Mind the Boundary:
Detecting Undefined Behavior Across Rust's FFI

Andreea Costea,
TU Delft

IFIP W.G 2.4 Savannah March 2026

Mind the Boundary:
Detecting Undefined Behavior Across Rust's FFI

WIP

We need a safer systems progr language

Security Research & Defense / By MSRC Team / July 18, 20

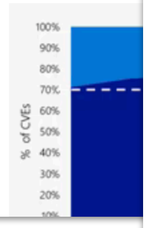
In our first post in this series, we discussed t
preventing this class of vulnerabilities; mem
over a decade. We feel that using memory-s

In this post, we'll explore some real-world e
prevented by using a memory-safe languag

Memory Safety

Memory safety is a property of programmin
are memory-safe because they use some fo
underlying systems other software depends
collector are usually not memory-safe.

As was pointed out in our previous post, the
(Common Vulnerabilities and Exposures) are
static analysis, and more.



Google Security Blog

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

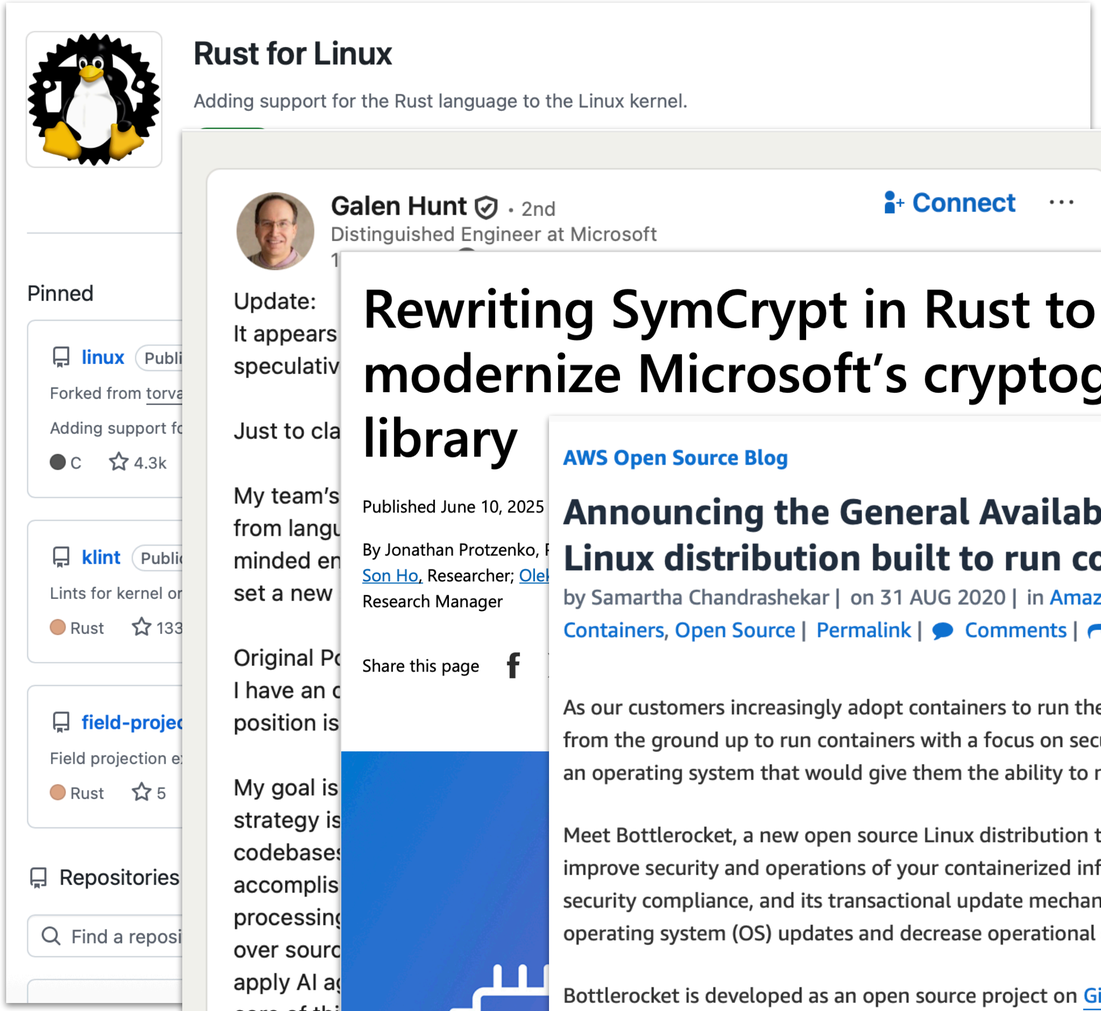
Safer with Google: Advancing Memory Safety

