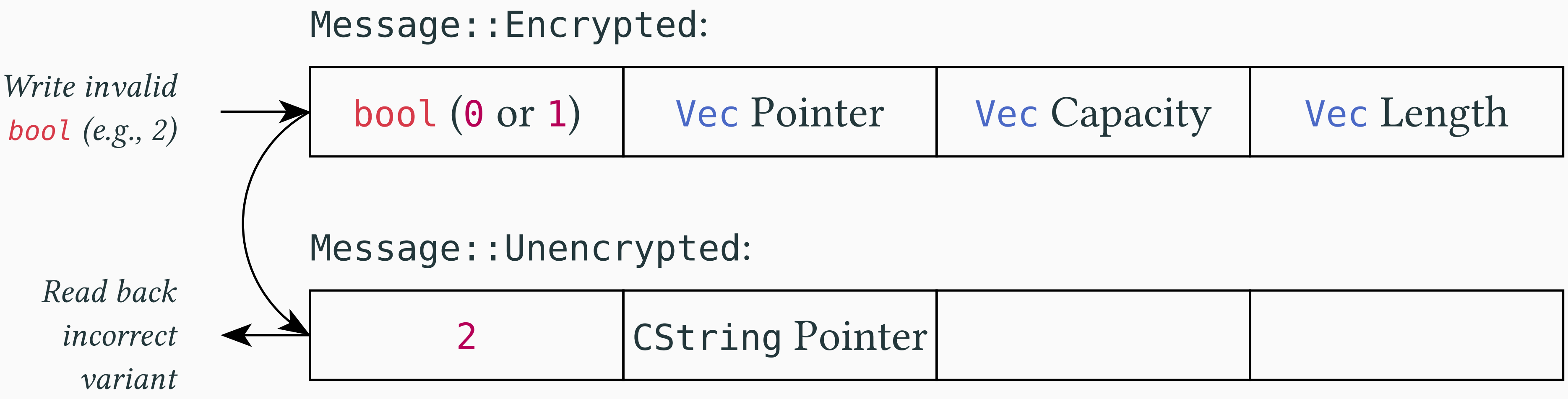
```rust
enum bool {
  false = 0,
  true = 1,
}
```



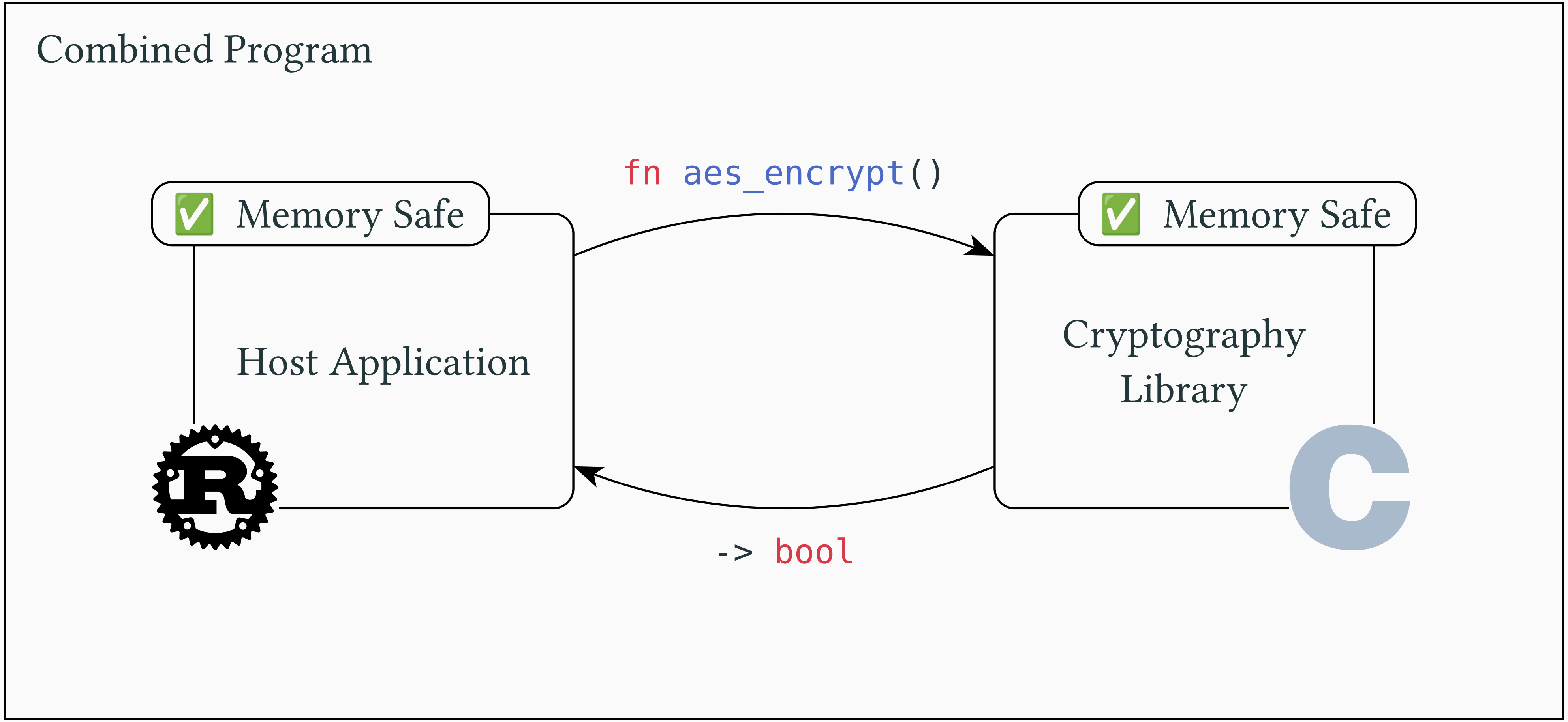
A Type Safety Violation Breaks Memory Safety

```
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

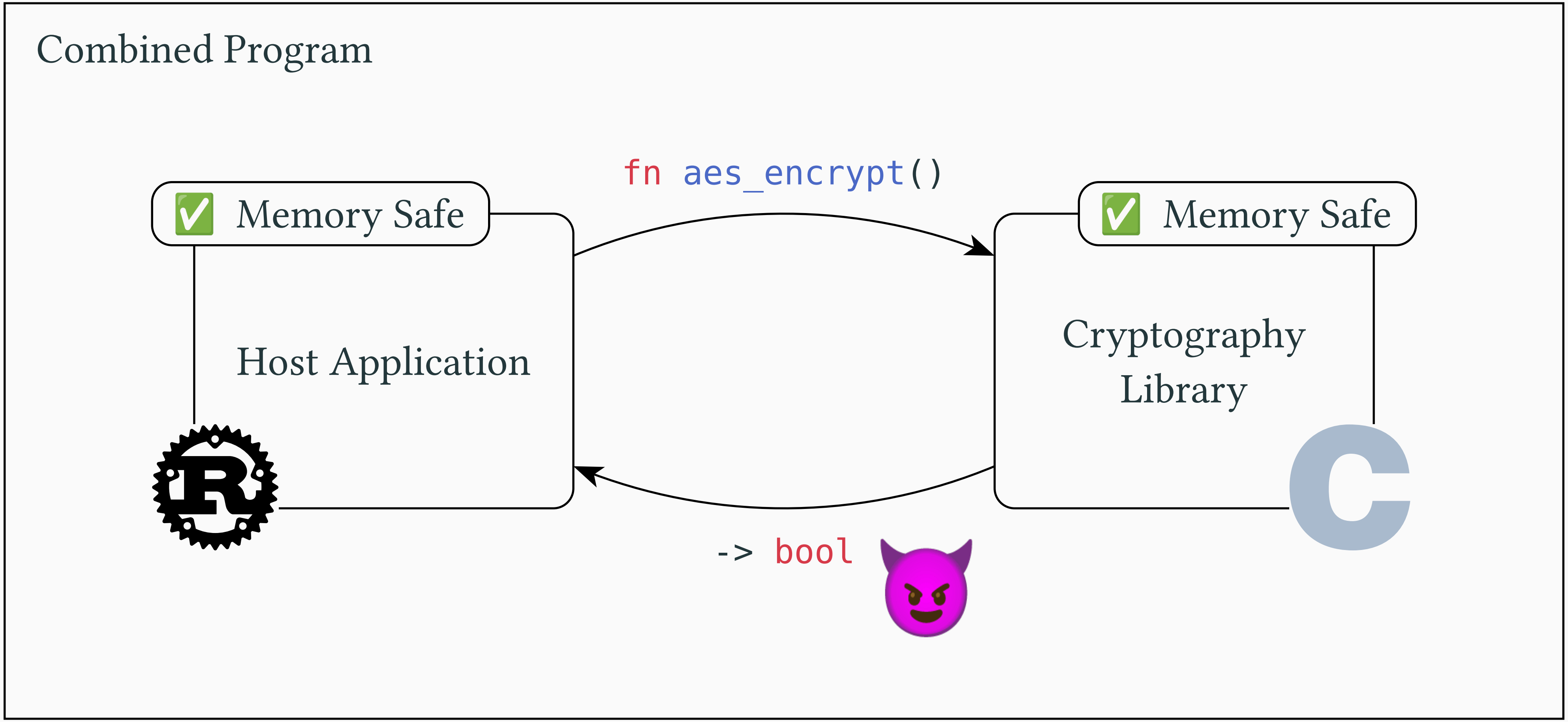
```
enum bool {
  false = 0,
  true = 1,
}
```



A Type Safety Violation Breaks Memory Safety



A Type Safety Violation Breaks Memory Safety



A Type Safety Violation Breaks Memory Safety

Behavior Considered Und

https://doc.rust-lang.org/reference/behavior-considered-undefined.html

Invalid values

The Rust compiler assumes that all values produced during program execution are “valid”, and producing an invalid value is hence immediate UB.

Whether a value is valid depends on the type:

- A `bool` value must be `false` (`0`) or `true` (`1`).
- A `fn` pointer value must be non-null.
- A `char` value must not be a surrogate (i.e., must not be in the range `0xD800..=0xDFFF`) and must be equal to or less than `char::MAX`.

Memory safety *alone* is not a useful property when reasoning about the composition of program components.

Memory safety *alone* is not a useful property when reasoning about the composition of program components.

→ We must also consider type safety and other invariants.

Additional Invariants for a Foreign Function Call

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

Additional Invariants for a Foreign Function Call

✅ Memory Safe

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

Additional Invariants for a Foreign Function Call

✅ Memory Safe

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

Valid Values

Additional Invariants for a Foreign Function Call

✅ Memory Safe

