BorrowSanitizer

Efficiently Finding Aliasing Bugs in Multilanguage Rust Applications


Ian McCormack
Carnegie Mellon University


Oliver Braunsdorf
Ludwig Maximilian University


Johannes Kinder
Ludwig Maximilian University


Jonathan Aldrich
Carnegie Mellon University

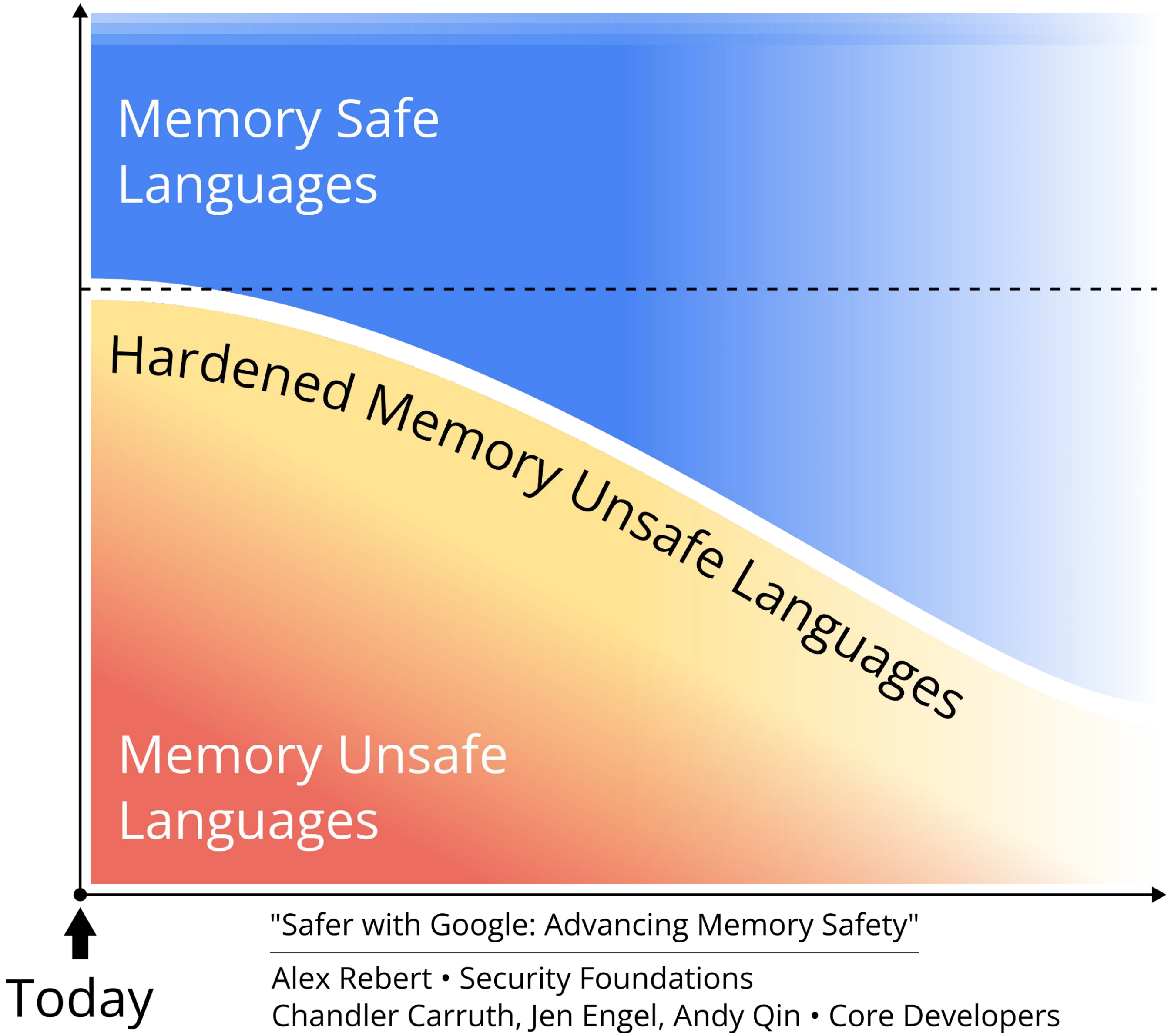

Joshua Sunshine
Carnegie Mellon University

Project Site

Background & Motivation

Design Principles

Future Work



"Safer with Google: Advancing Memory Safety"

Alex Rebert • Security Foundations
Chandler Carruth, Jen Engel, Andy Qin • Core Developers

Rust developers use a set of ”unsafe” features to interoperate with other languages.

Calling `unsafe` functions

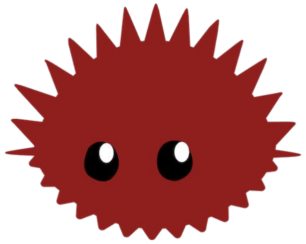
Intrinsics & inline assembly

Manipulating uninitialized memory

Accessing global, mutable state

Dereferencing raw pointers

Implementing an `unsafe` trait



Developers can use unsafe code to break Rust's aliasing rules.

Safe References

`&T`
Shared, Read-only

`&mut T`
Unique, Write

Raw Pointers

`*const/mut T`
Shared, Write

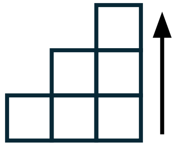
```
let mut x: i8 = 0;
let rx = &mut x;
let ptr = rx as *mut _;

example(rx, unsafe { &mut *ptr });

fn example(x: &mut i8, y: &mut i8) {
  *x = 0
  *y = 1;
  *x;
}
```