October 15, 2024

Posted by Alex Rebert, Security Foundations, and Chandler Carruth, Jen Engel, Andy Qin, Core Developers

Error-prone interactions between software and memory[1] are widely understood to create safety issues in software. It is estimated that about 70% of severe vulnerabilities[2] in memory-unsafe codebases are due to memory safety bugs. Malicious actors exploit these vulnerabilities and continue to create real-world harm. In 2023, Google's threat intelligence teams conducted an industry-wide study and observed a close to all-time high number of vulnerabilities exploited in the wild. Our internal analysis estimates that 75% of CVEs used in zero-day exploits are memory safety vulnerabilities




Rust for Linux
Adding support for the Rust language to the Linux kernel.
Pinned
linux
Forked from torva
Adding support fo
C
4.3k
klint
Lints for kernel or
Rust
133
field-projec
Field projection e
Rust
5
Repositories
Find a reposi

Galen Hunt
2nd
Distinguished Engineer at Microsoft
Connect
Update:
It appears
speculativ
Just to cla
My team's
from langu
minded en
set a new
Original Pc
I have an c
position is
My goal is
strategy is
codebases
accomplis
processing
over sourc
apply AI ag
core of thi

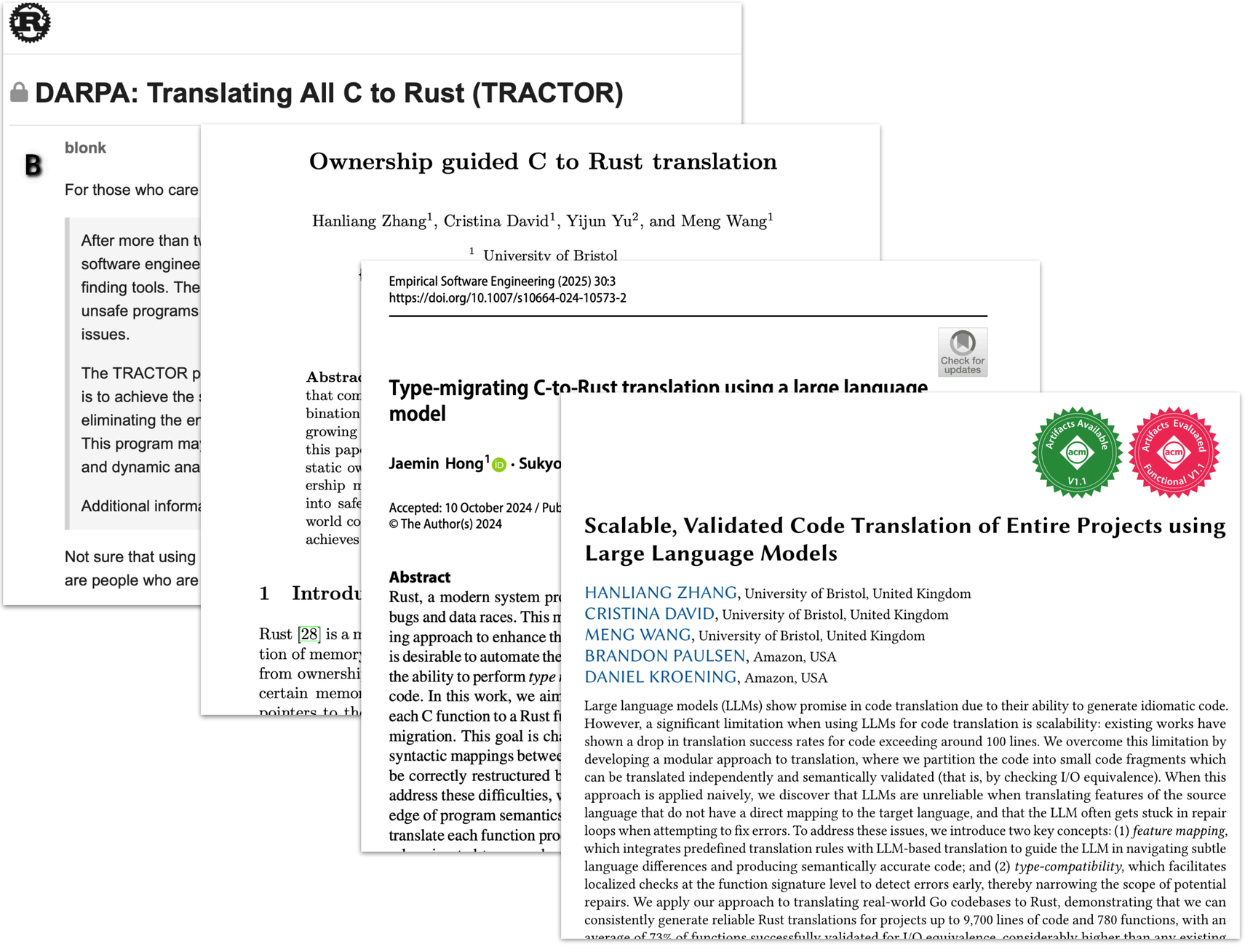
Rewriting SymCrypt in Rust to modernize Microsoft's cryptographic library
Published June 10, 2025
By Jonathan Protzenko,
Son Ho, Researcher; Olek
Research Manager
Share this page