```
extern "C" {
  fn aes_encrypt(buf: *mut u8, len: usize) -> bool;
}
```

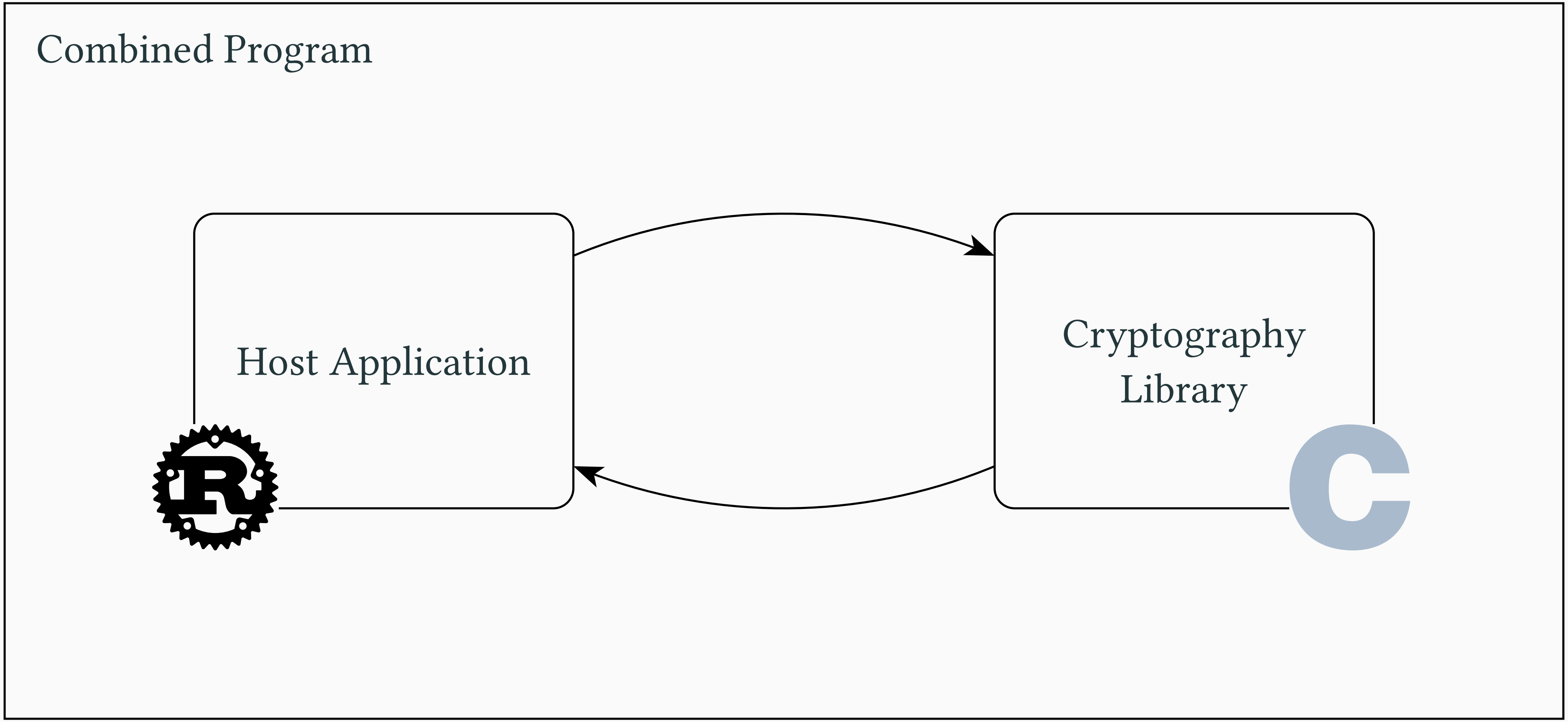
Aliasing ⊕ Mutability

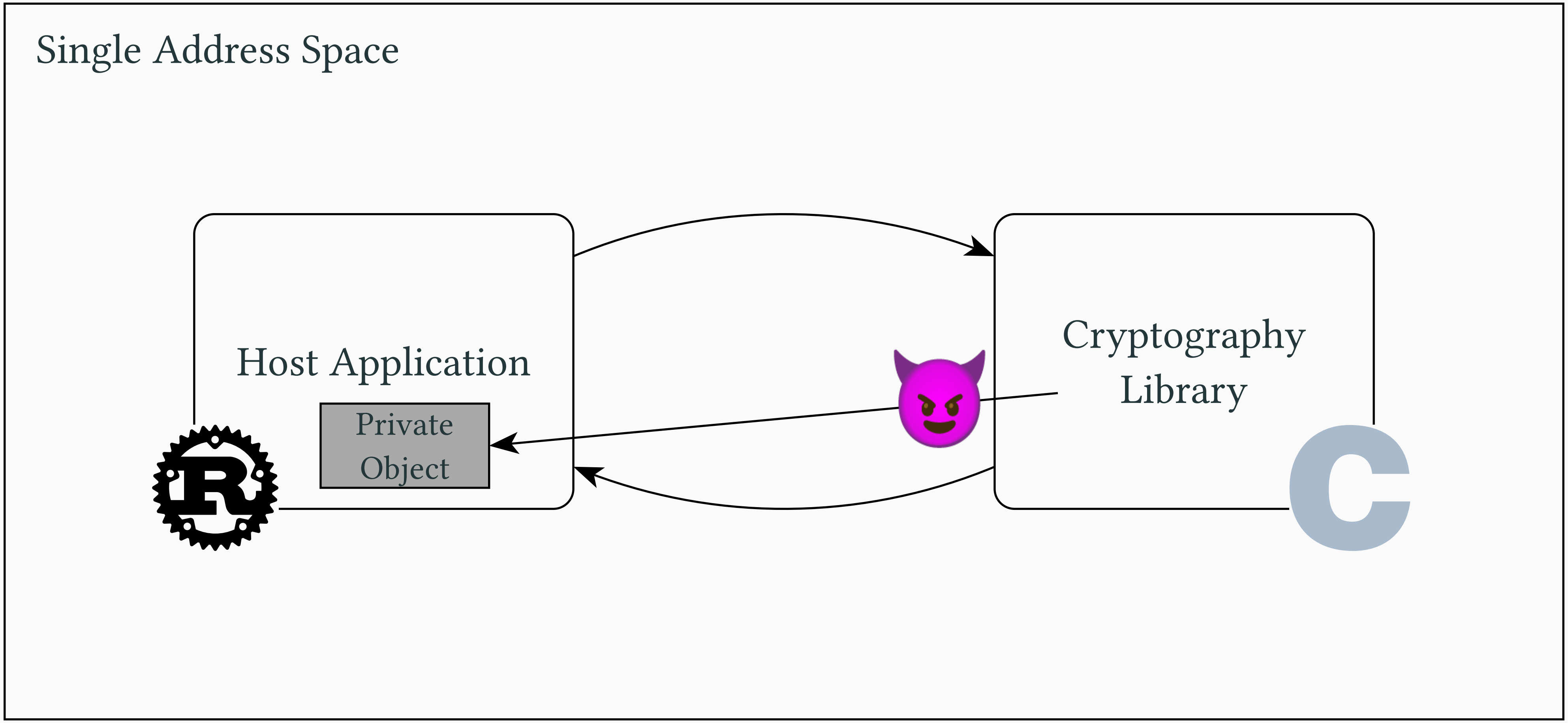
Valid Values

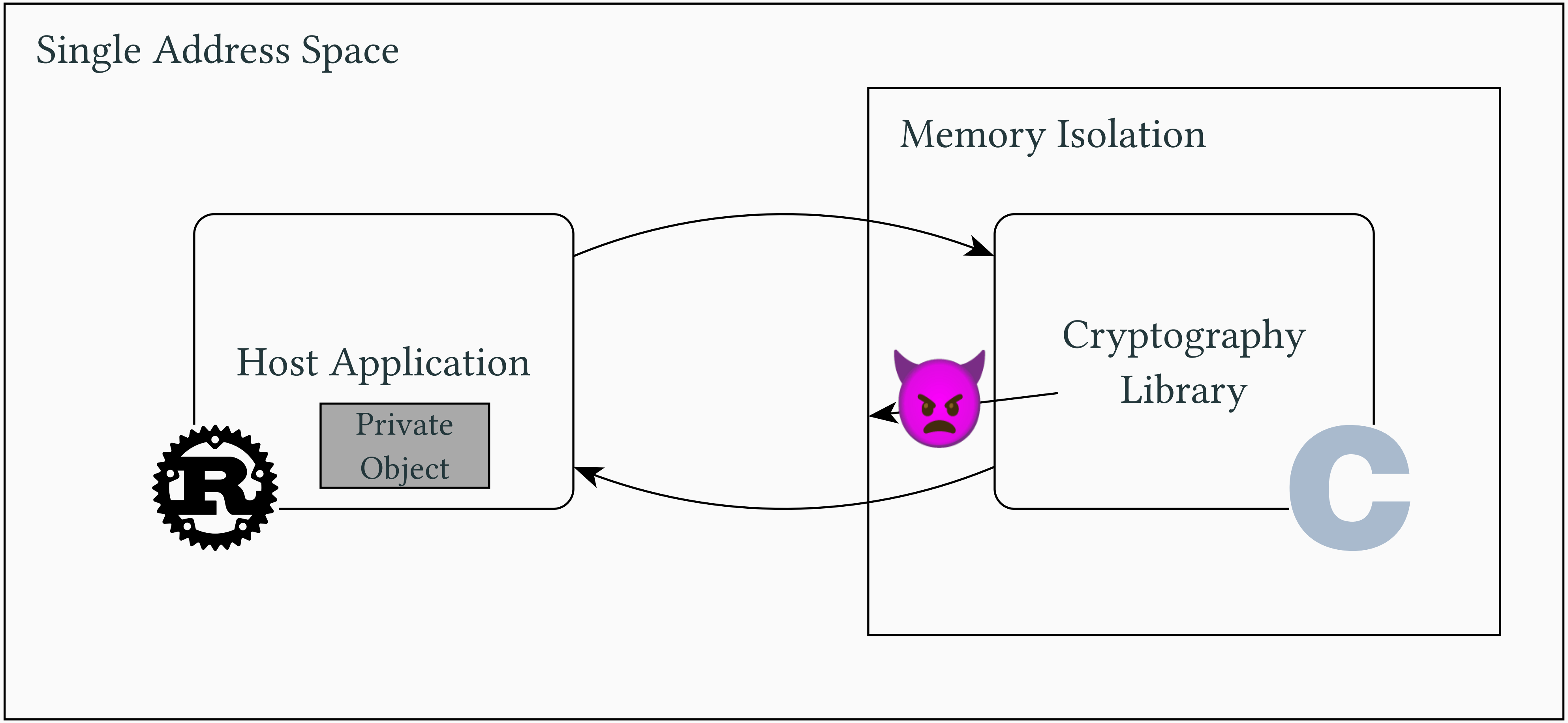
1. What makes interactions with foreign languages *safe*?

---

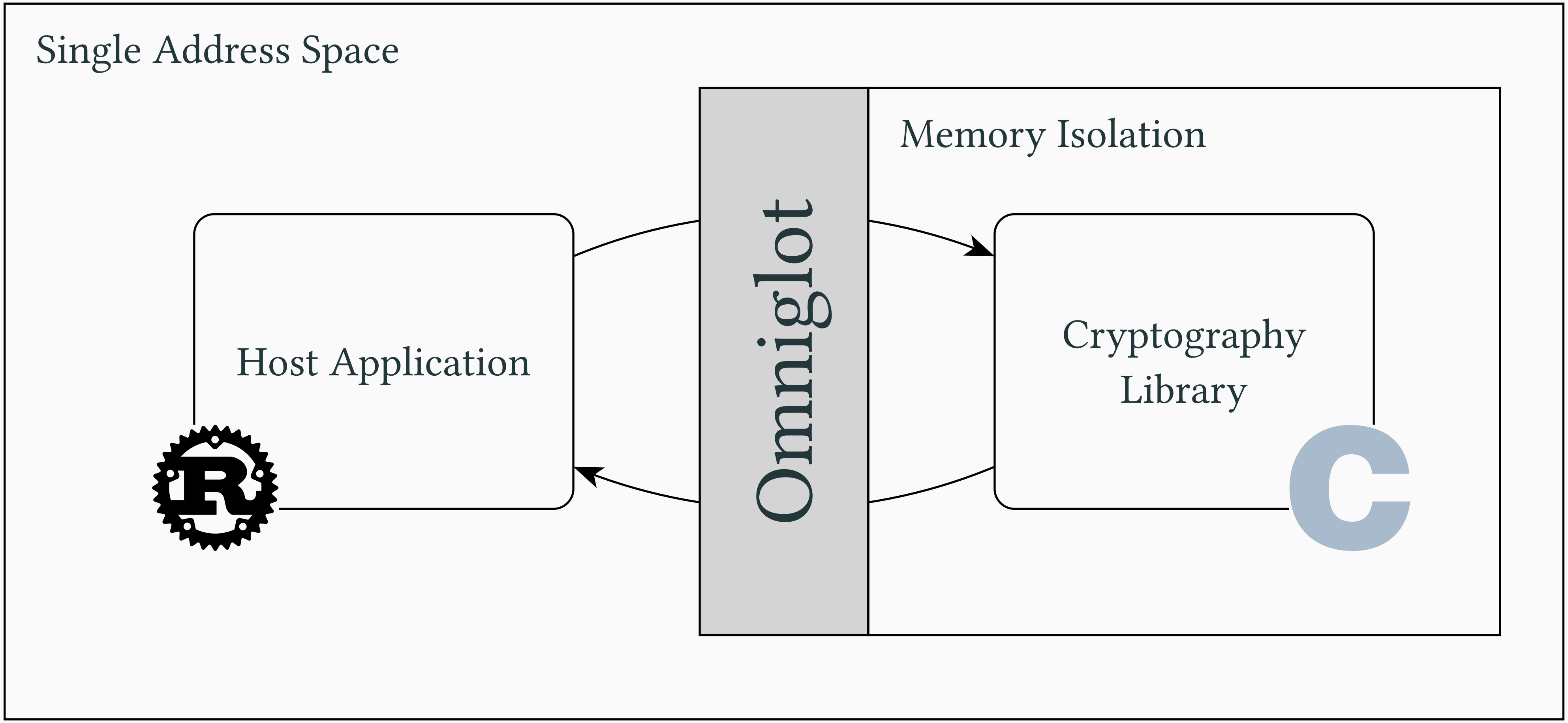
2. Safely integrate arbitrary, untrusted foreign libraries with Omniglot


Combined Program
Host Application
Cryptography Library

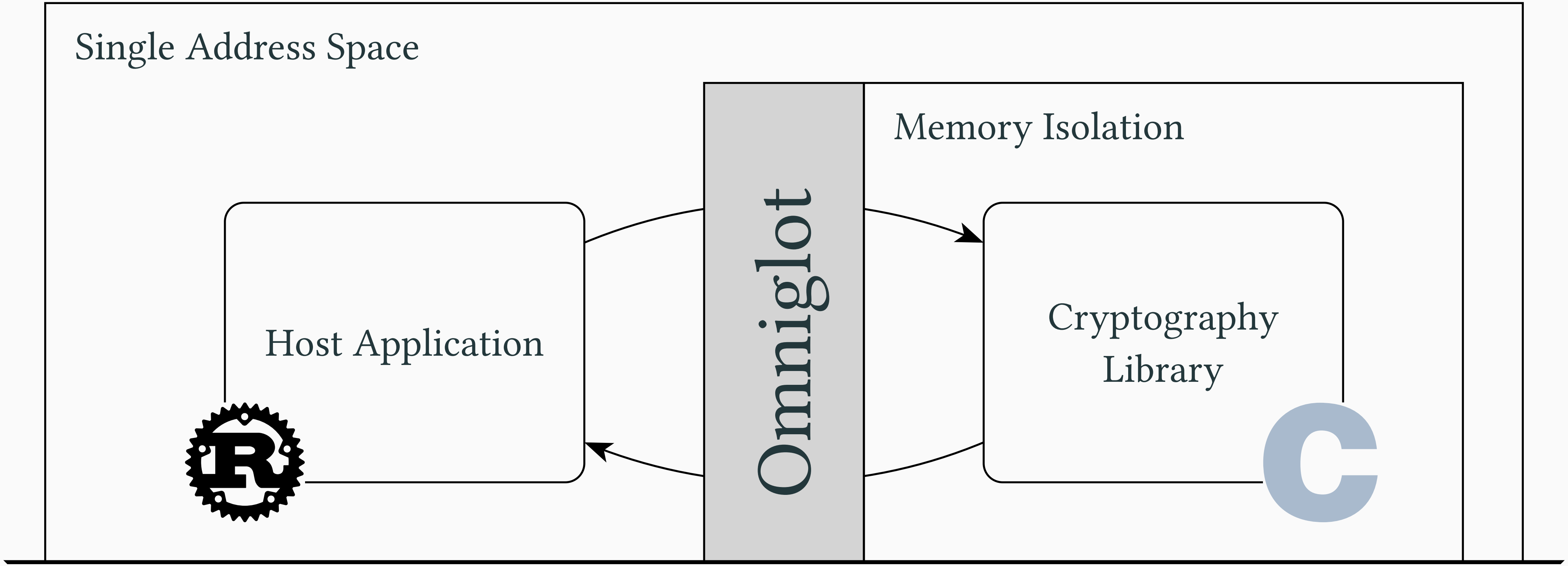

Single Address Space
Host Application
Private Object
Cryptography Library


Single Address Space
Memory Isolation
Host Application
Private Object
Cryptography Library

Safe Interactions with Foreign Languages through Omniglot



Safe Interactions with Foreign Languages through Omniglot



Omniglot mediates interactions between Rust and foreign code:

→ Validating values

→ Preventing mutable aliasing

Safe Interactions with Foreign Languages through Omniglot

Native, Unsafe FFI

Rust

```rust
unsafe { aes_encrypt(); }
```

Direct FFI call:

```rust
extern "C"
  fn aes_encrypt()
  -> bool
```

Foreign Library

C

```c
bool aes_encrypt() {
  return (bool) 2;
}
```

Native, Unsafe FFI (Rust)

```rust
unsafe { aes_encrypt(); }
```

Direct FFI call:

```rust
extern "C"
  fn aes_encrypt()
  -> bool
```

Foreign Library (C)

```c
bool aes_encrypt() {
  return (bool) 2;
}
```

```rust
extern "C"
  fn aes_encrypt'()
  -> c_char
```

Sandboxed FFI call through *weaker* function binding

Native, Unsafe FFI

Rust

```
unsafe { aes_encrypt(); }
```

Direct FFI call:

```
extern "C"
  fn aes_encrypt()
  -> bool
```

Foreign Library

C

```
bool aes_encrypt() {
  return (bool) 2;
}
```

Omniglot Wrapper

Rust

```
  let res: c_char = aes_encrypt'();
```

```
extern "C"
  fn aes_encrypt'()
  -> c_char
```

Sandboxed FFI call through *weaker* function binding

Safe Interactions with Foreign Languages through Omniglot

Native, Unsafe FFI

Rust

```rust
unsafe { aes_encrypt(); }
```

Direct FFI call:

```rust
extern "C"
  fn aes_encrypt()
  -> bool
```

Foreign Library

C

```c
bool aes_encrypt() {
  return (bool) 2;
}
```

Omniglot Wrapper

Rust

```rust
fn aes_encrypt_wrapped() -> Result<bool> {
  let res: c_char = aes_encrypt'();
  if res >= 0 && res < 2 {
    // `res` is a valid bool enum variant
    return Ok(bool::from(res));
  } else {
    return Err(OmniglotError::ValidationFail);
  }
}
```

```rust
extern "C"
  fn aes_encrypt'()
  -> c_char
```

Sandboxed FFI call through *weaker* function binding

Native, Unsafe FFI

Rust

```rust
unsafe { aes_encrypt(); }
```

Omniglot Bindings

```rust
aes_encrypt_wrapped();
```

Direct FFI call:

```rust
extern "C"
  fn aes_encrypt()
  -> bool
```

Foreign Library

C

```c
bool aes_encrypt() {
  return (bool) 2;
}
```

Omniglot Wrapper

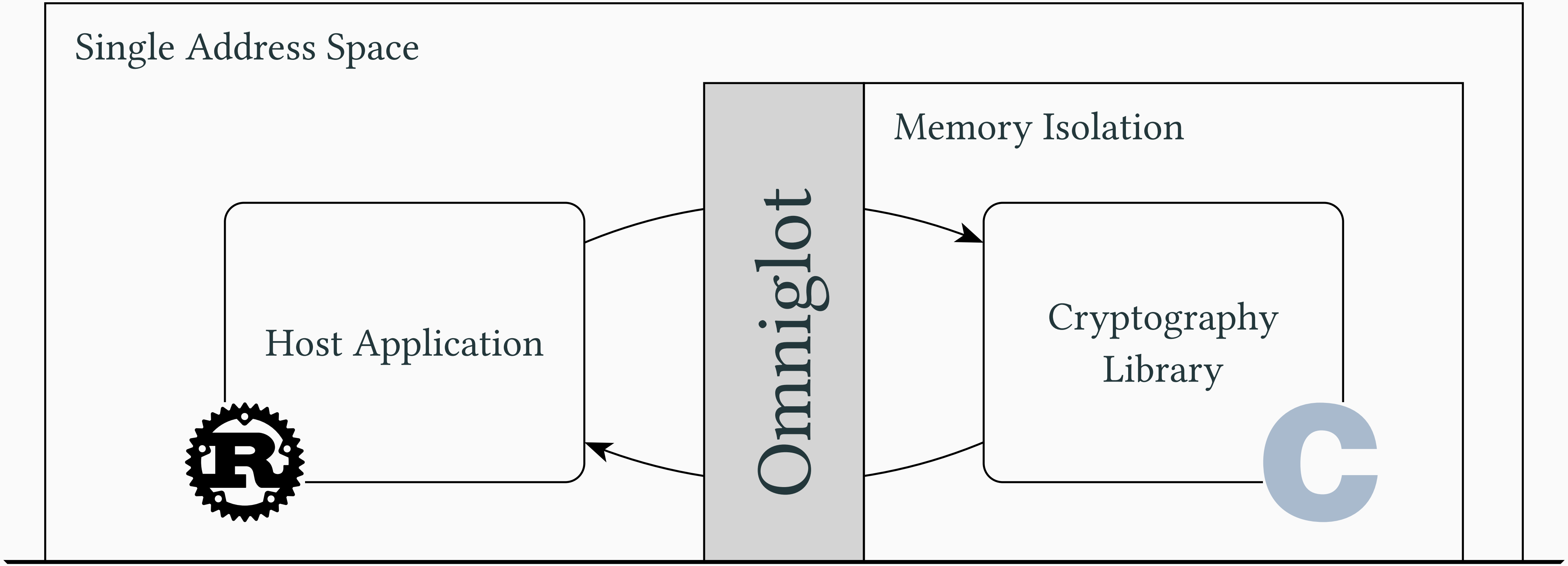
Rust

```rust
fn aes_encrypt_wrapped() -> Result<bool> {
  let res: c_char = aes_encrypt'();
  if res >= 0 && res < 2 {
    // `res` is a valid bool enum variant
    return Ok(bool::from(res));
  } else {
    return Err(OmniglotError::ValidationFail);
  }
}
```

```rust
extern "C"
  fn aes_encrypt'()
  -> c_char
```

Sandboxed FFI call through *weaker* function binding

Safe Interactions with Foreign Languages through Omniglot

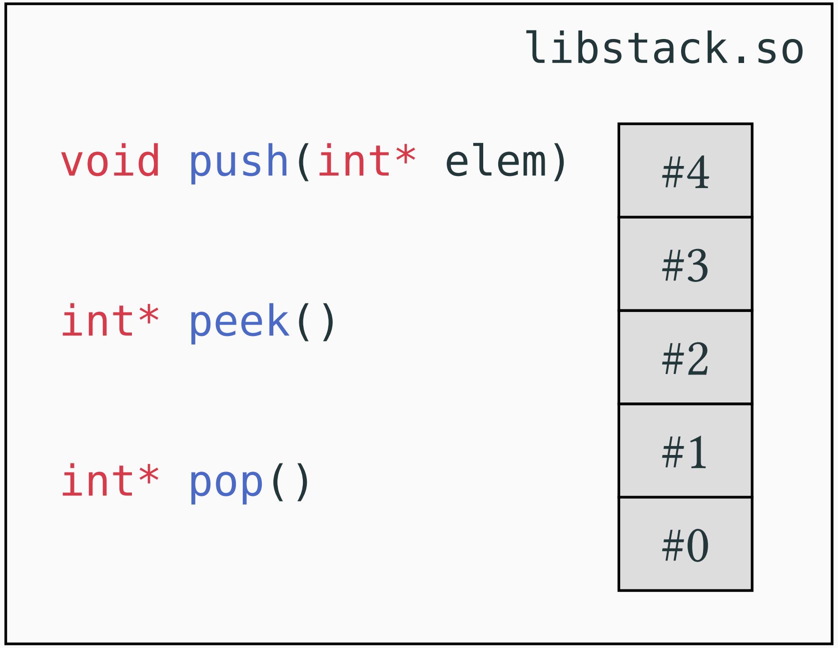


Omniglot mediates interactions between Rust and foreign code:

✅ Validating values → Preventing mutable aliasing

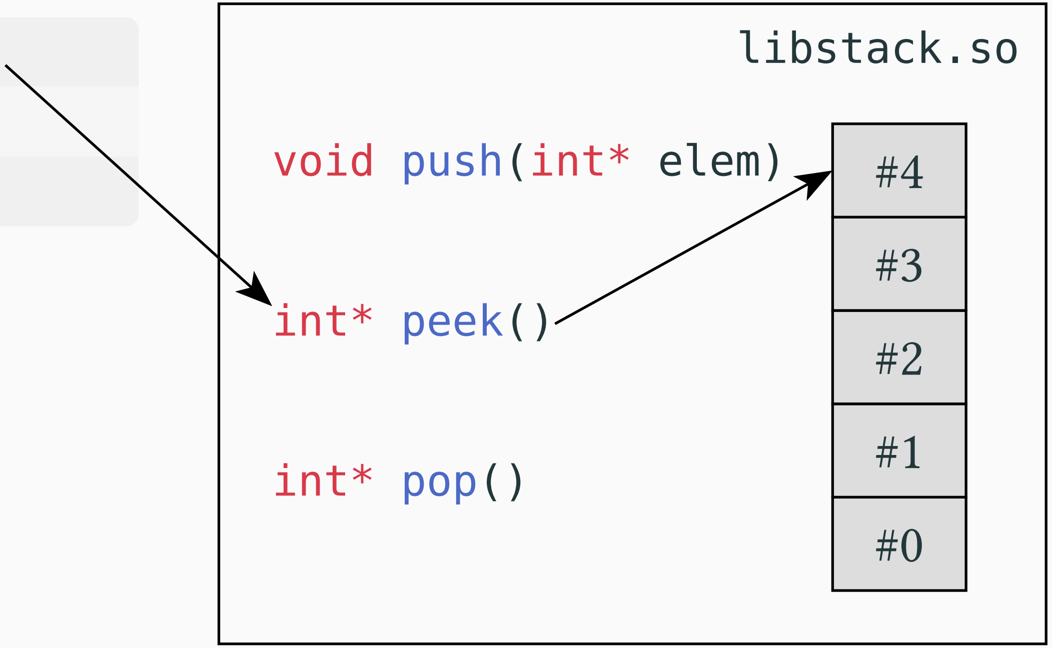
Aliasing $\oplus$ Mutability: there can never be two references pointing to overlapping memory, if at least one of them is mutable.

Aliasing ⊕ Mutability: there can never be two references pointing to overlapping memory, if at least one of them is mutable.



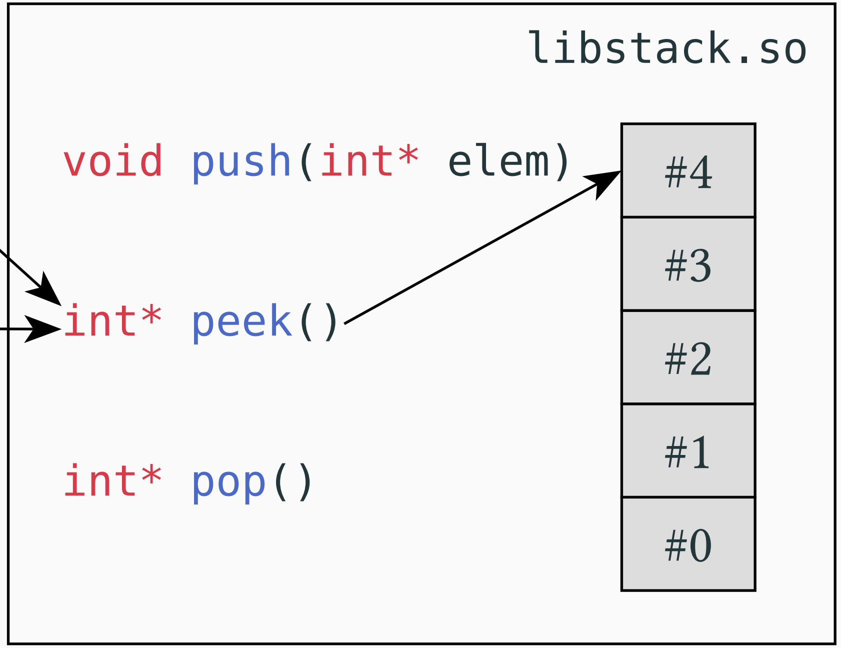
Aliasing ⊕ Mutability: there can never be two references pointing to overlapping memory, if at least one of them is mutable.

```
let ptr_a = peek(); // *mut #4
let ref_a = unsafe { &*ptr_a };
println!("{}", ref_a);
```



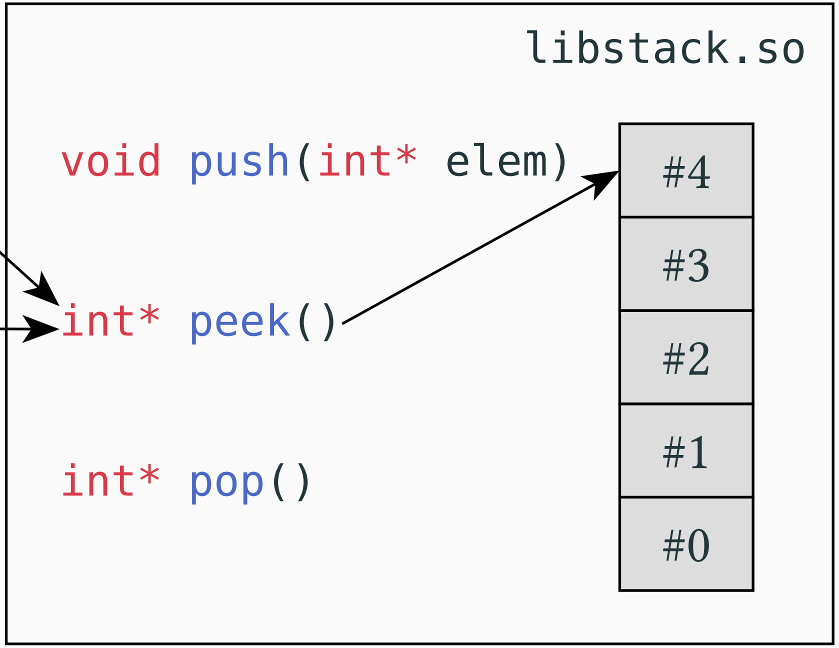
Aliasing ⊕ Mutability: there can never be two references pointing to overlapping memory, if at least one of them is mutable.

```rust
let ptr_a = peek(); // *mut #4
let ref_a = unsafe { &*ptr_a };
println!("{}", ref_a);

let ptr_b = peek(); // *mut #4
let ref_b = unsafe { &mut *ptr_b };
*ref_b = 42;
```



Aliasing ⊕ Mutability: there can never be two references pointing to overlapping memory, if at least one of them is mutable.

```
let ptr_a = peek(); // *mut #4
let ref_a = unsafe { &*ptr_a };
println!("{}", ref_a);

let ptr_b = peek(); // *mut #4
let ref_b = unsafe { &mut *ptr_b };
*ref_b = 42;