Credit: Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer

Miri, a Rust interpreter, can find these aliasing bugs

Stacked Borrows

Ralf Jung, Hong-Hai Dang,
Jeehon Kang, and Derek Dreyer

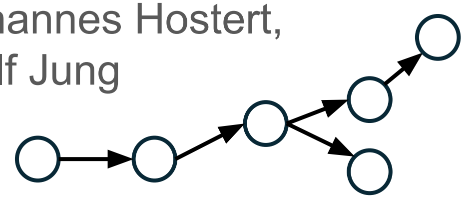


POPL '20

OR

Tree Borrows

Neven Villani, Johannes Hostert,
Derek Dreyer, Ralf Jung



Preprint

Bounds Checking

Liveness Checking

Data Race Detection

Pointer

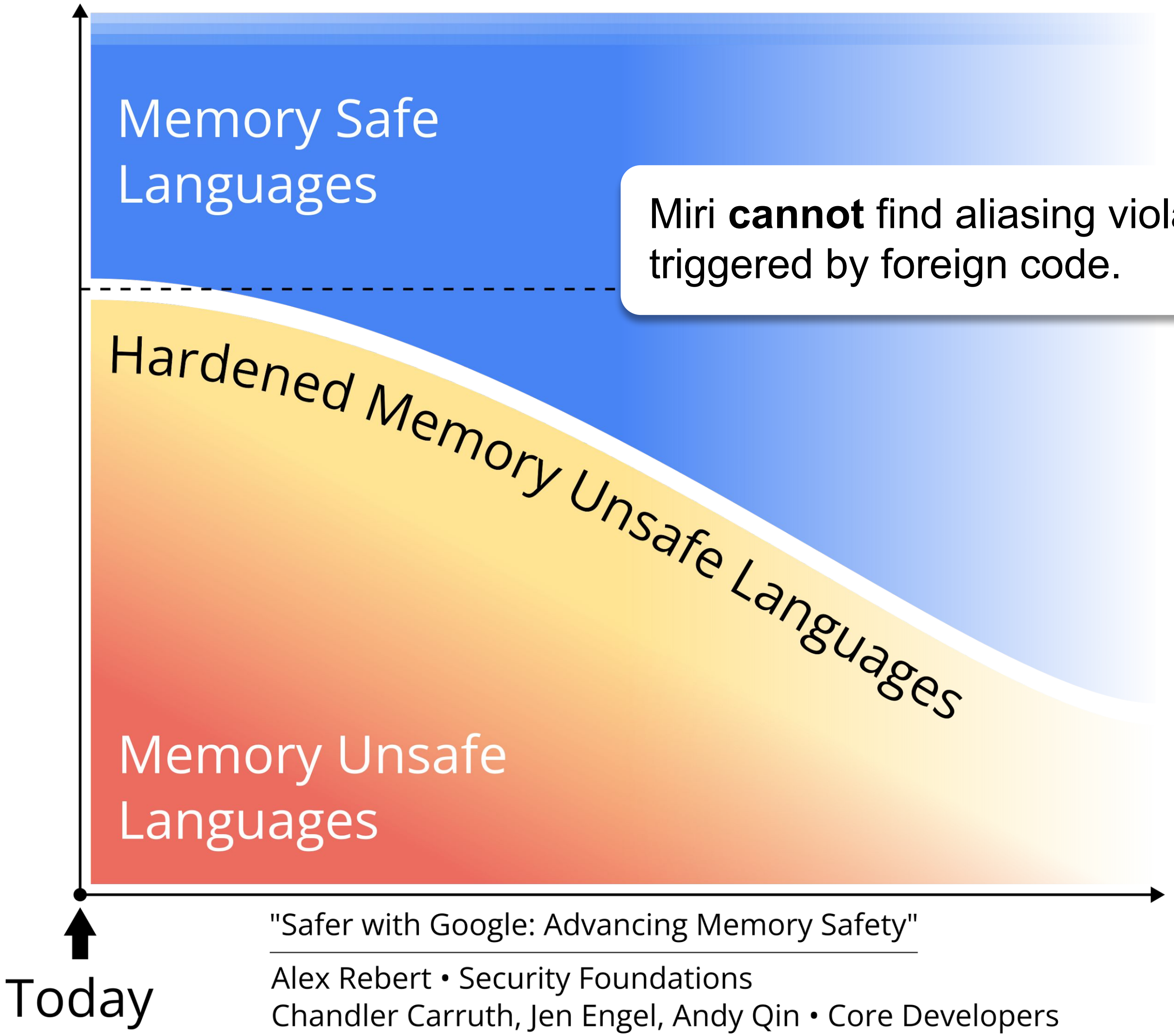
(Address, Provenance)

$\mathbb{N}$

Provenance

(Allocation ID, Tag)

$\mathbb{N}$ $\mathbb{N}$



Miri **cannot** find aliasing violations triggered by foreign code.

"Safer with Google: Advancing Memory Safety"

Alex Rebert • Security Foundations
Chandler Carruth, Jen Engel, Andy Qin • Core Developers

Are aliasing violations hiding, undetected, in multilanguage Rust programs?

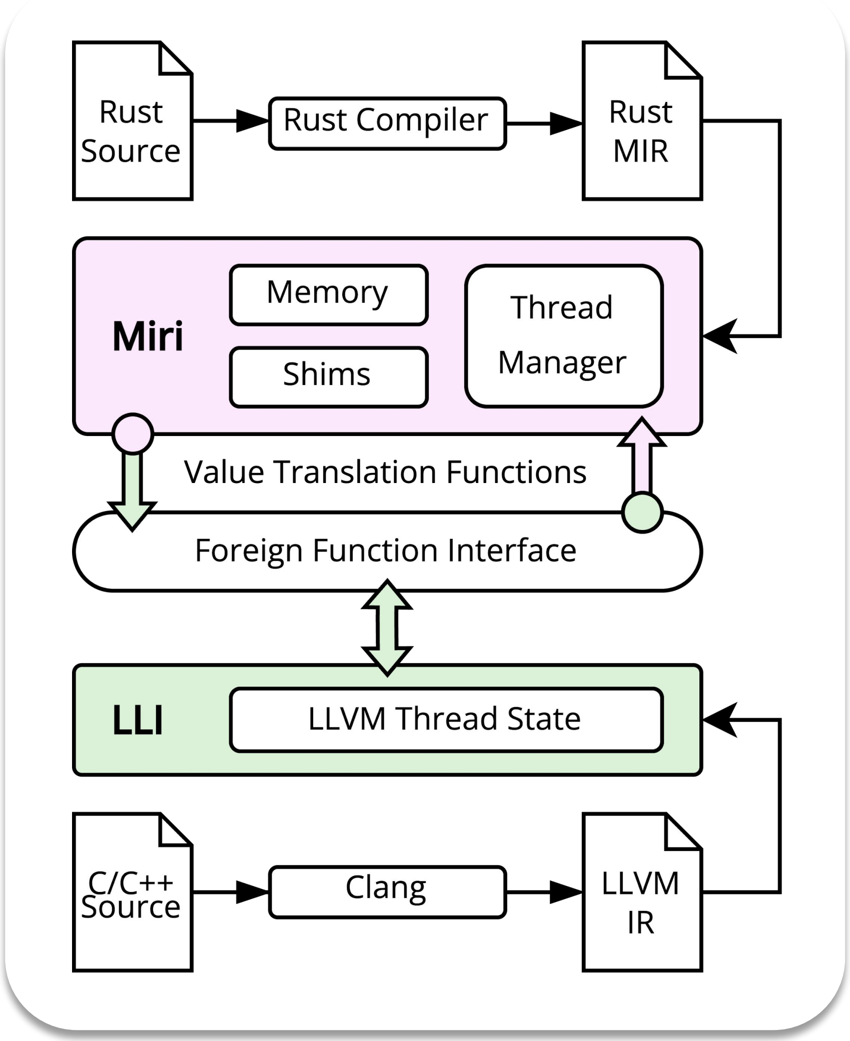
A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries

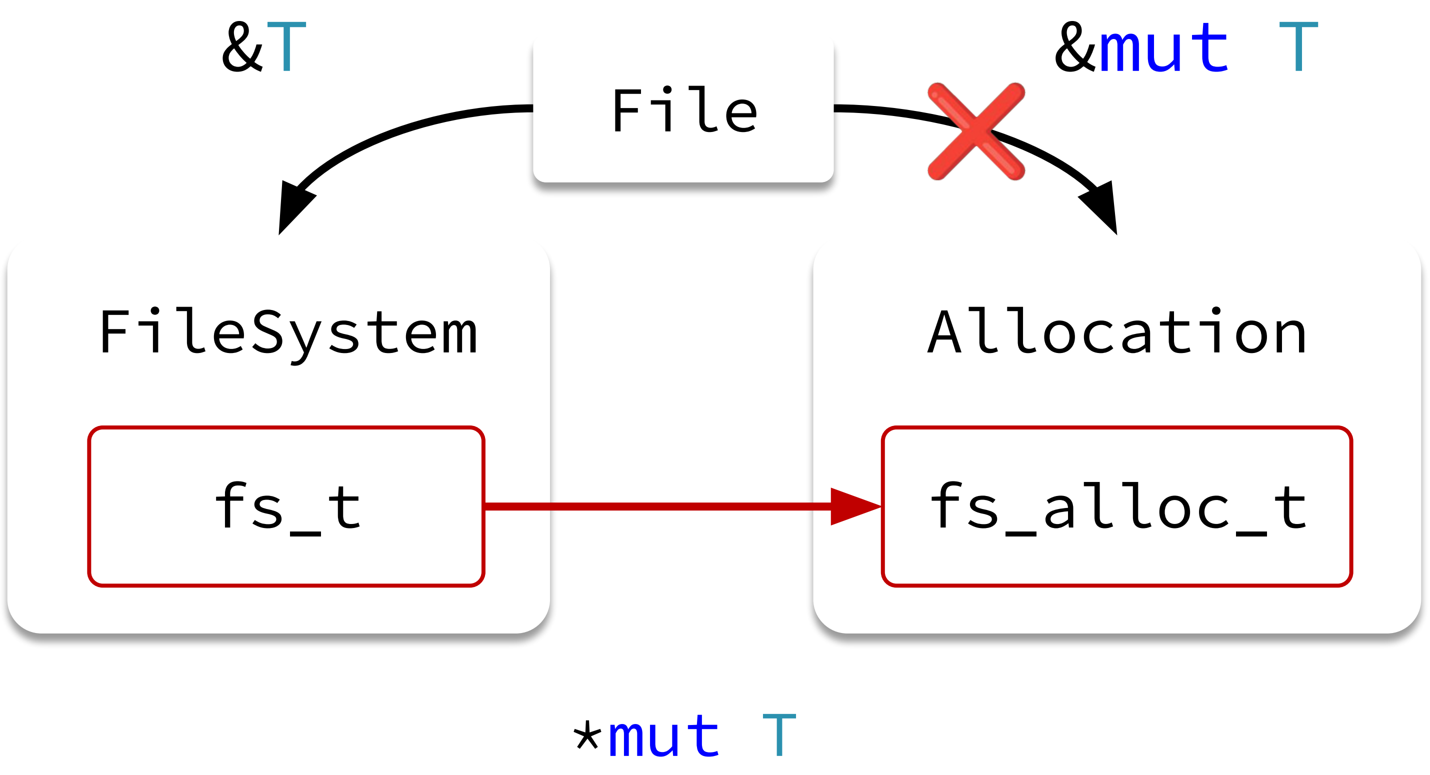
Ian McCormack, Jonathan Aldrich, Joshua Sunshine