AWS Open Source Blog
Announcing the General Availability of Bottlerocket, an open so
Linux distribution built to run containers
by Samartha Chandrashekar | on 31 AUG 2020 | in Amazon EC2, Announcements, AWS Partner Network, Comput
Containers, Open Source | Permalink | Comments | Share
As our customers increasingly adopt containers to run their workloads, we saw a need for a Linux distribution desi
from the ground up to run containers with a focus on security, operations, and manageability at scale. Customers
an operating system that would give them the ability to manage thousands of hosts running containers with auto
Meet Bottlerocket, a new open source Linux distribution that is built to run containers. Bottlerocket is designed to
improve security and operations of your containerized infrastructure. Its built-in security hardening helps simplify
security compliance, and its transactional update mechanism enables the use of container orchestrators to automa
operating system (OS) updates and decrease operational costs.
Bottlerocket is developed as an open source project on GitHub with a public roadmap. We're looking forward to bu
a community around Bottlerocket on GitHub and welcome your feature requests, bug reports, or contributions.
Bottlerocket technology
We began designing and building Bottlerocket based on the things we've learned from how customers use Amazo
to run containers and from running services such as AWS Fargate. At every step of the design process, we optimize
Bottlerocket for security, speed, and ease of maintenance.
Bottlerocket improves security by including only the software needed to run containers, which reduced the securit