println!("{}", ref_a);
```

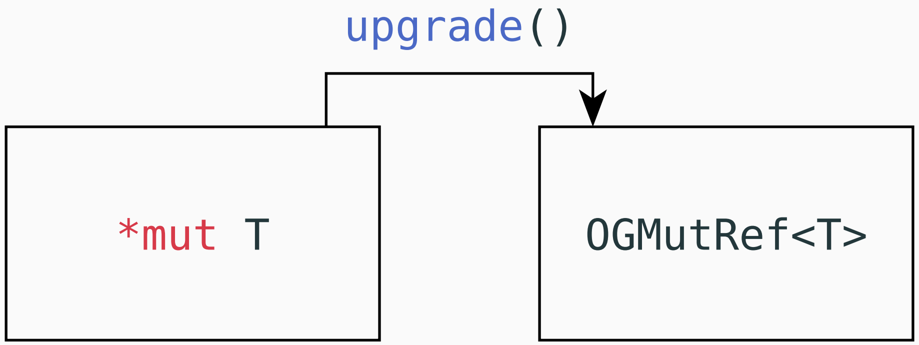


Omniglot Rejects Unsound Mutable Aliasing

`*mut T`

`*mut T`: Arbitrary Pointer into Foreign Memory

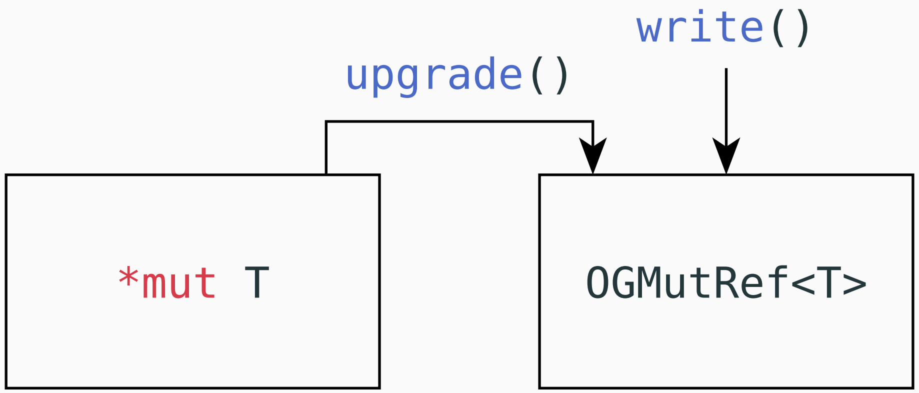
Omniglot Rejects Unsound Mutable Aliasing



`*mut T`: Arbitrary Pointer into Foreign Memory

`OGMutRef<T>`: Well-aligned, Mutably Accessible Object

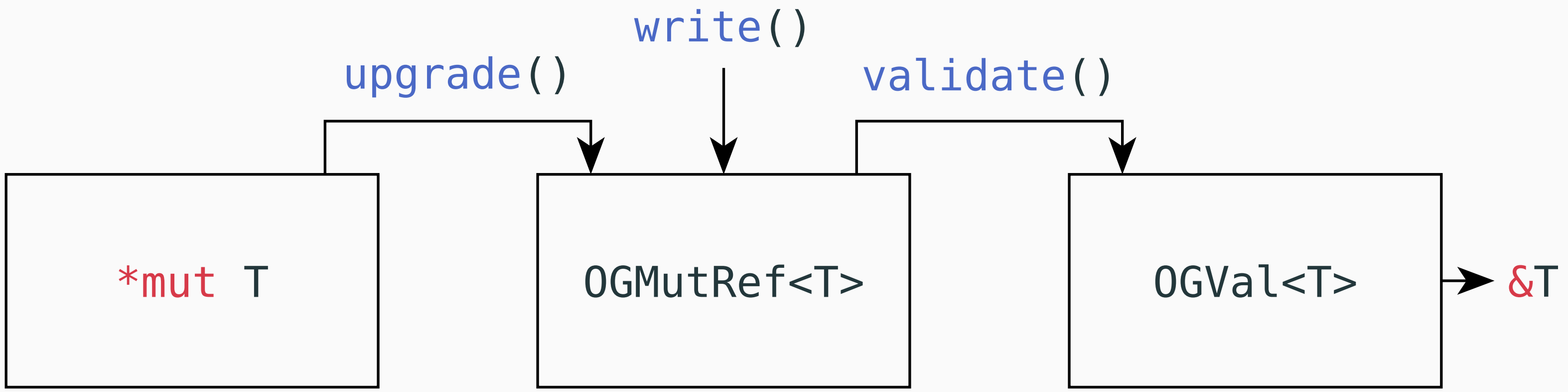
Omniglot Rejects Unsound Mutable Aliasing



`*mut T`: Arbitrary Pointer into Foreign Memory

`OGMutRef<T>`: Well-aligned, Mutably Accessible Object

Omniglot Rejects Unsound Mutable Aliasing

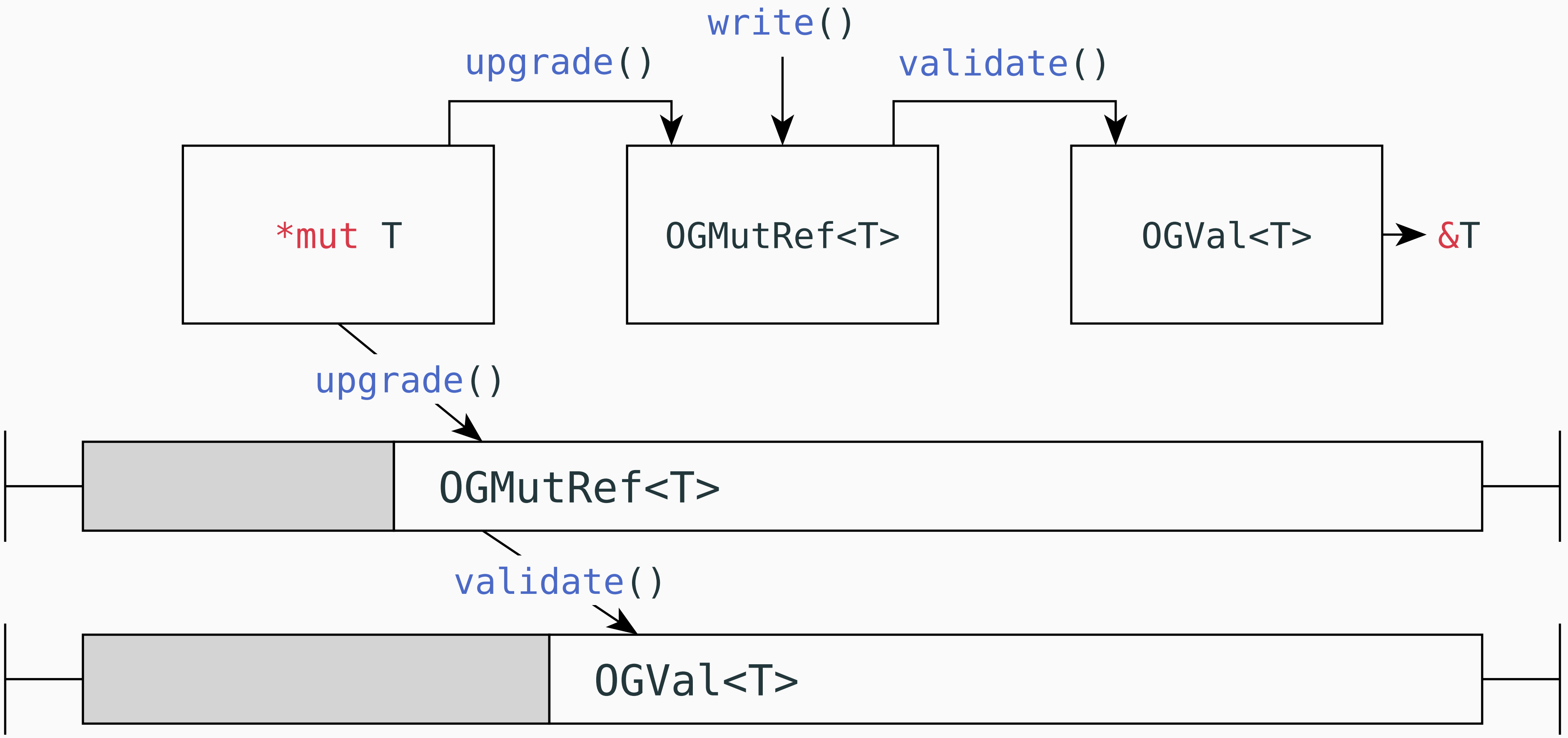


`*mut T`: Arbitrary Pointer into Foreign Memory

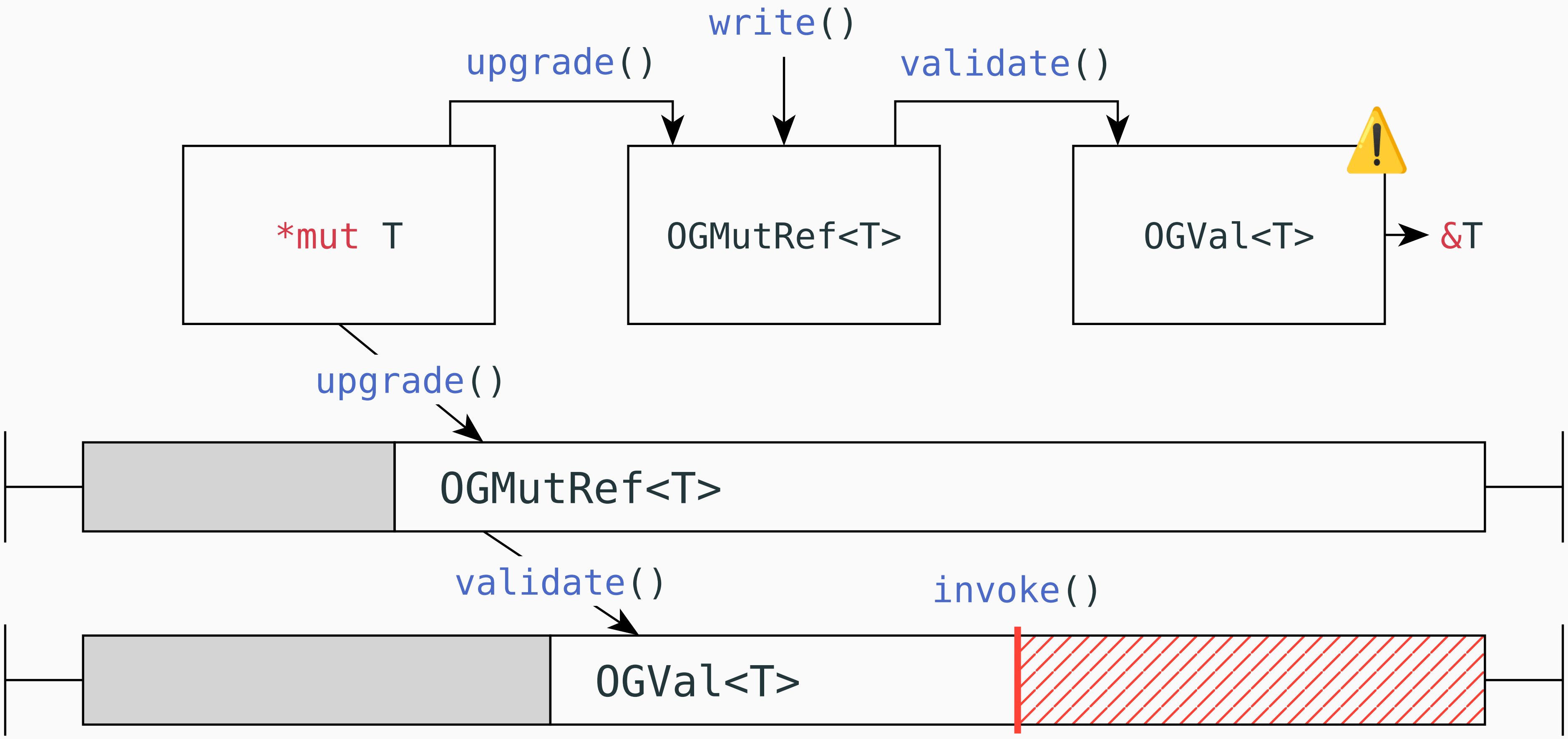
`OGMutRef<T>`: Well-aligned, Mutably Accessible Object

`OGVal<T>`: Object Conforming to Rust's Requirements on Valid Values

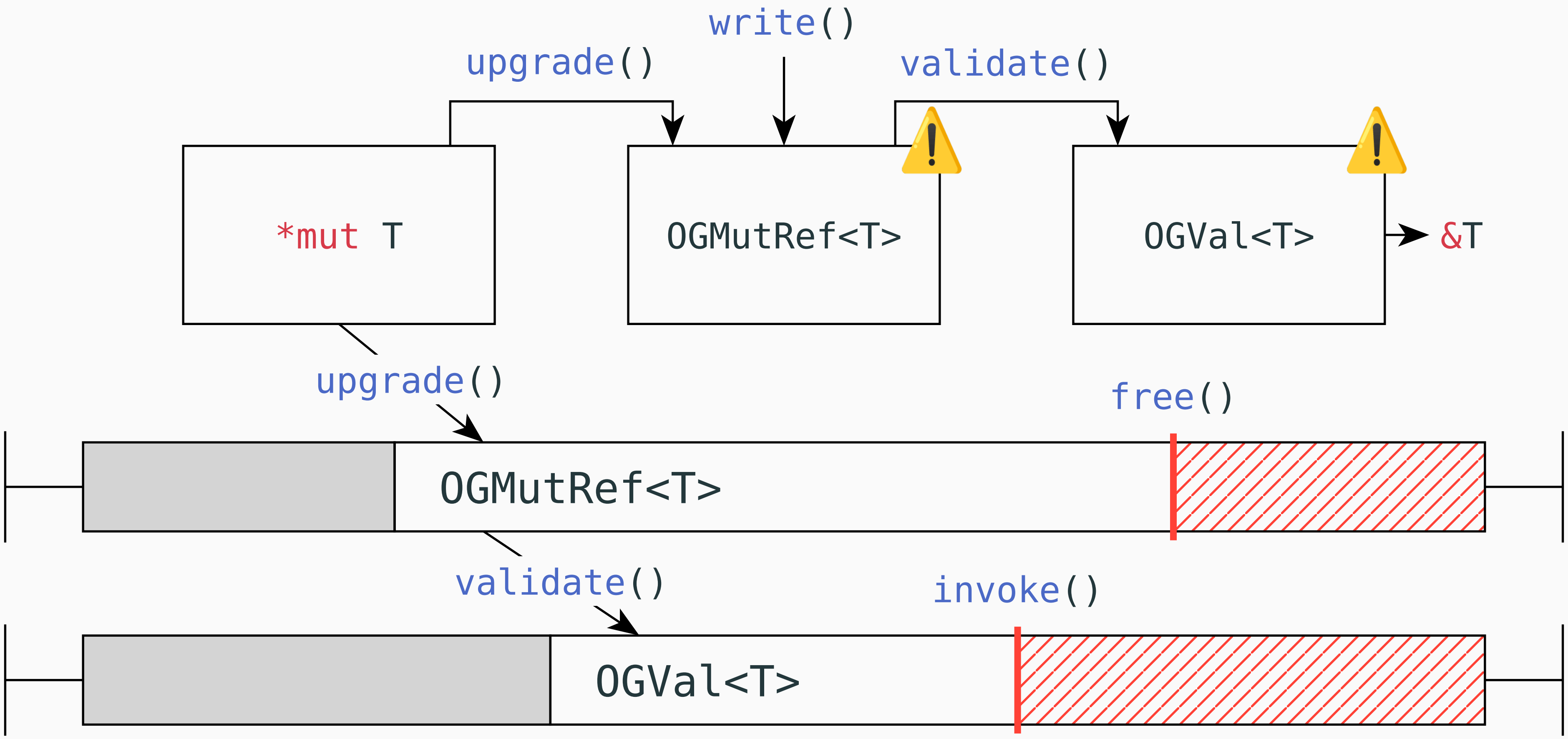
Omniglot Rejects Unsound Mutable Aliasing



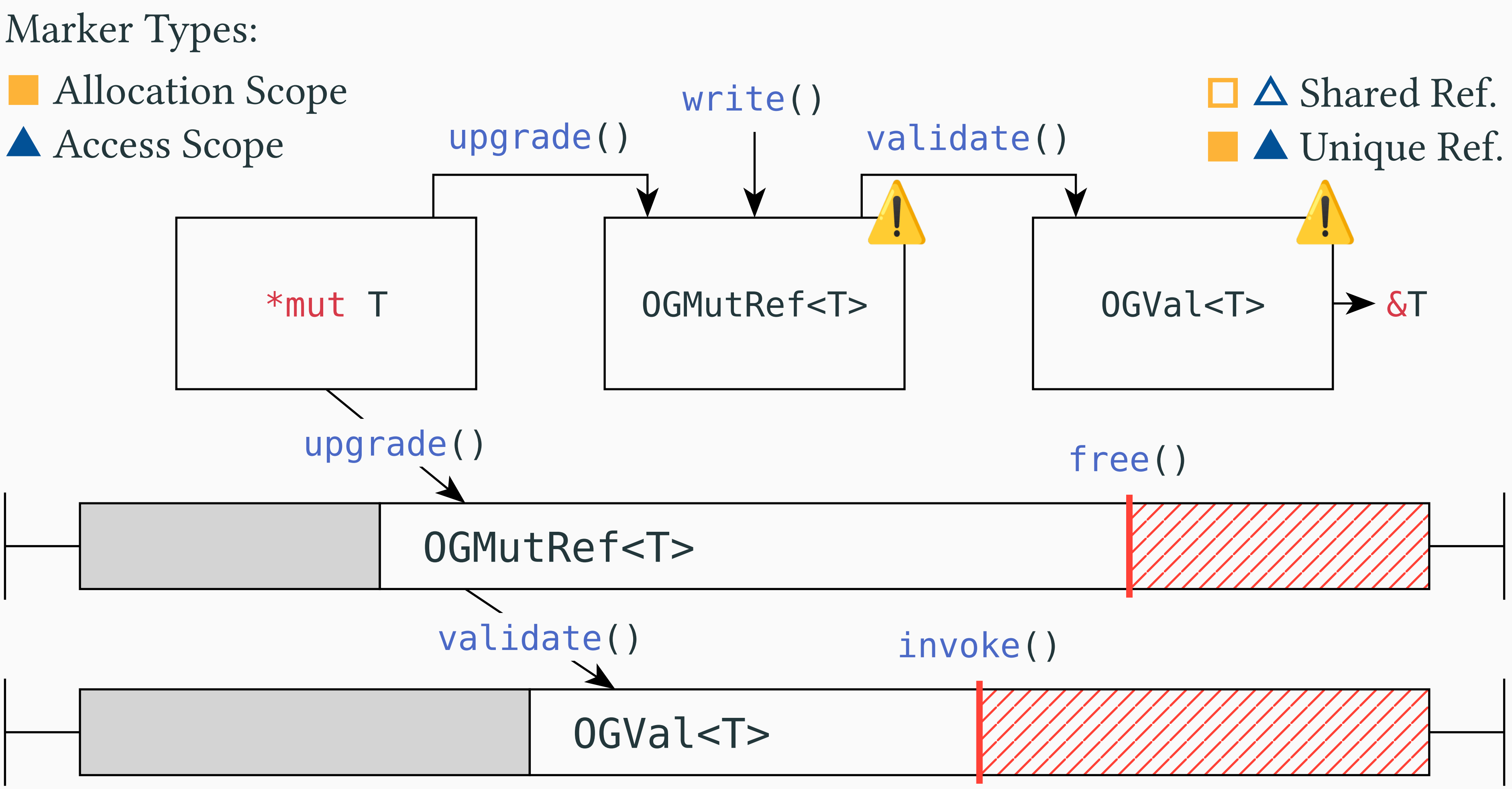
Omniglot Rejects Unsound Mutable Aliasing



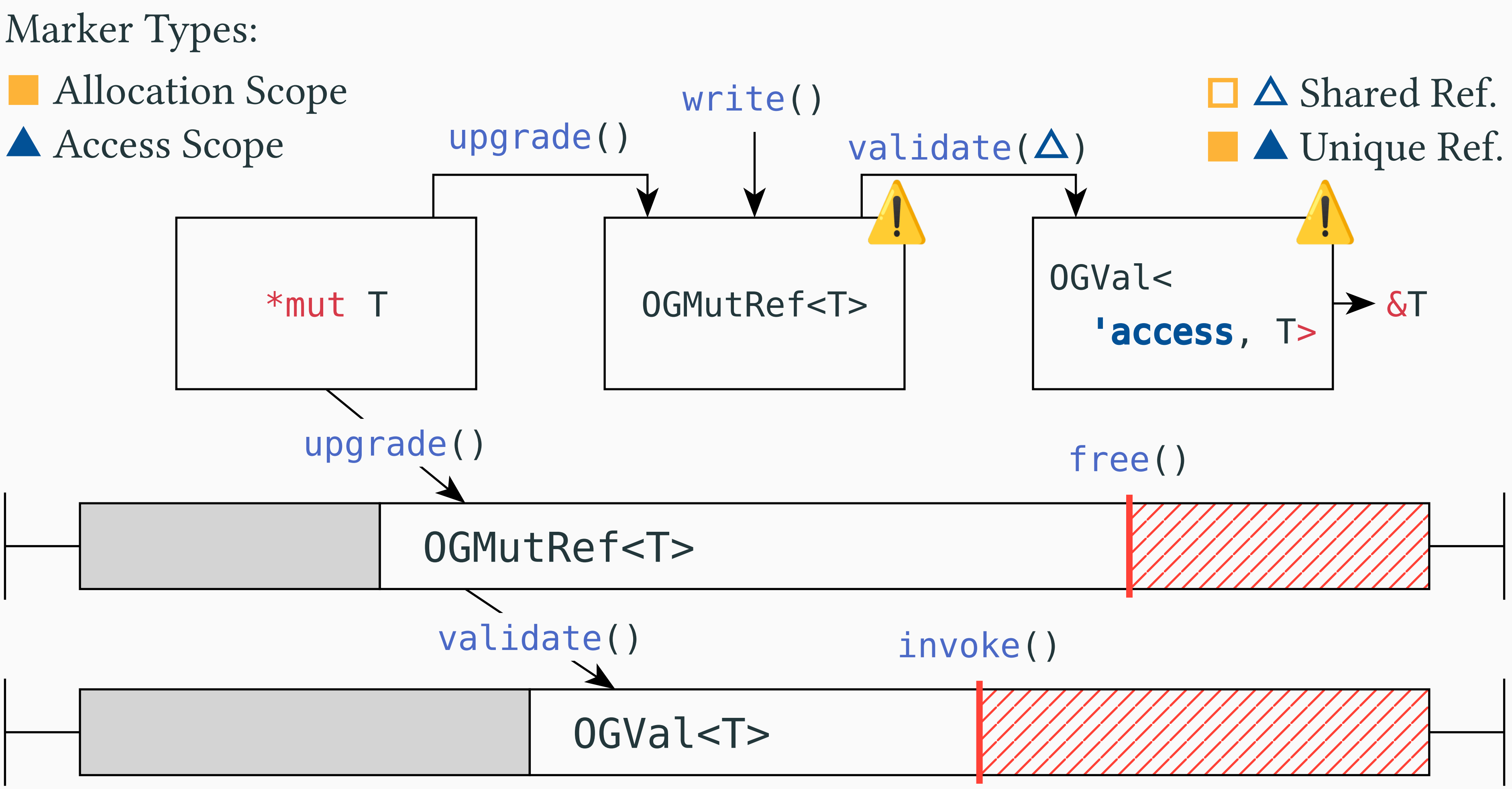
Omniglot Rejects Unsound Mutable Aliasing



Omniglot Rejects Unsound Mutable Aliasing



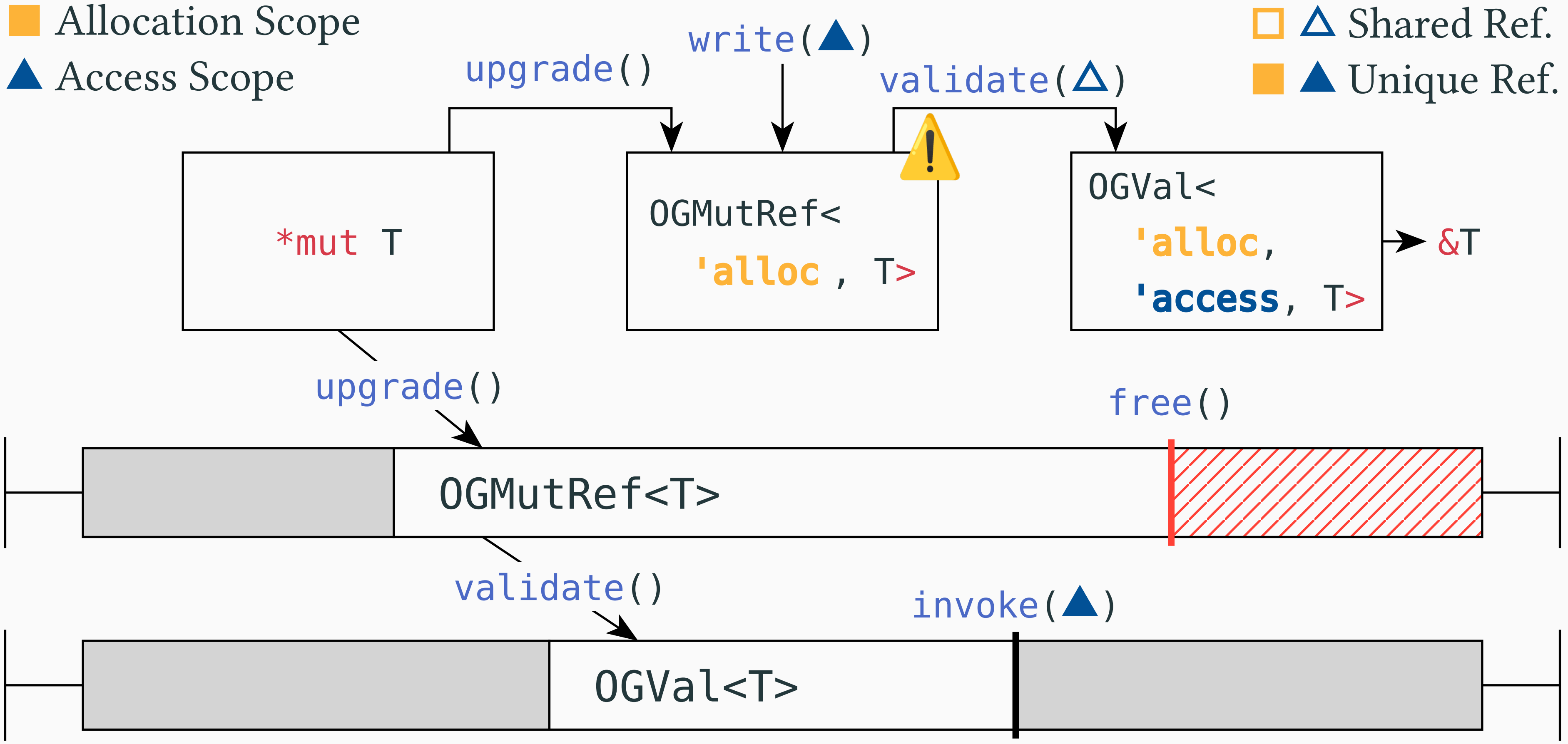
Omniglot Rejects Unsound Mutable Aliasing



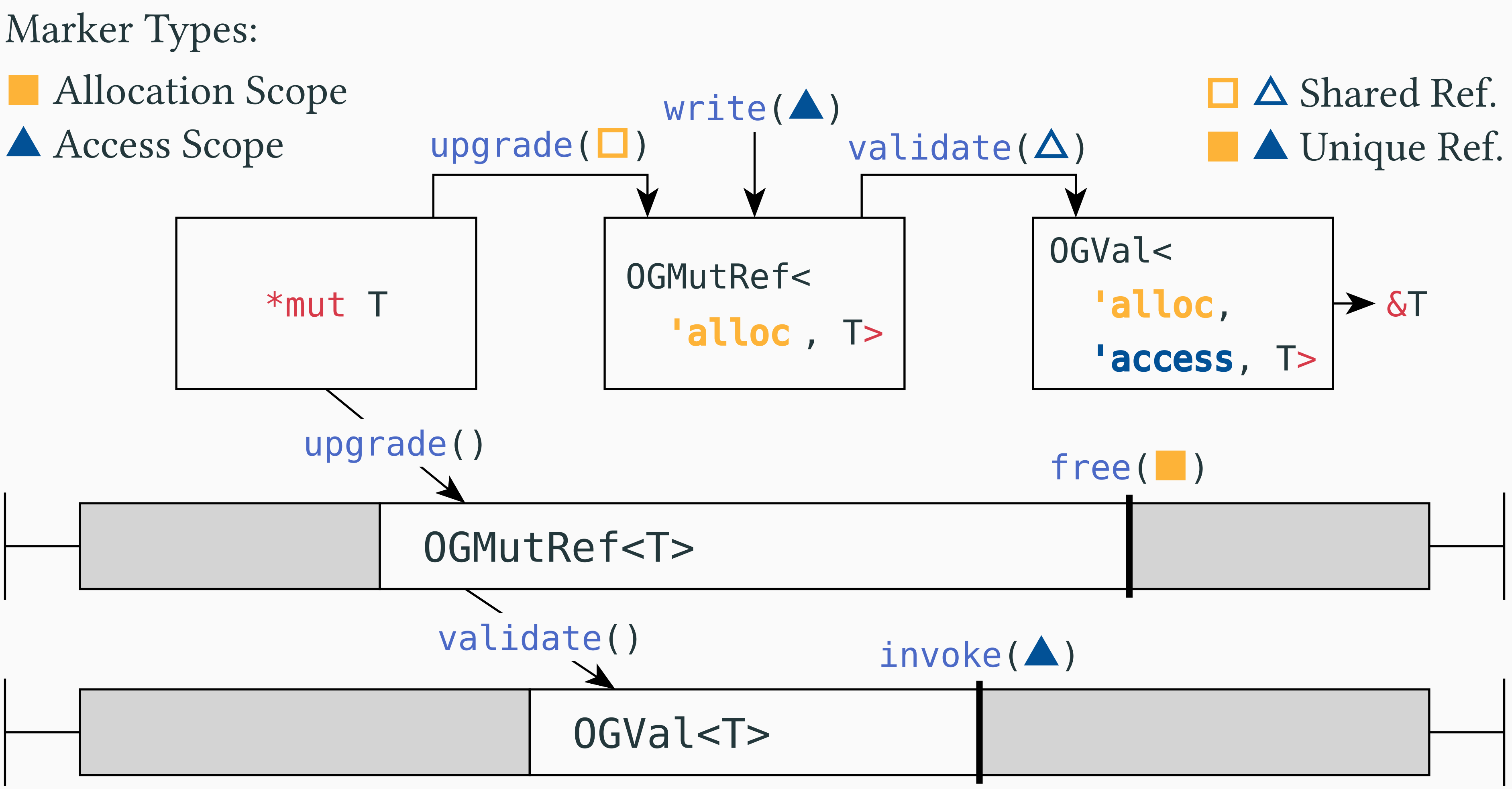
Omniglot Rejects Unsound Mutable Aliasing

Marker Types:

- Allocation Scope
- Access Scope

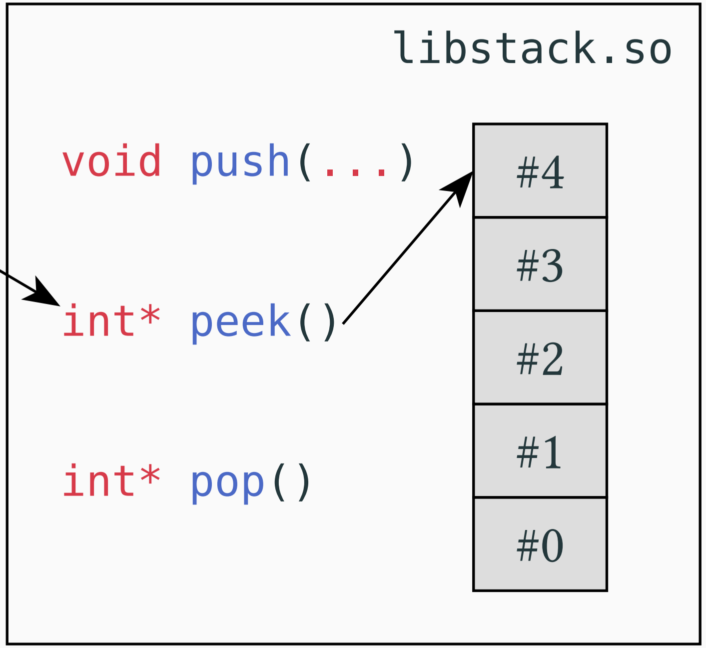


Omniglot Rejects Unsound Mutable Aliasing

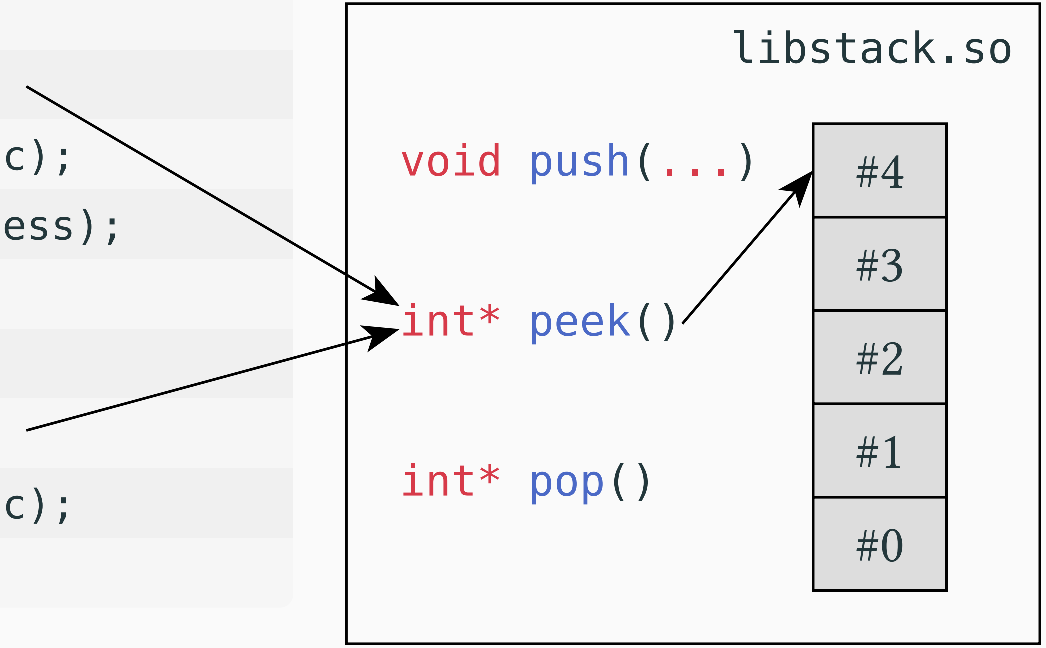


Omniglot Rejects Unsound Mutable Aliasing