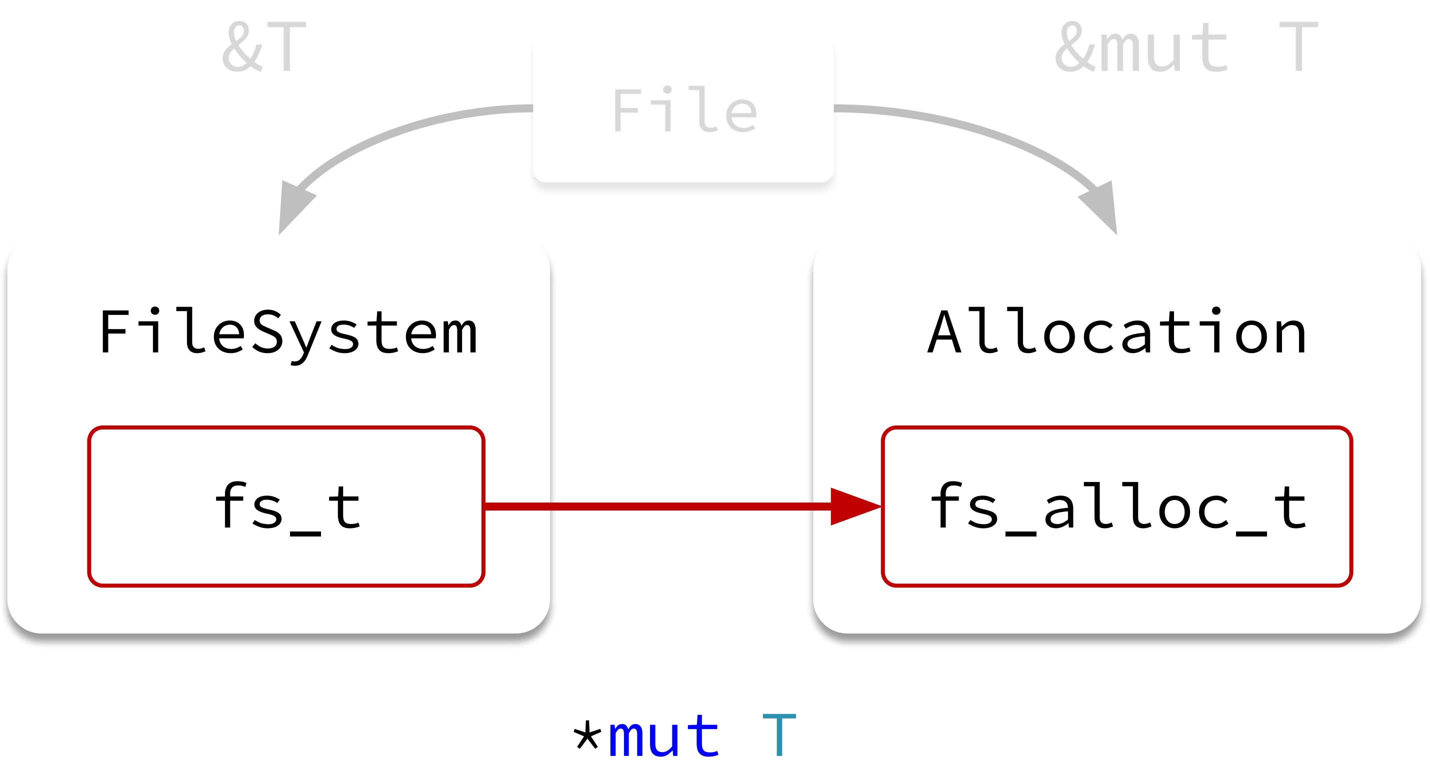
ICSE 2025

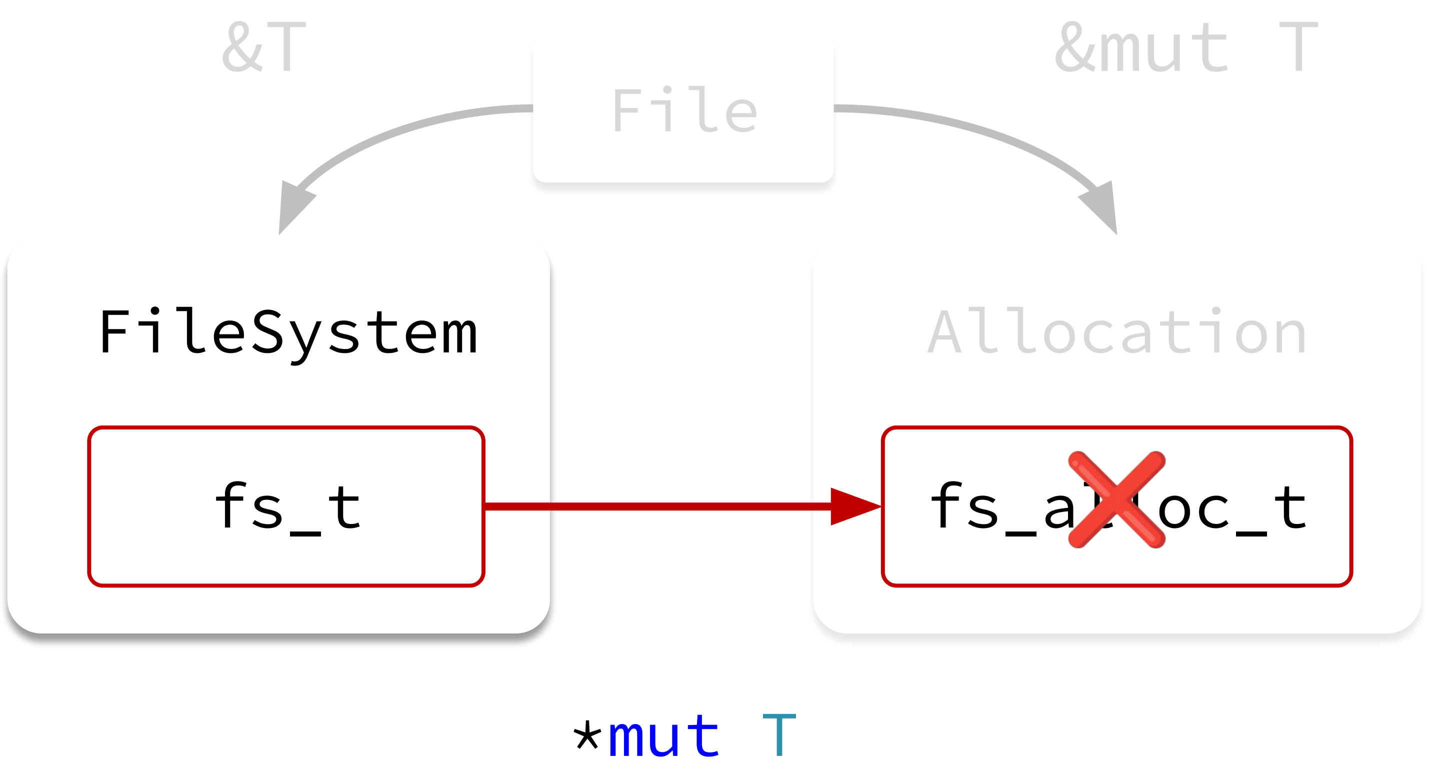
We combined Miri with LLI, an LLVM interpreter, to create **MiriLLI**.