ada's
SPARK

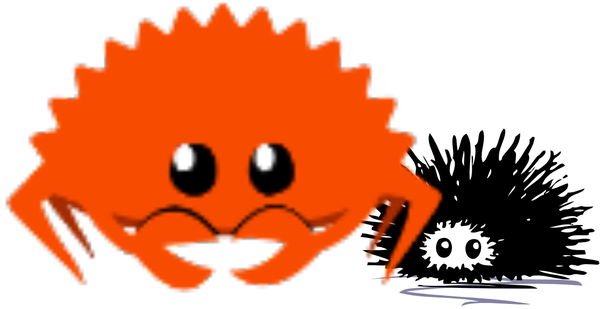





DARPA: Translating All C to Rust (TRACTOR)
blonk
For those who care
After more than t
software enginee
finding tools. The
unsafe programs
issues.
The TRACTOR p
is to achieve the
eliminating the er
This program ma
and dynamic ana
Additional informa
Not sure that using
are people who are
Ownership guided C to Rust translation
Hanliang Zhang[1], Cristina David[1], Yijun Yu[2], and Meng Wang[1]
[1] University of Bristol
Empirical Software Engineering (2025) 30:3
https://doi.org/10.1007/s10664-024-10573-2
Check for updates
Type-migrating C-to-Rust translation using a large language model
Jaemin Hong[1] · Sukyo
Accepted: 10 October 2024 / Pub
© The Author(s) 2024
Abstrac
that com
bination
growing
this pap
static ow
ership m
into safe
world co
achieves
1 Introdu
Rust [28] is a m
tion of memor
from ownershi
certain memo
pointers to th
Artifacts Available
acm
V1.1
Artifacts Evaluated
acm
Functional V1.1
Scalable, Validated Code Translation of Entire Projects using Large Language Models
HANLIANG ZHANG, University of Bristol, United Kingdom
CRISTINA DAVID, University of Bristol, United Kingdom
MENG WANG, University of Bristol, United Kingdom
BRANDON PAULSEN, Amazon, USA
DANIEL KROENING, Amazon, USA
Abstract
Large language models (LLMs) show promise in code translation due to their ability to generate idiomatic code. However, a significant limitation when using LLMs for code translation is scalability: existing works have shown a drop in translation success rates for code exceeding around 100 lines. We overcome this limitation by developing a modular approach to translation, where we partition the code into small code fragments which can be translated independently and semantically validated (that is, by checking I/O equivalence). When this approach is applied naively, we discover that LLMs are unreliable when translating features of the source language that do not have a direct mapping to the target language, and that the LLM often gets stuck in repair loops when attempting to fix errors. To address these issues, we introduce two key concepts: (1) feature mapping, which integrates predefined translation rules with LLM-based translation to guide the LLM in navigating subtle language differences and producing semantically accurate code; and (2) type-compatibility, which facilitates localized checks at the function signature level to detect errors early, thereby narrowing the scope of potential repairs. We apply our approach to translating real-world Go codebases to Rust, demonstrating that we can consistently generate reliable Rust translations for projects up to 9,700 lines of code and 780 functions, with an

Are We Safe Now?









Are We Safe Now?

Potential issues when crossing language boundaries between Rust and C:

- Type Mismatch: Layout and Representation
- Error Handling
- Panics and unwinding
- Threading and synchronization
- Memory ownership and lifetime mismatches
- Undefined behavior leaks across the boundary
- Aliasing & mutability violations

It’s the developer’s responsibility to avoid UB in unsafe code (FFI included), by maintaining Rust’s type invariants.

Are We Safe Now?

Potential issues when crossi

- **Type Mismatch: Layout**
- Error Handling
- Panics and unwinding
- Threading and synchroni
- Memory ownership and
- Undefined behavior leak
- Aliasing & mutability viol

It's the developer's responsibility to
maintaining Rust's type i

The Embedded Rust Book

A little C with your Rust

Using C or C++ inside

- Wrapping the ex
- Building your C

As C++ does not have
ABI when combining

Defining the i

Before consuming C o
function signatures ex
file which defines this
Rust, or use a tool to g

First, we will cover ma

Wrapping C funct

Typically, libraries writ

The bindgen User Guide

Introduction

`bindgen` **automaticall**

For example, given the

```
typedef struct CoolS
    int x;
    int y;
} CoolStruct;

void cool_function(
```

`bindgen` produces Rust
types:

```
/* automatically ge

#[repr(C)]
pub struct CoolStruc
    pub x: ::std::os
    pub y: ::std::os
}

extern "C" {
    pub fn cool_func
```

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATE usenix ASSOCIATI FUNCTION

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 conven-

engineers. Pragmatic development does not immediately d
card such thoroughly tested code but incrementally replac
it with safe languages when new features or complex fix
would otherwise be required.

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

Recent contributions propose isolating foreign code in se
arate protection domains [4, 16, 29, 35]. Unfortunately, me
ory isolation *alone* is insufficient to maintain safety. Diff
ences in semantics across languages mean that even int
actions with internally correct foreign libraries—ones th

Are We Safe Now?

Potential issues when crossing language boundaries between Rust and C:

- Type Mismatch: Layout and Representation
- Error Handling
- Panics and unwinding
- Threading and synchronization
- Memory ownership and lifetime mismatches
- Undefined behavior leaks across the boundary
- Aliasing & mutability violations