```
let (alloc, access) = scopes!();

let ptr_a = peek(); // *mut #4
let ref_a = ptr_a.upgrade(&alloc);
let val_a = ptr_a.validate(&access);
println!("{}", val_a);
```

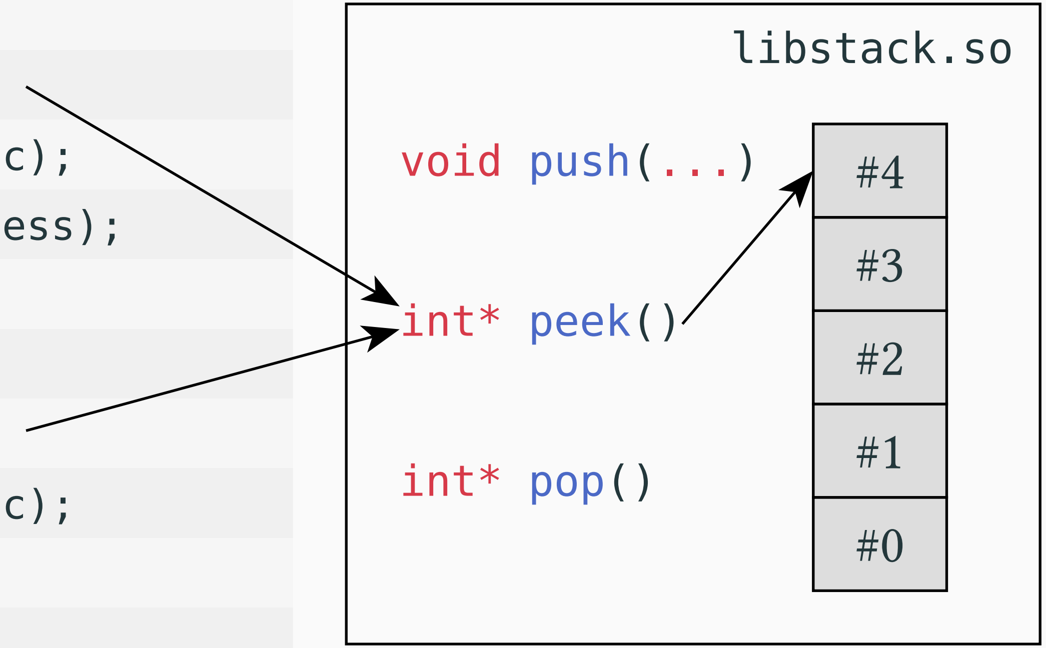