Our tool uses each interpreter to jointly execute programs defined across Rust and **LLVM IR**.




&T
File
&mut T
FileSystem
fs_t
Allocation
fs_alloc_t
*mut T


&T
File
&mut T
FileSystem
fs_t
Allocation
fs_alloc_t
*mut T


&T
File
&mut T
FileSystem
Allocation
fs_t
fs_alloc_t
*mut T

Rust for Linux

```
fn get_or_create_inode(&self, ino:Ino) ->
    Result<Either<ARef<INode<T>>, inode::New<T>>>
```

Filesystem in Rust • Kent Overstreet, 2024

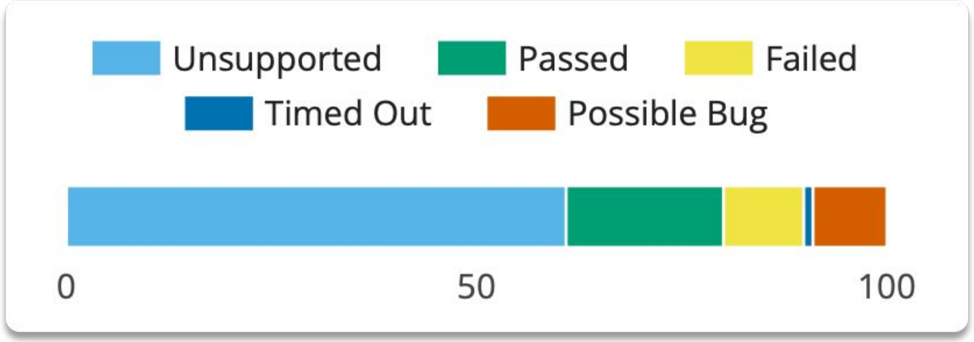
Miri is not enough for large-scale, multilanguage applications.

Compatibility

We evaluated MiriLLI on every compatible crate.

There were **9,130** compatible tests from 957 crates.

61% encountered an unsupported operation.



Performance

Miri is several orders of magnitude slower than native execution

What should a new tool look like?

Usable

`cargo <tool> run`

Fast

Native instrumentation…

C/C++ Support

…through a common format.

Pointer-Level Metadata

Pointer

(Address, Provenance)

$\mathbb{N}$

→

Provenance

(Allocation ID, Tag)

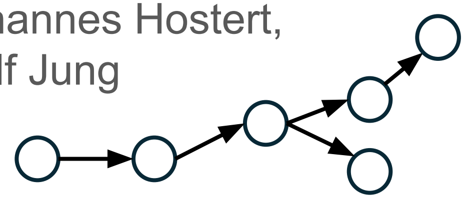
$\mathbb{N}$ $\mathbb{N}$

Allocation-Level Metadata

Tree Borrows

Neven Villani, Johannes Hostert,
Derek Dreyer, Ralf Jung

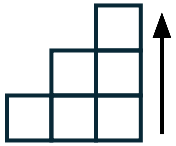
Preprint



Stacked Borrows

Ralf Jung, Hong-Hai Dang,
Jeehon Kang, and Derek Dreyer

POPL ‘20



“Identity-Based Access Checking”

SoK: Sanitizing for Security • Song et al., 2019

Valgrind injects instrumentation into compiled programs.

Usable ✅

Fast ✅

C/C++ Support ✅

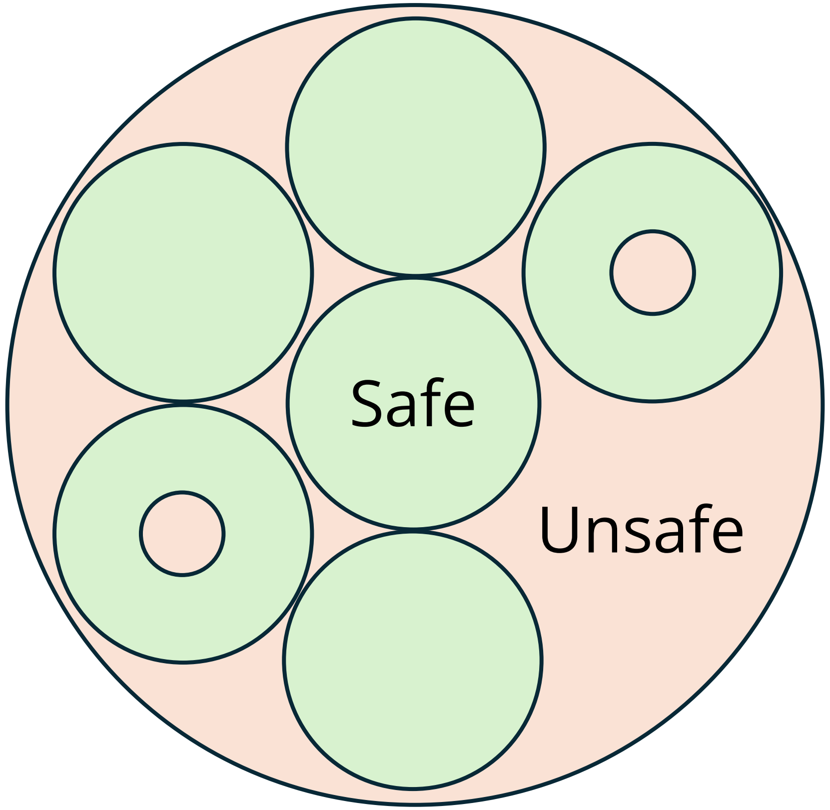
In 2023, the **Krabcake** project proposed extending Valgrind to support detecting Stacked Borrows violations.

Felix Klock, Bryan Garza • AWS

RW2023!