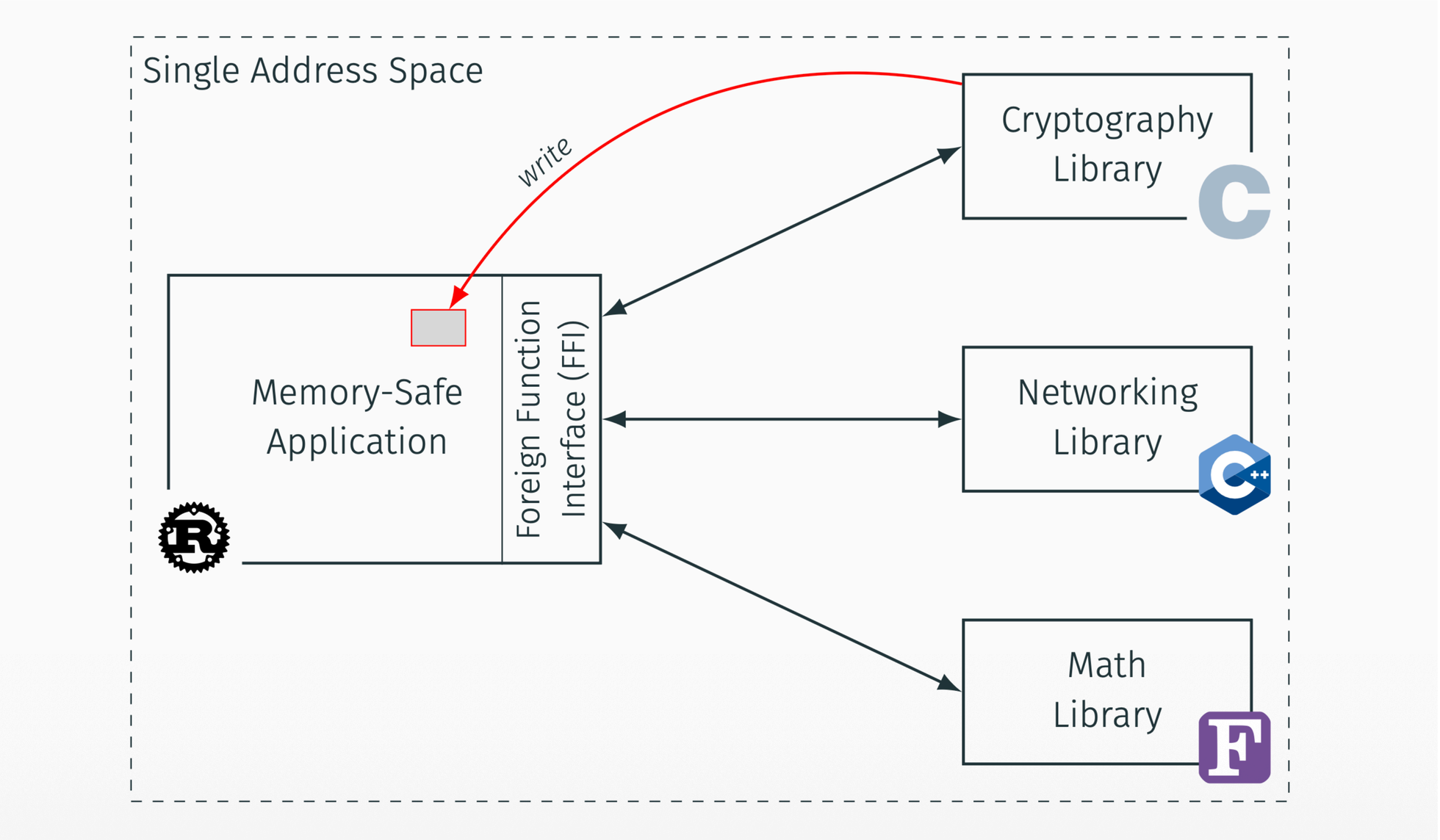
It’s the developer’s responsibility to avoid UB in unsafe code (FFI included), by maintaining Rust’s type invariants.

Are We Safe Now?

Potential issues when crossing language boundaries between Rust and C:

- Type Mismatch: Layout and Representation
- Error Handling
- Panics and unwinding
- Threading and synchronization
- **Memory ownership and lifetime mismatches**
- **Undefined behavior leaks across the boundary**
- **Aliasing & mutability violations**

It's the developer's responsibility to avoid UB in unsafe code (FFI included), by maintaining Rust's type invariants.



Leon Schuermann, Jack Toubes, Tyler Potyondy, Mae Milano, Amit Levy, Safe Interactions with Foreign Languages through Encapsulated Functions, *RustNL 2024*

Verification and UB Detection

FORMAL REASONING:

- RustBelt, Prusti, Aeneas, Cr
RUXt, Verus, etc

BUG DETECTION TOOLS:

Miri

- *detects UB (does not ensure*
- *requires a good test suite*
- *only supports Rust code (no*

Soteria

- *symbolic execution engine fo*



Verification and UB Detection

FORMAL REASONING:

- RustBelt, Prusti, Aeneas, Creusot, Kani, Mirai, Rudra, CompCert + Rust, SeaHorn + Rust, RUXt, Verus, etc