```
let (alloc, access) = scopes!();

let ptr_a = peek(); // *mut #4
let ref_a = ptr_a.upgrade(&alloc);
let val_a = ptr_a.validate(&access);
println!("{}", val_a);

let ptr_b = peek(); // *mut #4
let ref_b = ptr_b.upgrade(&alloc);
ref_b.write(42, &mut access);
```

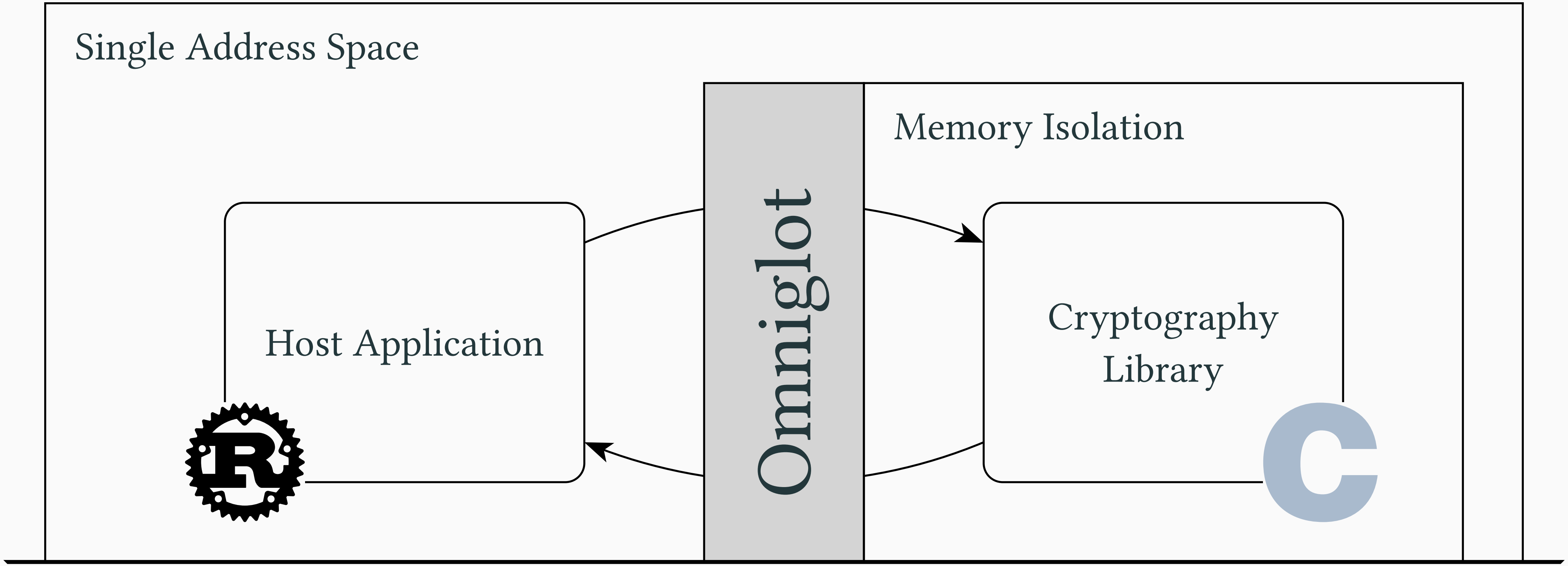


Omniglot Rejects Unsound Mutable Aliasing

```
let (alloc, access) = scopes!();

let ptr_a = peek(); // *mut #4
let ref_a = ptr_a.upgrade(&alloc);
let val_a = ptr_a.validate(&access);
println!("{}", val_a);

let ptr_b = peek(); // *mut #4
let ref_b = ptr_b.upgrade(&alloc);
ref_b.write(42, &mut access);