Valgrind's baseline overhead is still **4x.**

Components written in safe Rust *can* be provably **free of undefined behavior**



BorrowSanitizer

Finding aliasing bugs at-scale

borrowsanitizer.com

Compile-time Instrumentation

Usable

Fast

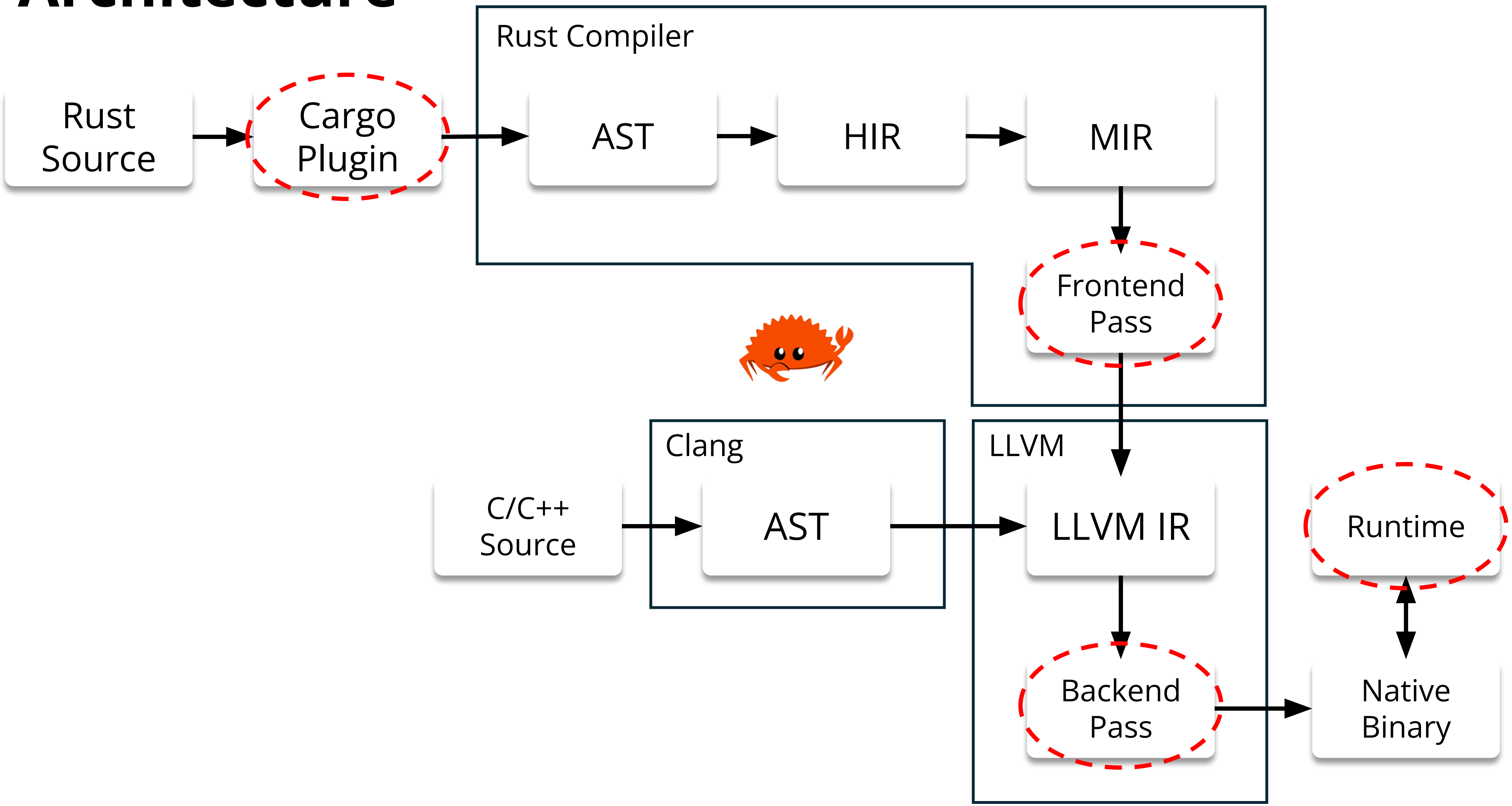
C/C++ Support

Background & Motivation

Design Principles

Future Work

Architecture



Frontend Pass

```rust
fn max<'a>(x: &'a mut i8, y: &'a mut i8) -> &'a mut i8
{
  retag(x);
  retag(y);
  if x > y {
    retag(x);
    x
  } else {
    retag(y);
    y
  }
}
```

```
@llvm.retag(ptr, usize, u8, u8, u8)
```

The base address

The access size

Whether to apply a protector

Whether the place is "unpin"

Whether the place is "frozen"

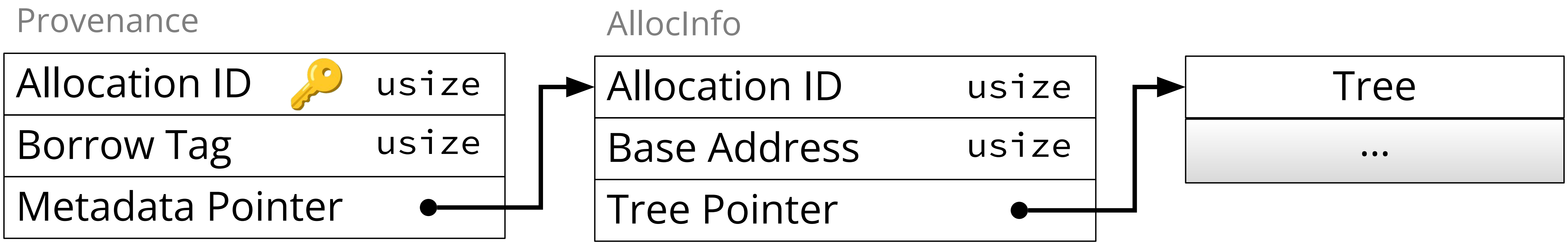
Backend Pass

```
call @llvm.retag(ptr %p1, 4, …)
; Allocate 4 bytes on the heap for i32
%p2 = call @__rust_alloc(4)
; Load value from %p1
%val_p1 = load i32, ptr %p1
; store val_p1 into heap object
store i32 %val_p1, ptr %p2
...
```

→

```
call @__bsan_retag(ptr %p1, 4, …)
%p2 = call @__rust_alloc(4)
%p2_prov = call i8* @__bsan_alloc(%p2, 4)
call @__bsan_load(%p1, %p1_prov, 4)
%val_p1 = load i32, ptr %p1
call @__bsan_store(%p2, %p2_prov, 4)
store i32 %val_p1, ptr %p2
...
```

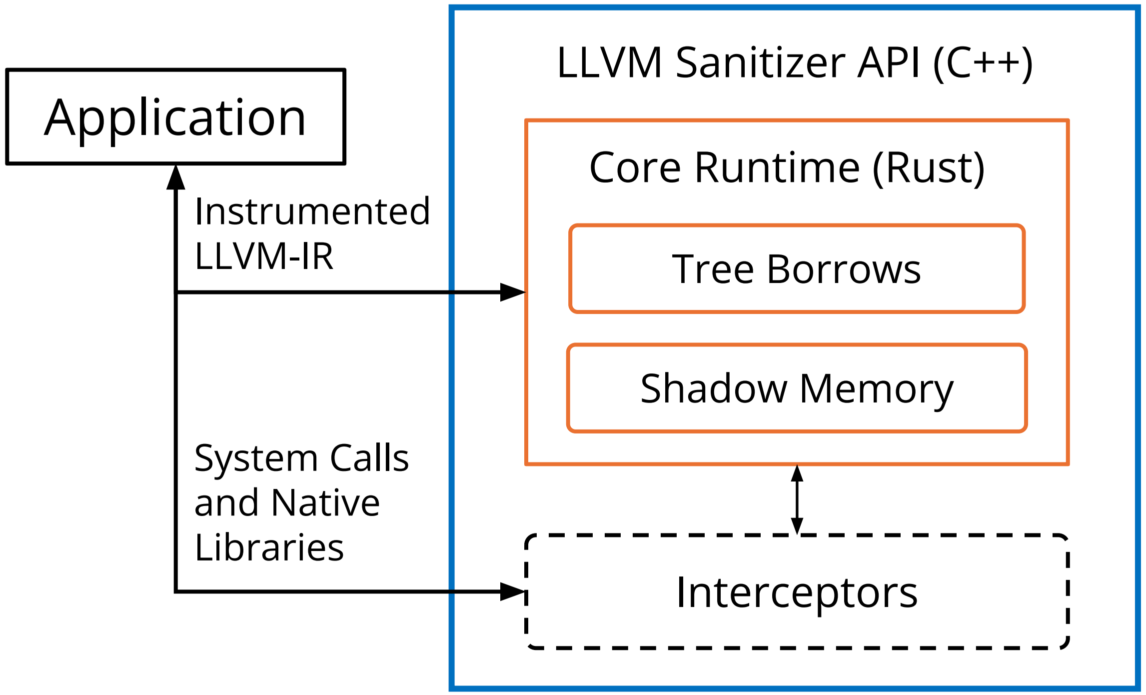
Pointer & Object Metadata



Runtime

Core functions are implemented as a reusable Rust component encapsulated by a C++ wrapper.