println!("{}", ref_a); // compile error!
```



Omniglot Enables Safe Interactions Between Rust and Foreign Code



Omniglot mediates interactions between Rust and foreign code:

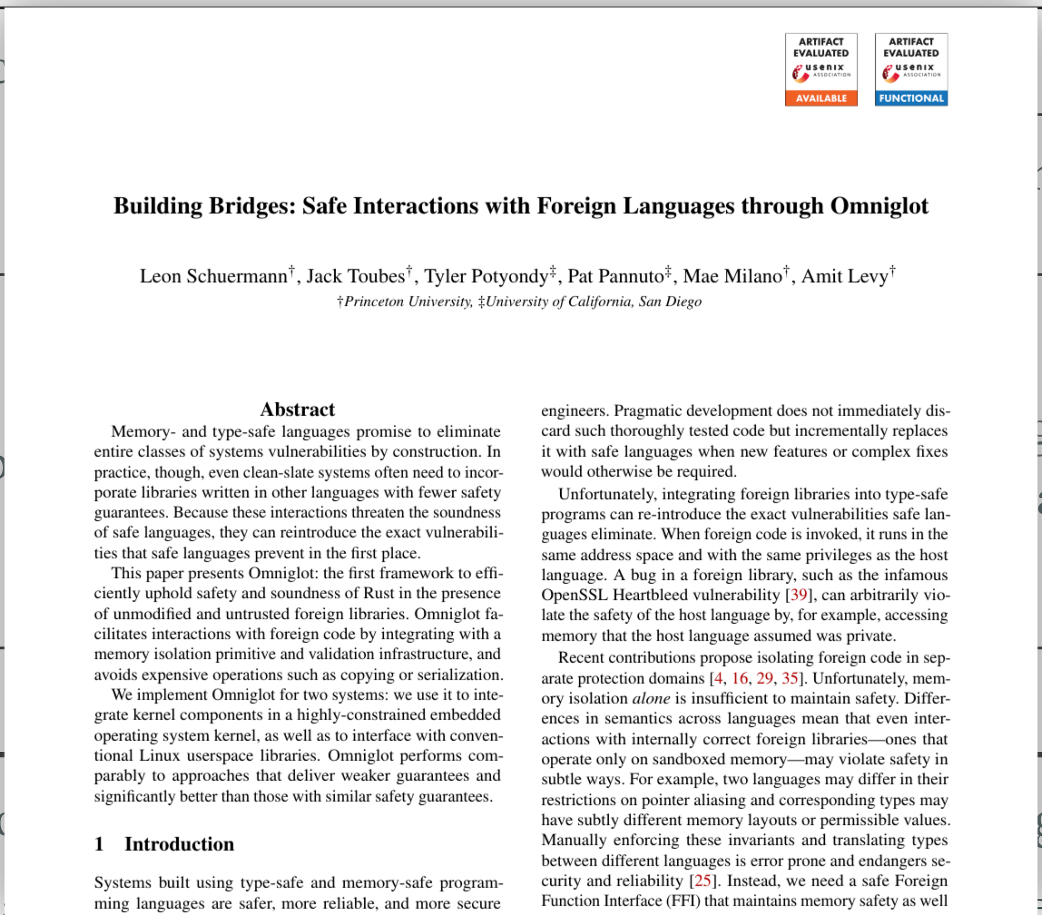
✅ Validating values

✅ Preventing mutable aliasing

Omniglot Enables Safe Interactions Between Rust and Foreign Code

Single Address Spac

Host Ap


ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Building Bridges: Safe Interactions with Foreign Languages through Omniglot

Leon Schuermann[†], Jack Toubes[†], Tyler Potyondy[‡], Pat Pannuto[‡], Mae Milano[†], Amit Levy[†]
†Princeton University, ‡University of California, San Diego

Abstract

Memory- and type-safe languages promise to eliminate entire classes of systems vulnerabilities by construction. In practice, though, even clean-slate systems often need to incorporate libraries written in other languages with fewer safety guarantees. Because these interactions threaten the soundness of safe languages, they can reintroduce the exact vulnerabilities that safe languages prevent in the first place.

This paper presents Omniglot: the first framework to efficiently uphold safety and soundness of Rust in the presence of unmodified and untrusted foreign libraries. Omniglot facilitates interactions with foreign code by integrating with a memory isolation primitive and validation infrastructure, and avoids expensive operations such as copying or serialization.

We implement Omniglot for two systems: we use it to integrate kernel components in a highly-constrained embedded operating system kernel, as well as to interface with conventional Linux userspace libraries. Omniglot performs comparably to approaches that deliver weaker guarantees and significantly better than those with similar safety guarantees.

1 Introduction

Systems built using type-safe and memory-safe programming languages are safer, more reliable, and more secure engineers. Pragmatic development does not immediately discard such thoroughly tested code but incrementally replaces it with safe languages when new features or complex fixes would otherwise be required.

Unfortunately, integrating foreign libraries into type-safe programs can re-introduce the exact vulnerabilities safe languages eliminate. When foreign code is invoked, it runs in the same address space and with the same privileges as the host language. A bug in a foreign library, such as the infamous OpenSSL Heartbleed vulnerability [39], can arbitrarily violate the safety of the host language by, for example, accessing memory that the host language assumed was private.

Recent contributions propose isolating foreign code in separate protection domains [4, 16, 29, 35]. Unfortunately, memory isolation *alone* is insufficient to maintain safety. Differences in semantics across languages mean that even interactions with internally correct foreign libraries—ones that operate only on sandboxed memory—may violate safety in subtle ways. For example, two languages may differ in their restrictions on pointer aliasing and corresponding types may have subtly different memory layouts or permissible values. Manually enforcing these invariants and translating types between different languages is error prone and endangers security and reliability [25]. Instead, we need a safe Foreign Function Interface (FFI) that maintains memory safety as well

n

graphy

ary

Omnigl

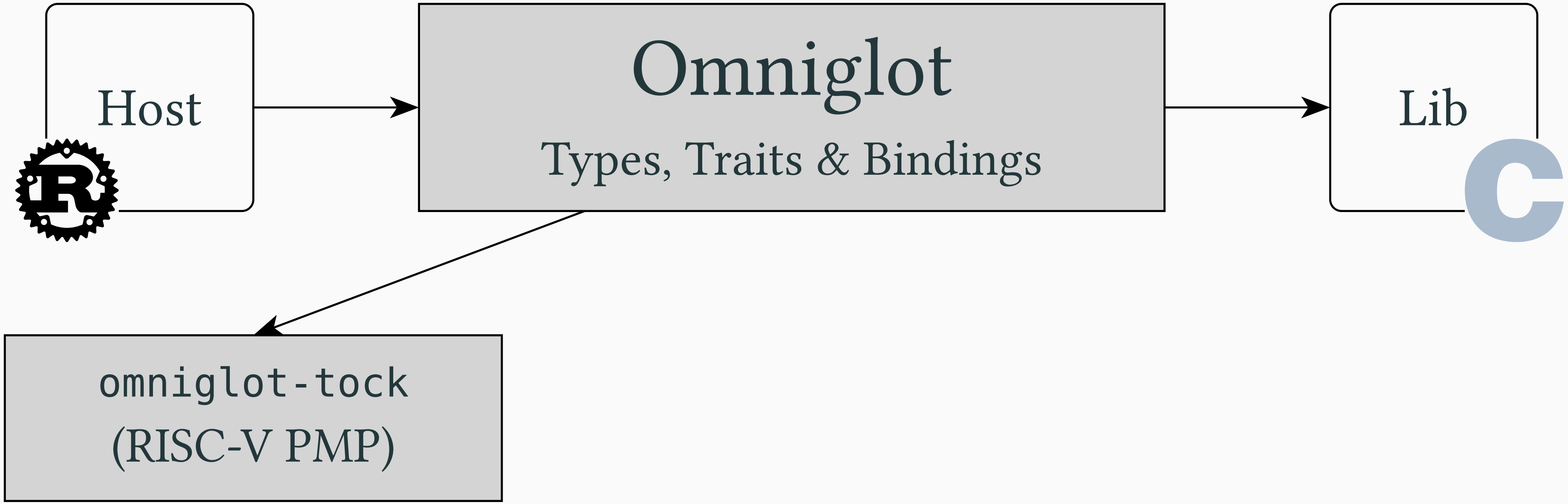
gn code:

✅ Valid

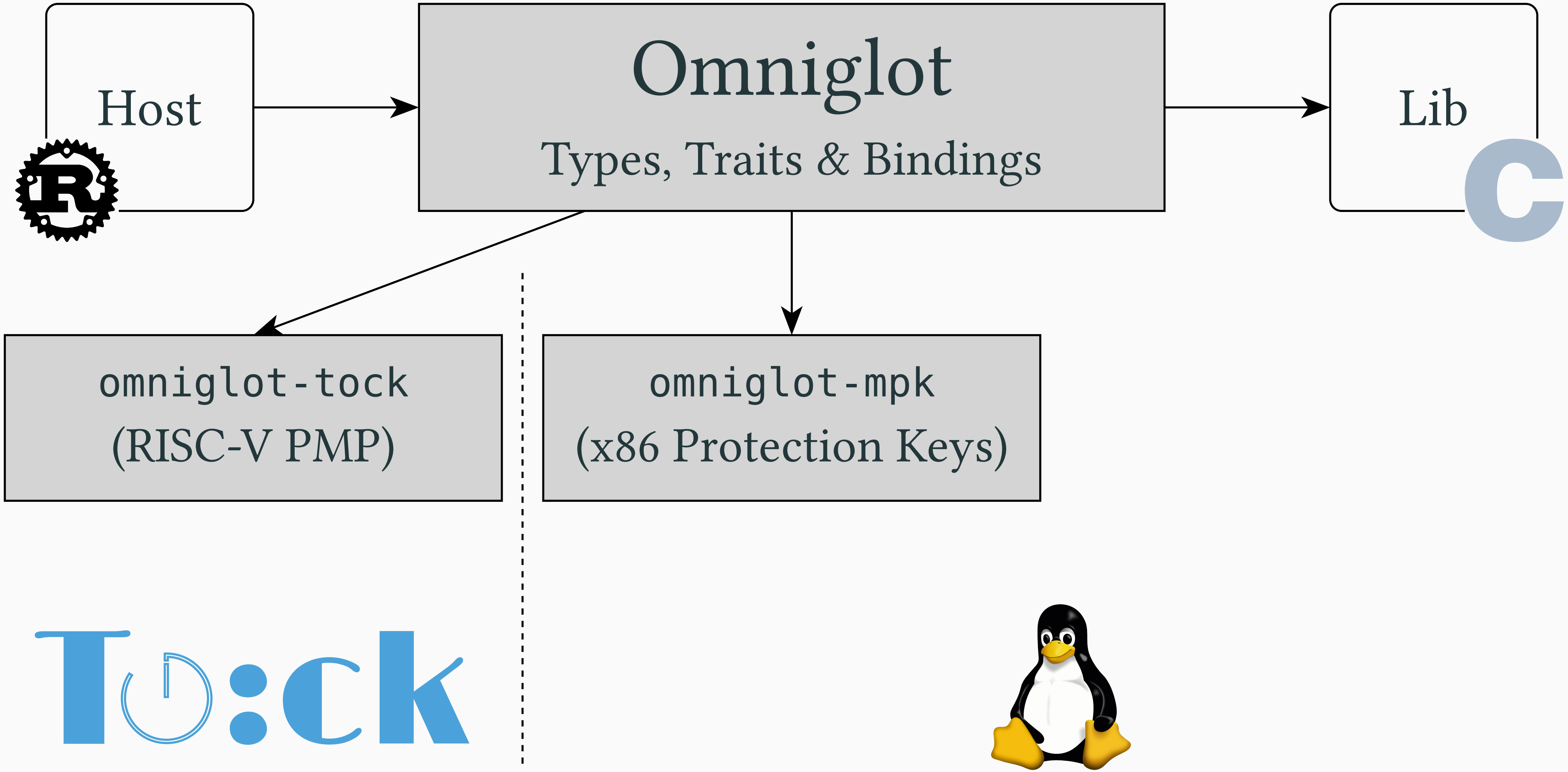
aliasing

Omniglot is General

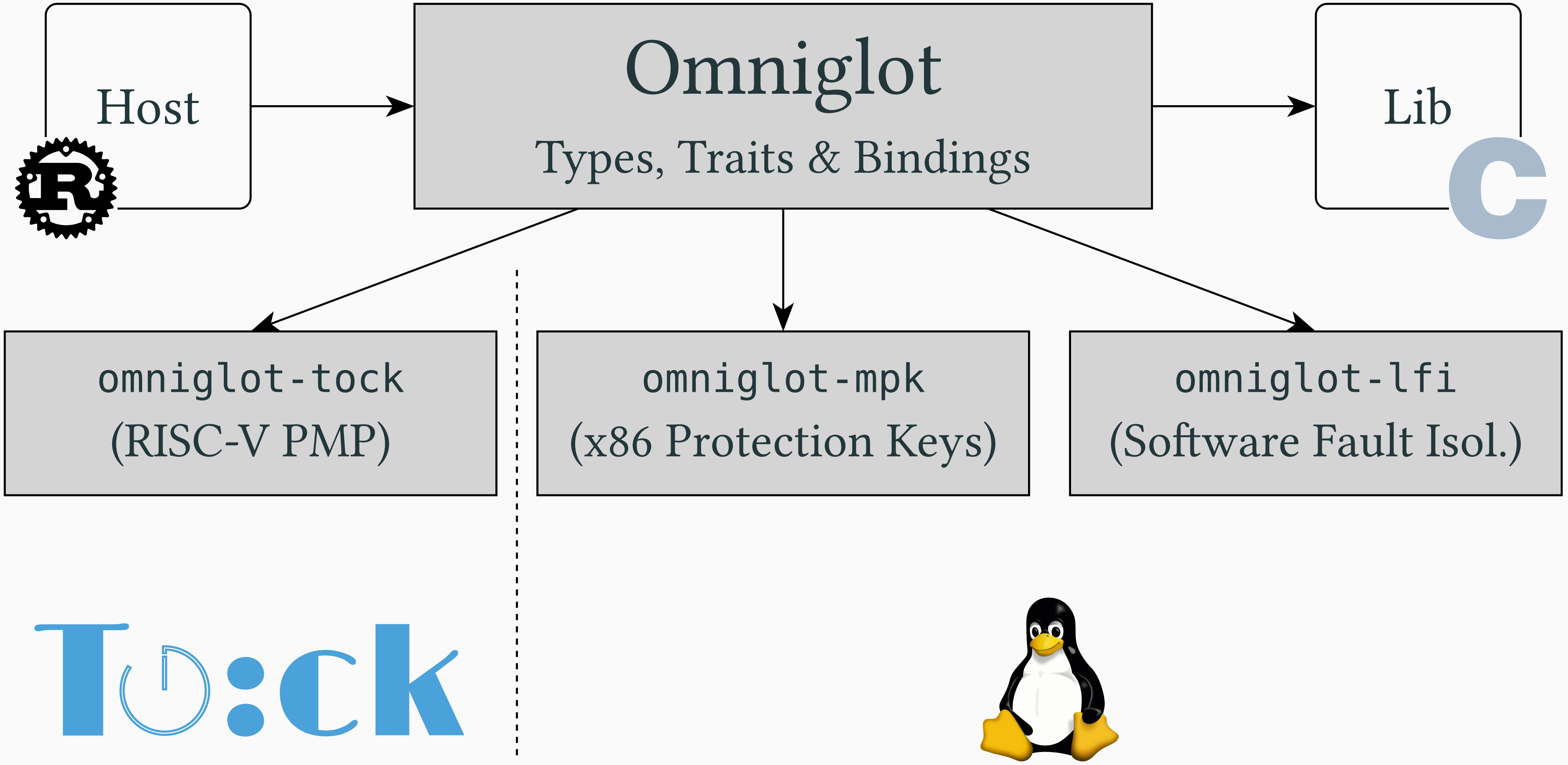
Omniglot is General

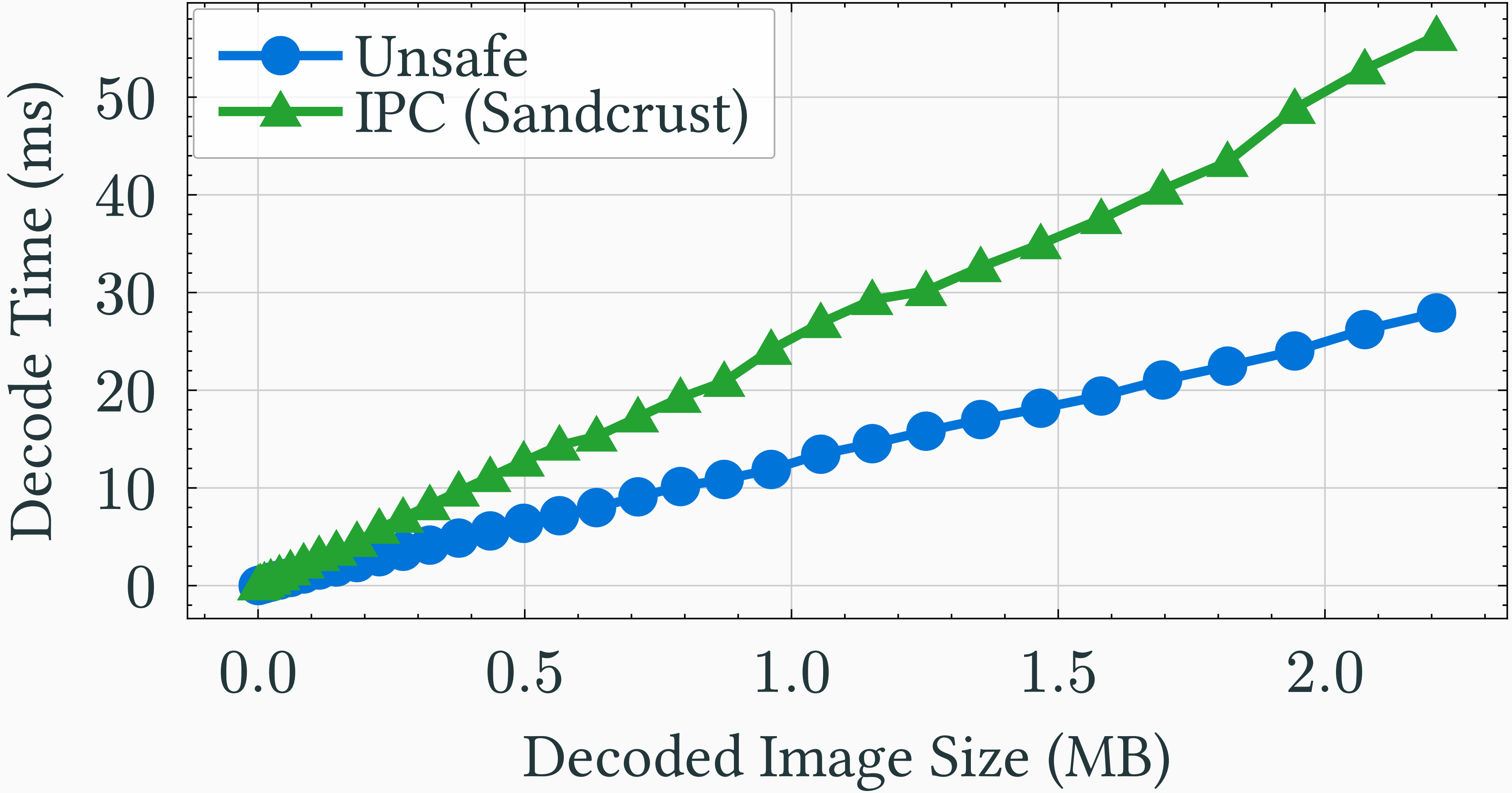


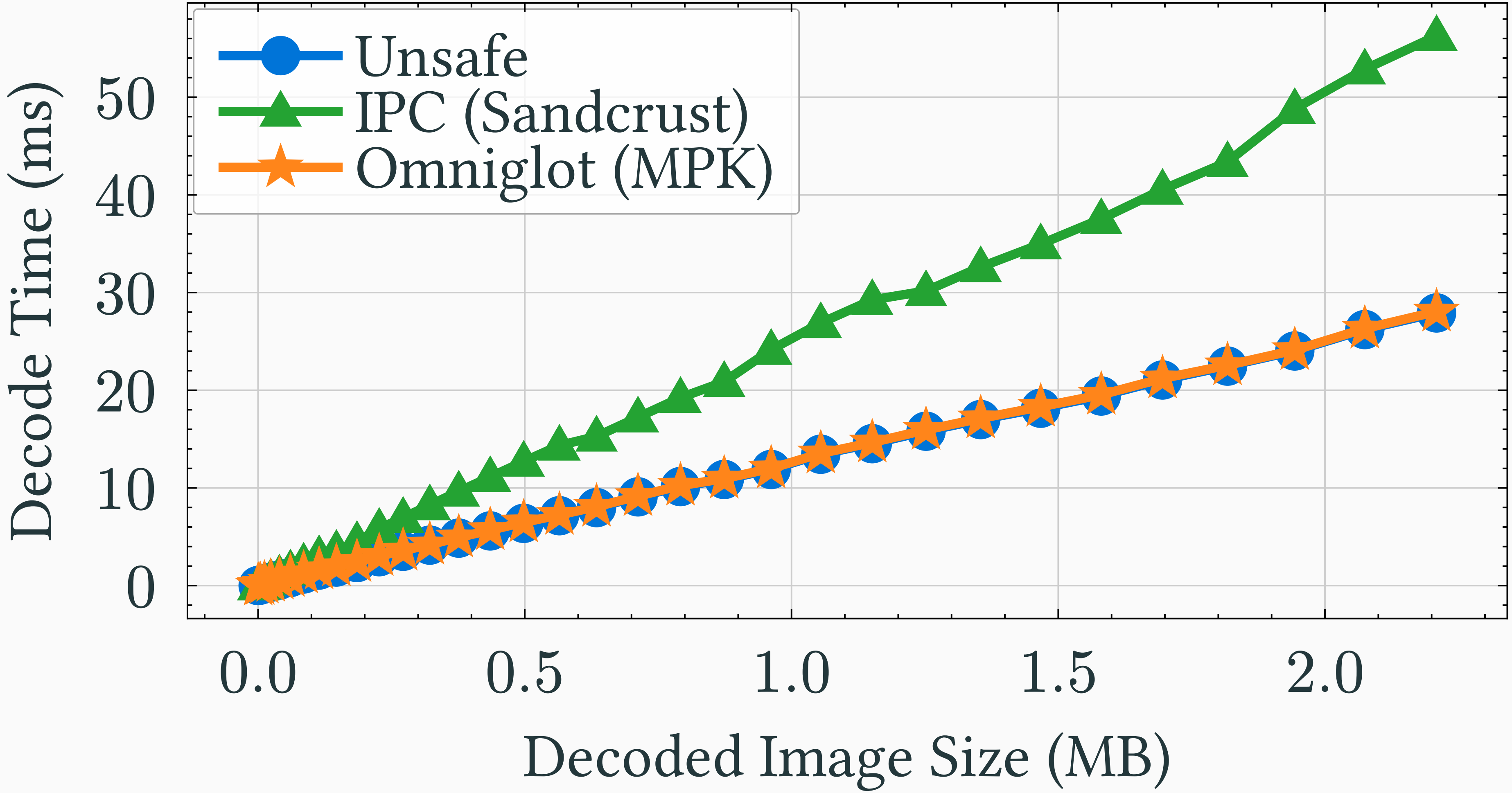
Omniglot is General



Omniglot is General




Unsafe
IPC (Sandcrust)
Decode Time (ms)
0
10
20
30
40
50
0.0
0.5
1.0
1.5
2.0
Decoded Image Size (MB)


Unsafe
IPC (Sandcrust)
Omniglot (MPK)
Decode Time (ms)
0
10
20
30
40
50
0.0
0.5
1.0
1.5
2.0
Decoded Image Size (MB)

Key Takeaways

1. We Need Safe Foreign Function Interfaces

Key Takeaways

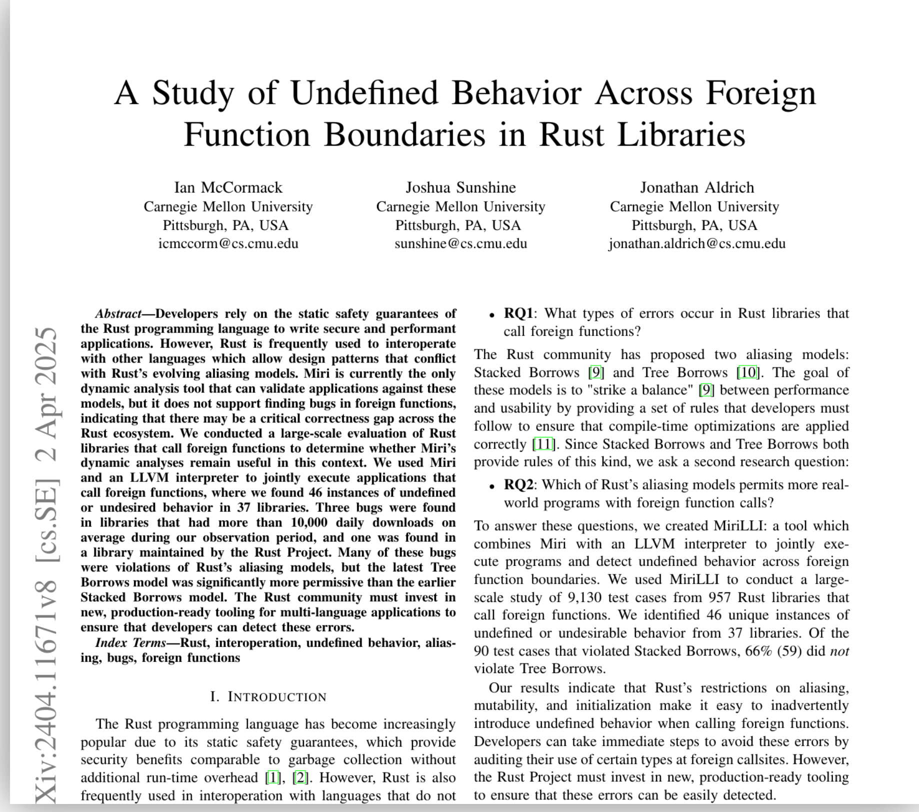
1. We Need Safe Foreign Function Interfaces
2. Memory Safety is Merely Table Stakes

Key Takeaways

1. We Need Safe Foreign Function Interfaces
2. Memory Safety is Merely Table Stakes
3. We Need Systematic Approaches to Maintain Rust's Invariants Across the FFI

Detecting Undefined Behavior across the FFI with testing and fuzzing

Ian McCormack, et. al, 2025



A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries

Ian McCormack
Carnegie Mellon University
Pittsburgh, PA, USA
icmccorm@cs.cmu.edu

Joshua Sunshine
Carnegie Mellon University
Pittsburgh, PA, USA
sunshine@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

***Abstract*—Developers rely on the static safety guarantees of the Rust programming language to write secure and performant applications. However, Rust is frequently used to interoperate with other languages which allow design patterns that conflict with Rust's evolving aliasing models. Miri is currently the only dynamic analysis tool that can validate applications against these models, but it does not support finding bugs in foreign functions, indicating that there may be a critical correctness gap across the Rust ecosystem. We conducted a large-scale evaluation of Rust libraries that call foreign functions to determine whether Miri's dynamic analyses remain useful in this context. We used Miri and an LLVM interpreter to jointly execute applications that call foreign functions, where we found 46 instances of undefined or undesired behavior in 37 libraries. Three bugs were found in libraries that had more than 10,000 daily downloads on average during our observation period, and one was found in a library maintained by the Rust Project. Many of these bugs were violations of Rust's aliasing models, but the latest Tree Borrows model was significantly more permissive than the earlier Stacked Borrows model. The Rust community must invest in new, production-ready tooling for multi-language applications to ensure that developers can detect these errors.**

***Index Terms*—Rust, interoperation, undefined behavior, aliasing, bugs, foreign functions**

I. Introduction

The Rust programming language has become increasingly popular due to its static safety guarantees, which provide security benefits comparable to garbage collection without additional run-time overhead [1], [2]. However, Rust is also frequently used in interoperation with languages that do not

- **RQ1**: What types of errors occur in Rust libraries that call foreign functions?

The Rust community has proposed two aliasing models: Stacked Borrows [9] and Tree Borrows [10]. The goal of these models is to "strike a balance" [9] between performance and usability by providing a set of rules that developers must follow to ensure that compile-time optimizations are applied correctly [11]. Since Stacked Borrows and Tree Borrows both provide rules of this kind, we ask a second research question:

- **RQ2**: Which of Rust's aliasing models permits more real-world programs with foreign function calls?

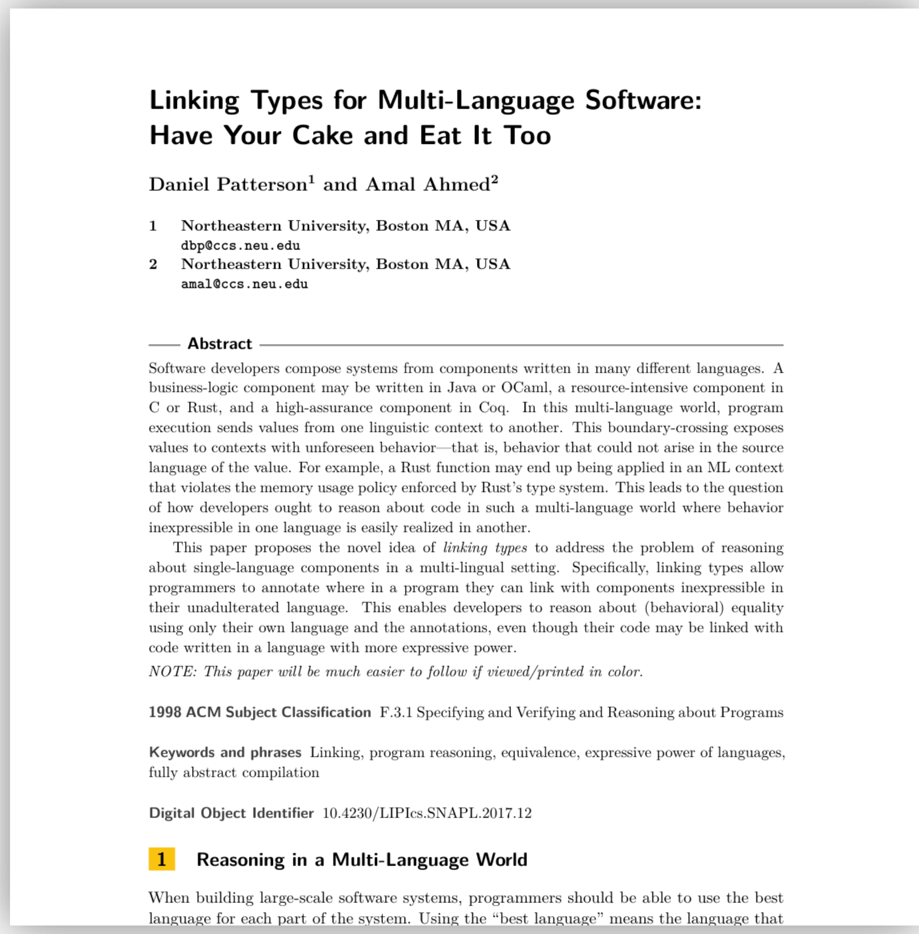
To answer these questions, we created MiriLLI: a tool which combines Miri with an LLVM interpreter to jointly execute programs and detect undefined behavior across foreign function boundaries. We used MiriLLI to conduct a large-scale study of 9,130 test cases from 957 Rust libraries that call foreign functions. We identified 46 unique instances of undefined or undesirable behavior from 37 libraries. Of the 90 test cases that violated Stacked Borrows, 66% (59) did *not* violate Tree Borrows.

Our results indicate that Rust's restrictions on aliasing, mutability, and initialization make it easy to inadvertently introduce undefined behavior when calling foreign functions. Developers can take immediate steps to avoid these errors by auditing their use of certain types at foreign callsites. However, the Rust Project must invest in new, production-ready tooling to ensure that these errors can be easily detected.

We Need Systematic Approaches to Maintain Soundness Across the FFI

Formally modeling a foreign languages' types in another language

David Patterson, et. al, 2017



Linking Types for Multi-Language Software: Have Your Cake and Eat It Too

Daniel Patterson[1] and Amal Ahmed[2]

1 Northeastern University, Boston MA, USA
dbp@ccs.neu.edu
2 Northeastern University, Boston MA, USA
amal@ccs.neu.edu

Abstract

Software developers compose systems from components written in many different languages. A business-logic component may be written in Java or OCaml, a resource-intensive component in C or Rust, and a high-assurance component in Coq. In this multi-language world, program execution sends values from one linguistic context to another. This boundary-crossing exposes values to contexts with unforeseen behavior—that is, behavior that could not arise in the source language of the value. For example, a Rust function may end up being applied in an ML context that violates the memory usage policy enforced by Rust's type system. This leads to the question of how developers ought to reason about code in such a multi-language world where behavior inexpressible in one language is easily realized in another.

This paper proposes the novel idea of *linking types* to address the problem of reasoning about single-language components in a multi-lingual setting. Specifically, linking types allow programmers to annotate where in a program they can link with components inexpressible in their unadulterated language. This enables developers to reason about (behavioral) equality using only their own language and the annotations, even though their code may be linked with code written in a language with more expressive power.

NOTE: This paper will be much easier to follow if viewed/printed in color.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Linking, program reasoning, equivalence, expressive power of languages, fully abstract compilation

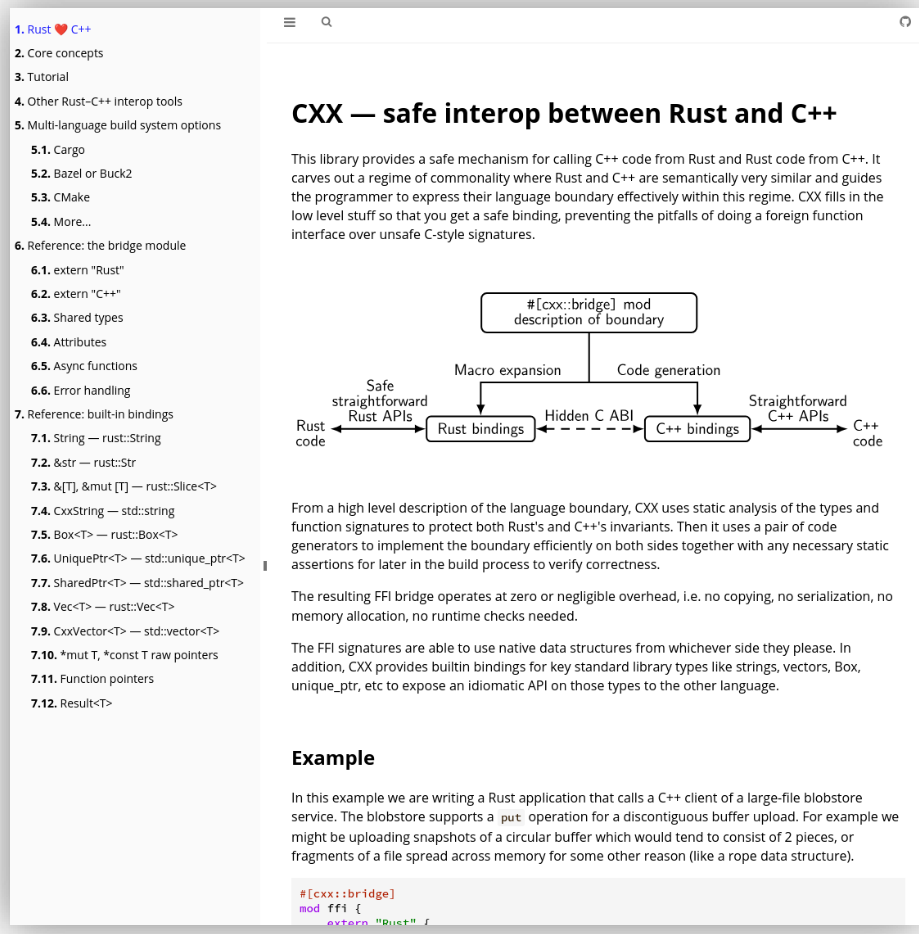
Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.12

1 Reasoning in a Multi-Language World

When building large-scale software systems, programmers should be able to use the best language for each part of the system. Using the "best language" means the language that

Bridging high-level language constructs for easier and more expressive bindings

David Tolnay's `cxx` crate, `wasm-bindgen`, …



CXX — safe interop between Rust and C++

This library provides a safe mechanism for calling C++ code from Rust and Rust code from C++. It carves out a regime of commonality where Rust and C++ are semantically very similar and guides the programmer to express their language boundary effectively within this regime. CXX fills in the low level stuff so that you get a safe binding, preventing the pitfalls of doing a foreign function interface over unsafe C-style signatures.

From a high level description of the language boundary, CXX uses static analysis of the types and function signatures to protect both Rust's and C++'s invariants. Then it uses a pair of code generators to implement the boundary efficiently on both sides together with any necessary static assertions for later in the build process to verify correctness.

The resulting FFI bridge operates at zero or negligible overhead, i.e. no copying, no serialization, no memory allocation, no runtime checks needed.

The FFI signatures are able to use native data structures from whichever side they please. In addition, CXX provides builtin bindings for key standard library types like strings, vectors, Box, unique_ptr, etc to expose an idiomatic API on those types to the other language.

Example

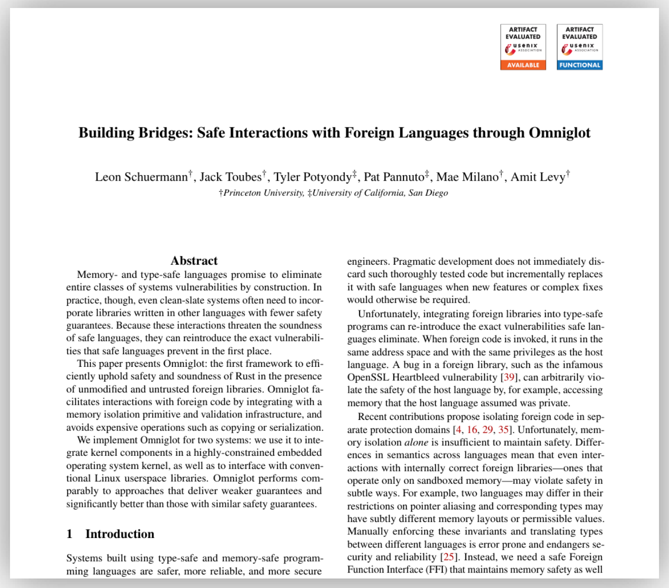
In this example we are writing a Rust application that calls a C++ client of a large-file blobstore service. The blobstore supports a `put` operation for a discontiguous buffer upload. For example we might be uploading snapshots of a circular buffer which would tend to consist of 2 pieces, or fragments of a file spread across memory for some other reason (like a rope data structure).

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
```