Requires `#[no_std]` to support intercepting common system calls



Design Choices - Overview

Object-Metadata Management

Pointer-Metadata Management

Support for Multithreading

Handling 3rd Party Libraries

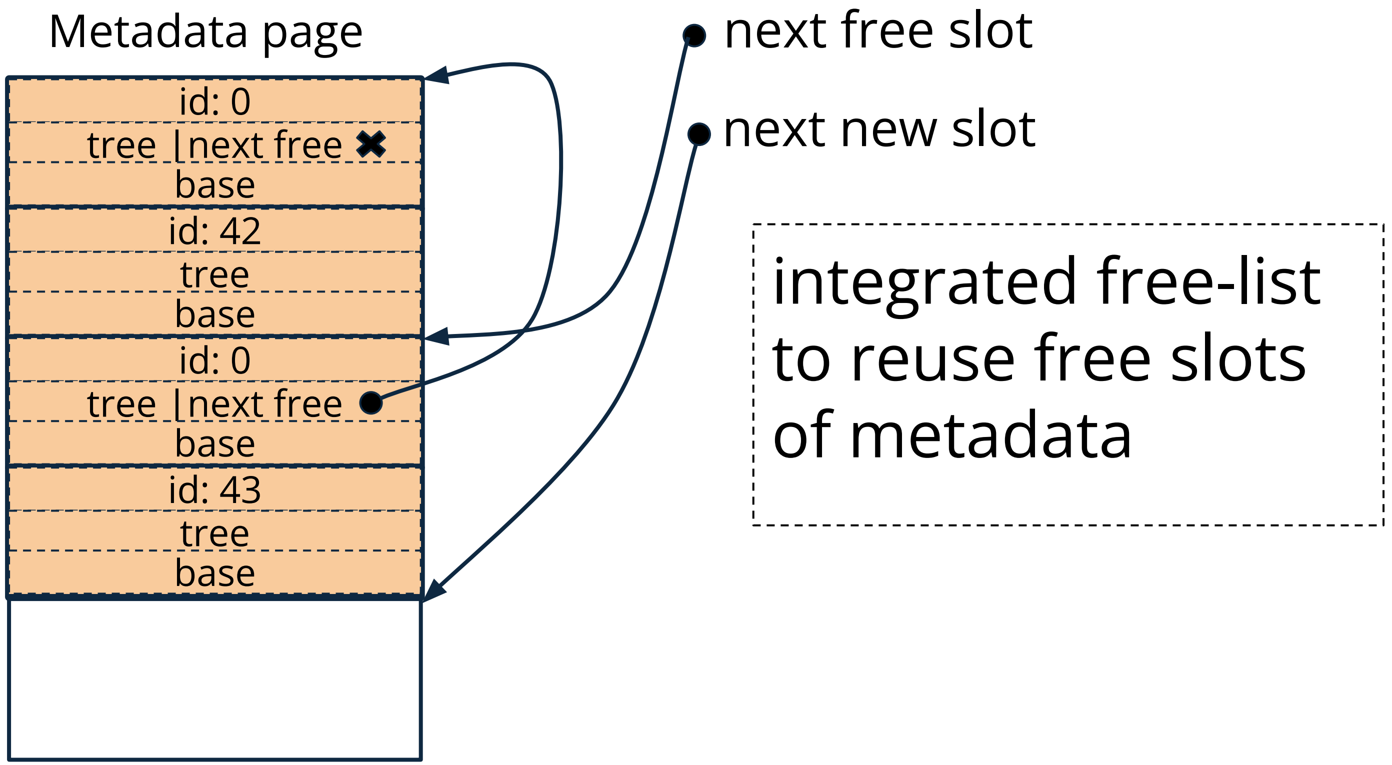
Object Metadata

Bump Allocator:

Allocate Metadata for each heap/stack object on special zero-initialized pages.

+ No wasted space for deallocated memory

+ Only one memory read per metadata field.



Pointer Metadata Management

Disjoint Metadata

```
fn a() {
 a = 42
 ptr = alloc(..)
 b(ptr)
}
```

ptr	00007ffd4b2f8000
id	fa8b82858c060f08
borrow tag	0000000000000003
lock addr	00002deadbeef00
a	000000000000042

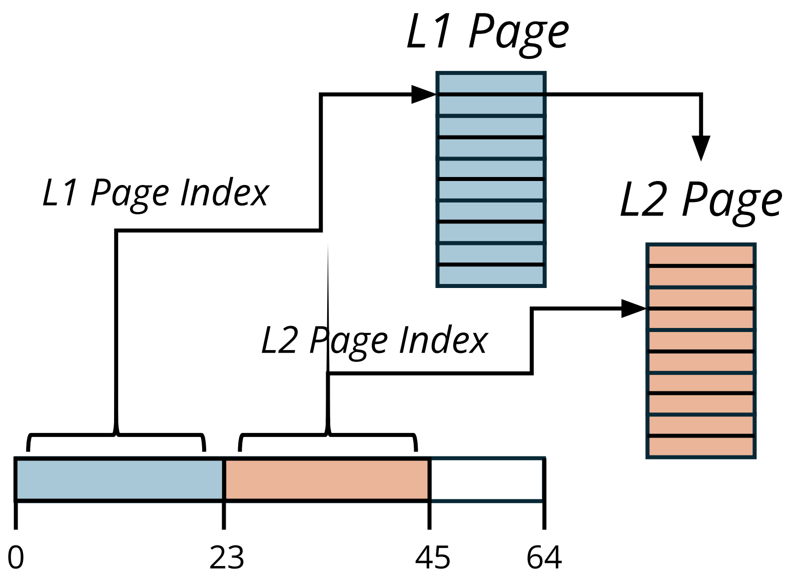
```
fn b(ptr) {

  ...

}
```

- \+ No modification of pointer layout
- \+ Can store more metadata
- – More effort to propagate

Pointer Metadata Management

When storing or loading pointers on the heap:

- Store/load pointer metadata using 2-level page-table
- The address itself is the index



CETS: Compiler-Enforced Temporal Safety for C
Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, Steve Zdancewic

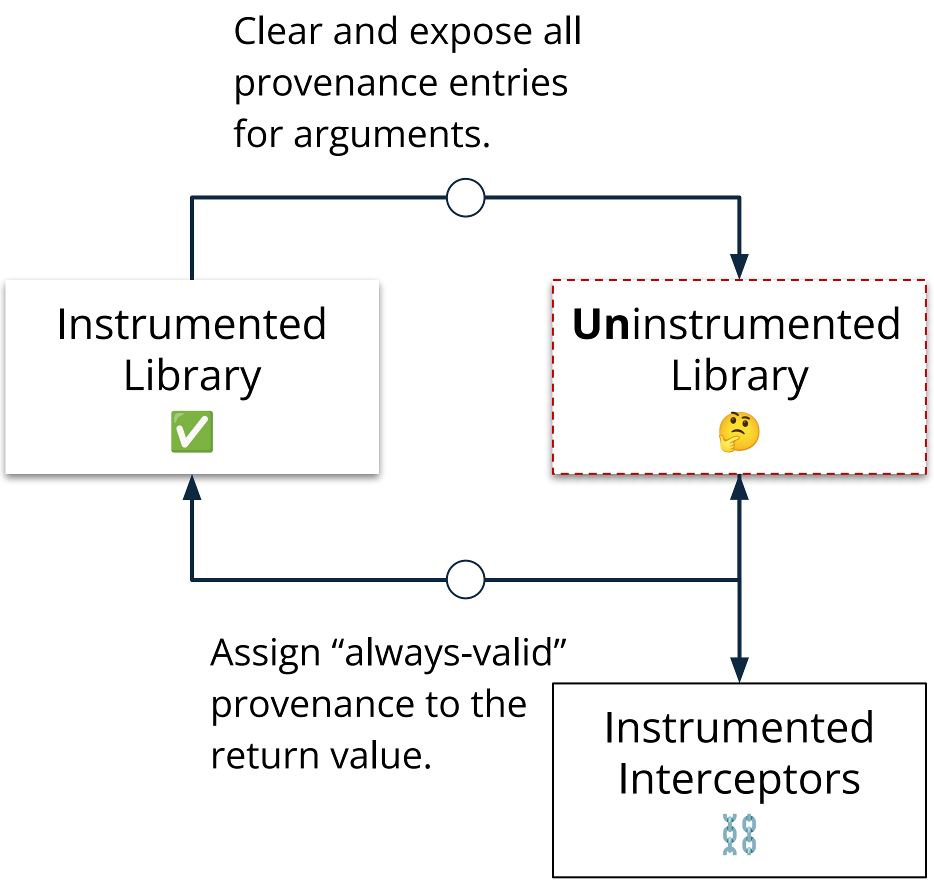
Handling Uninstrumented Libraries

Our default policy will match Miri's behavior for native library calls.

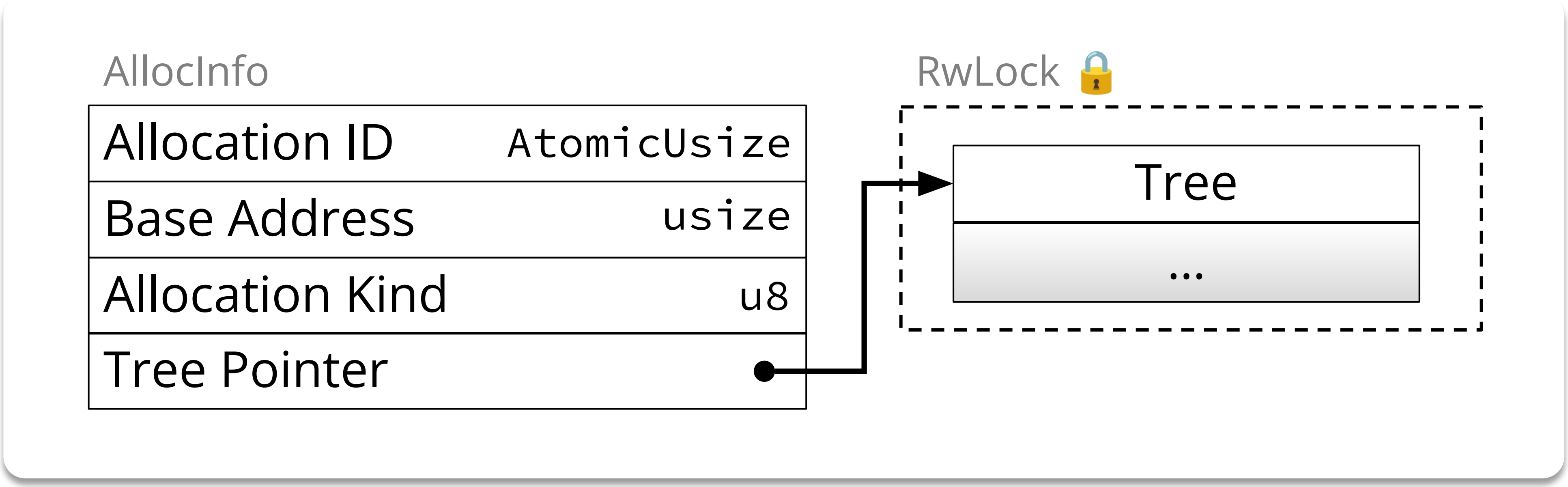
- ✏️ Expose all provenance entries for pointer arguments.
- ✳️ Overwrite shadow provenance entries in their underlying allocation with "wildcard" values.

Maintaining metadata integrity requires knowing whether the caller is instrumented.

Can we detect *some* violations in 3rd party code using interception?



Support for Multithreading

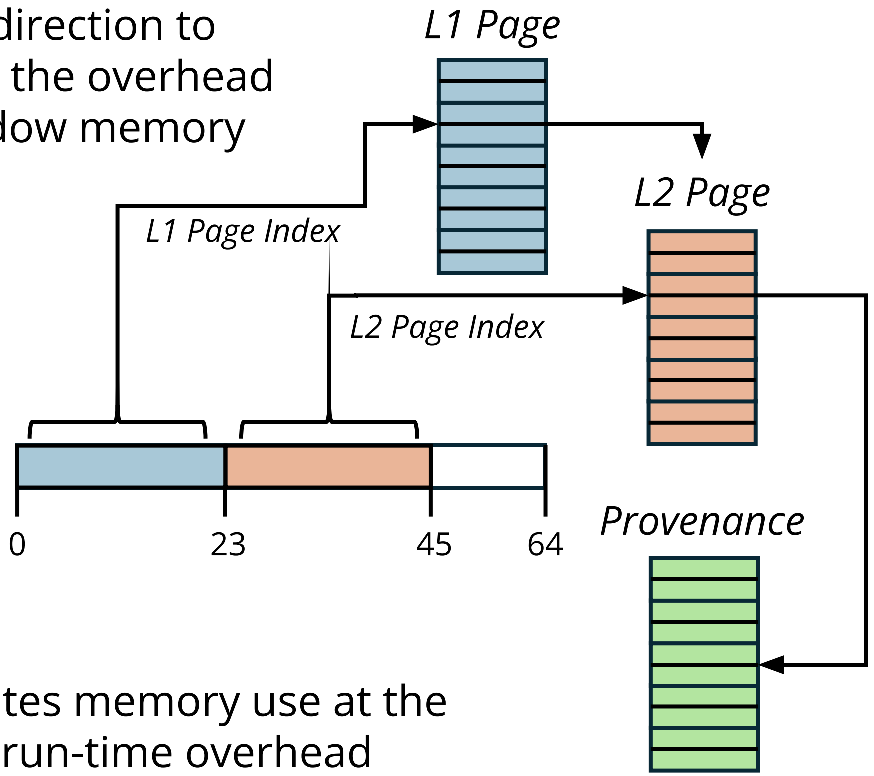


Big area of challenges, need lock-free data structures eventually

- Background & Motivation
- Design Principles
- Future Work

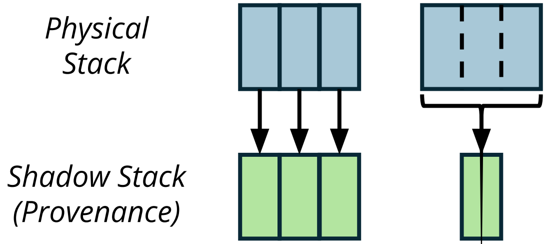
Dynamic Optimizations – Shadow Memory

Add indirection to reduce the overhead of shadow memory

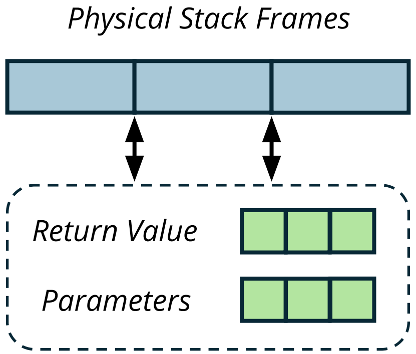


Eliminates memory use at the cost of run-time overhead

Track stack allocations as a single chunk of memory.