Omniglot mechanically maintains soundness, *without* assumptions about the foreign library's behavior

- Interposes on Rust's native foreign function interface,
- Integrates with memory isolation primitives, and
- Mediates all interactions with foreign code.

✅ Memory Safety ✅ Type Safety ✅ Efficiency


ARTIFACT EVALUATED usenix AVAILABLE
ARTIFACT EVALUATED usenix FUNCTIONAL

Building Bridges: Safe Interactions with Foreign Languages through Omniglot

Leon Schuermann[†], Jack Toubes[†], Tyler Potyondy[‡], Pat Pannuto[‡], Mae Milano[†], Amit Levy[†]
†Princeton University, ‡University of California, San Diego

Abstract

Memory- and type-safe languages promise to eliminate entire classes of systems vulnerabilities by construction. In practice, though, even clean-slate systems often need to incorporate libraries written in other languages with fewer safety guarantees. Because these interactions threaten the soundness of safe languages, they can reintroduce the exact vulnerabilities that safe languages prevent in the first place.

This paper presents Omniglot: the first framework to efficiently uphold safety and soundness of Rust in the presence of unmodified and untrusted foreign libraries. Omniglot facilitates interactions with foreign code by integrating with a memory isolation primitive and validation infrastructure, and avoids expensive operations such as copying or serialization.

We implement Omniglot for two systems: we use it to integrate kernel components in a highly-constrained embedded operating system kernel, as well as to interface with conventional Linux userspace libraries. Omniglot performs comparably to approaches that deliver weaker guarantees and significantly better than those with similar safety guarantees.

1 Introduction

Systems built using type-safe and memory-safe programming languages are safer, more reliable, and more secure engineers. Pragmatic development does not immediately discard such thoroughly tested code but incrementally replaces it with safe languages when new features or complex fixes would otherwise be required.

Unfortunately, integrating foreign libraries into type-safe programs can re-introduce the exact vulnerabilities safe languages eliminate. When foreign code is invoked, it runs in the same address space and with the same privileges as the host language. A bug in a foreign library, such as the infamous OpenSSL Heartbleed vulnerability [39], can arbitrarily violate the safety of the host language by, for example, accessing memory that the host language assumed was private.

Recent contributions propose isolating foreign code in separate protection domains [4, 16, 29, 35]. Unfortunately, memory isolation *alone* is insufficient to maintain safety. Differences in semantics across languages mean that even interactions with internally correct foreign libraries—ones that operate only on sandboxed memory—may violate safety in subtle ways. For example, two languages may differ in their restrictions on pointer aliasing and corresponding types may have subtly different memory layouts or permissible values. Manually enforcing these invariants and translating types between different languages is error prone and endangers security and reliability [25]. Instead, we need a safe Foreign Function Interface (FFI) that maintains memory safety as well

We Need Systematic Approaches to Maintain Soundness Across the FFI

Omniglot mechanically maintains soundness, *without* assumptions about the foreign library's behavior

- Interposes on Rust's native foreign function interface,
- Integrates with memory isolation primitives, and
- Mediates all interactions with foreign code.

✅ Memory Safety ✅ Type Safety ✅ Efficiency

Each solution has unique tradeoffs; there is no one size fits all.

Let's work together to bring guaranteed safety to Rust, even though foreign libraries are here to stay.

We Need Systematic Approaches to Maintain Soundness Across the FFI

Omniglot mechanically maintains soundness, *without* assumptions about the foreign library's behavior

- Interposes on Rust's native foreign function interface,
- Integrates with memory isolation primitives, and
- Mediates all interactions with foreign code.

✅ Memory Safety ✅ Type Safety ✅ Efficiency

Paper: Source code:

https://github.com/omniglot-rs/omniglot

Artifact reproduction instructions:

doi 10.5281/zenodo.15602886