Thread-local storage for passing provenance.



Add indirection to shadow memory. Compress the Tree.

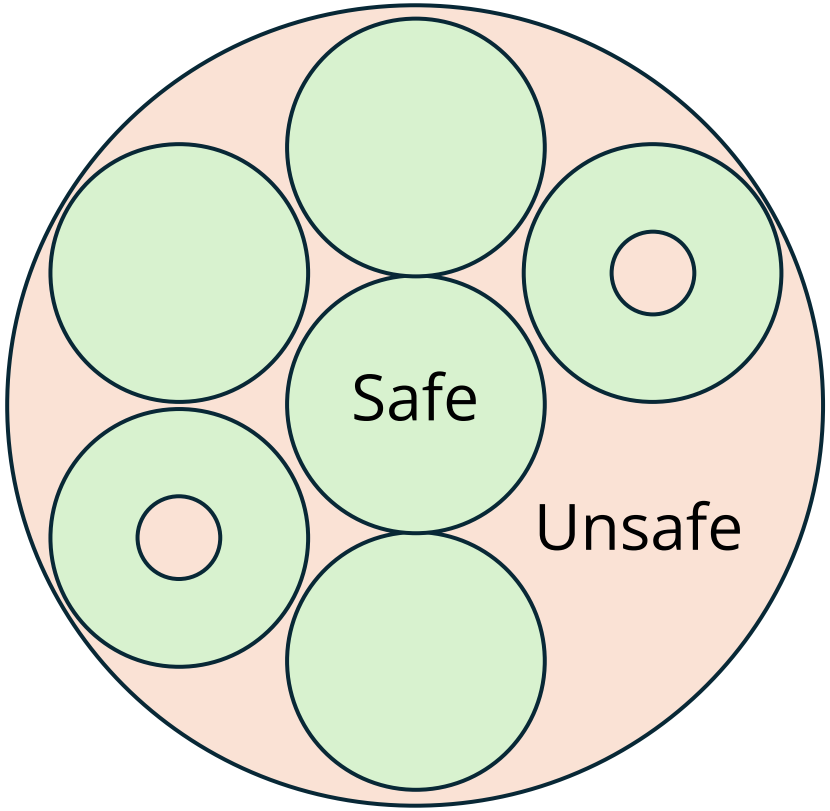
Defer initializing the Tree. LLVM temporaries. Inlining.

Tree Borrows is inherently expensive.

Reduce the size of the borrow tag. Only stack allocations.

Reduce the size of the Allocation ID. Tag-check for Frozen.

Components written in safe Rust *can* be provably **free of undefined behavior**



```rust
let mut x: i8 = 0;
let rx = &mut x;
let p = rx as *mut _;
let z = 1;
example(rx, unsafe { &mut *ptr }, &z);

fn example(rx: &mut i8, y: &mut i8, rz: &i8)
{
  *rx = 0;
  *y = 1;
  *rx;
}
```

Credit: Ralf Jung, Hoang-Hai Dang,
Jeehoon Kang, and Derek Dreyer

Borrow *Checking*

&T

Shared, Read-only

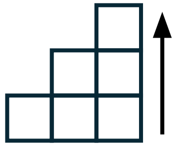
&mut T

Unique, Write

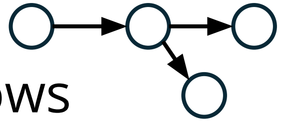
Static

Borrow *Tracking*

Stacked Borrows



Tree Borrows



Dynamic

Gradual Typing

*mut ? T

Phase 1
September 2025

Phase 2
December 2025

Phase 3
September 2026

Single-threaded
+
Static Proof Sketch

Multi-threaded
+
Dynamic Optimizations

Static Optimizations

BorrowSanitizer

Efficiently Finding Aliasing Bugs in Multilanguage Rust Applications

Project Site
borrowsanitizer.com


Ian McCormack
Carnegie Mellon University

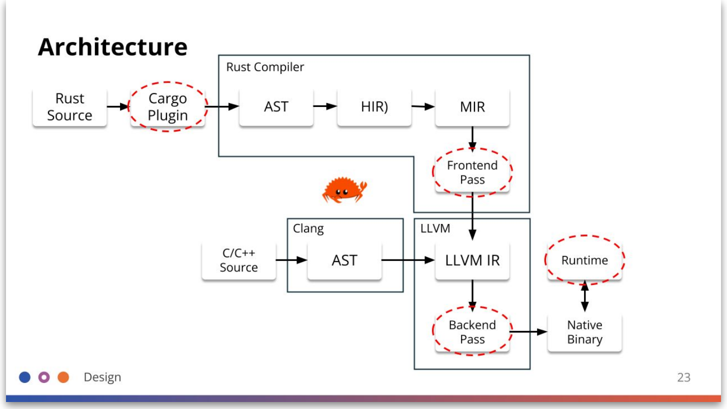

Oliver Braunsdorf
Ludwig Maximilian University

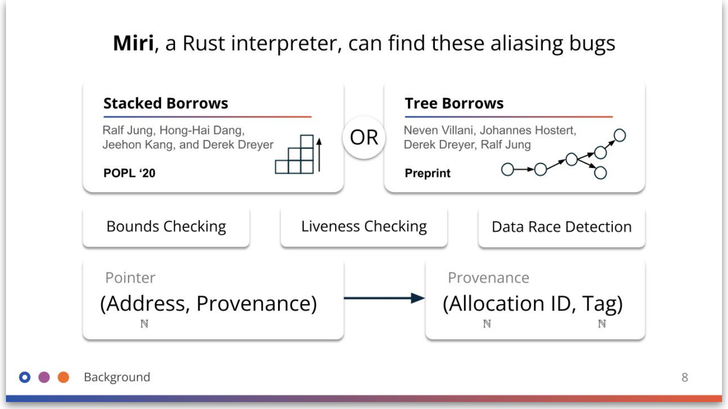

Johannes Kinder
Ludwig Maximilian University

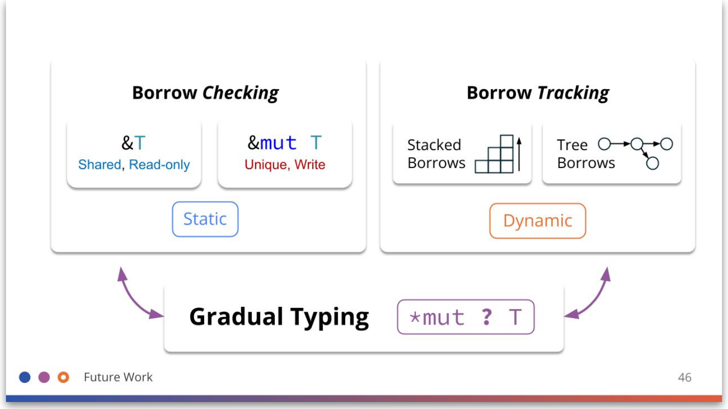

Jonathan Aldrich
Carnegie Mellon University


Joshua Sunshine
Carnegie Mellon University

bsan.zulipchat.com







This material is based on work supported by the Department of Defense and the National Science Foundation under Grant Nos. CCF-1901033, DGE1745016, and DGE2140739.