BUG DETECTION TOOLS:

Miri

- *detects UB (does not ensure absence of UB)*
- *requires a good test suite*
- *only supports Rust code (no C FFI)*

Soteria

- *symbolic execution engine for Rust and for C, b*

MiriLLI

- extends to C, but it's slow, requires good test s

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 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 47 instances of undefined or undesired behavior from 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.**

- **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 47 instances of undefined or

The Safety Pillars of Rust's Type System

```
let mut x = String::from("IFIP"); // ownership, mutability

let y = &x;                       // borrowing - RO

let z = &mut x;                   // borrowing - W
```

Safety Principle 1

Every value has a variable that's called its owner. There can only be one owner at a time.

Safety Principle 2

The lender needs to outlive all of its (alive) references.

Safety Principle 3

A mutable borrow can only be created if its lender has no other borrows living at that time.

Safety Principle 4

The lender cannot be modified as long as one of its (shared) borrowers still lives.

Example 1

```rust
fn add_one(i: &i32) {
    *i += 1;
}

fn
}
```

```
error[E0594]: cannot assign to `*i`, which is behind a `&` reference
 --> src/main.rs:2:2
  |
2 |     *i += 1;
  |     ^^^^^^^ `i` is a `&` reference, so the data it refers to cannot be written
  |
help: consider changing this to be a mutable reference
  |
1 | fn add_one(i: &mut i32) {
  |                +++
```



```c
void add_one(int *i)    {
    *i += 1;
}

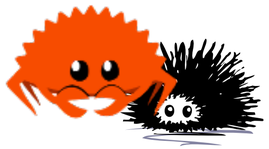
main() {
t i = 41;
d_one(&i);
intf("%d", i);
turn 0;
```



Example 1 - ext

```rust
unsafe extern "C" {
  fn add_one(i: &i32);
}

fn main() {
  let i = 41;
  unsafe { add_one(&i); }
  println!("{}",i);
}
```



```c
void add_one(int *i)    {
  *i += 1;
}

int main() {
  int i = 41;
  add_one(&i);
  printf("%d", i);
  return 0;
}
```

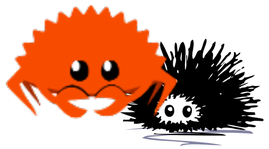


Example 2

```
unsafe extern "C" {
  fn foo(i: &i32, b: i32);
}

fn main() {
  let p = Box::new(42);
  let b = // some random i32
  unsafe { foo(&*p, b); }
  println!("{}",p);
}
```

```
void foo(int *i, int b)    {
  if(b>10 && <cond>) {
     free(i);
  }
}
```



UAF

DF

Invalid cross-language deallocation

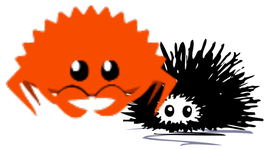


Working towards *a* solution

Obs1: use type info to generate dynamic checks

```
unsafe extern "C" {
   fn add_one(i: &i32);
}
```

```
void add_one(int *i)    {
   *i += 1;
}
```





Obs1: use type info to generate dynamic checks

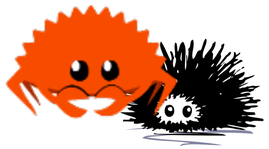
```
unsafe extern "C" {
   fn add_one(i: &i32);
}
```

RO

```
fn test_add_one(i: &i32) {
   let x = *i;
   add_one(i);
   assert_eq!(x,*i);
}
```

Rust Type Constraints

```
void add_one(int *i)    {
   *i += 1;
}
```



What about mutable borrow xor?



Obs2: instead of checking only the final state, observe behavior in each state

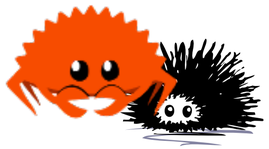
```
unsafe extern "C" {
   fn add_one(i: &i32);
}

#[proptest]
fn test_add_one(I: &i32) {
   let x = *i;
   add_one(i);
   assert_eq!(x, *i);
}
```

RO

```
void add_one(int *i)    {
   *i += 1;
}
```

Rust Type Constraints



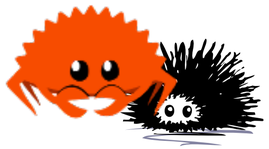
We replace standard execution with MiriLLI interpretation



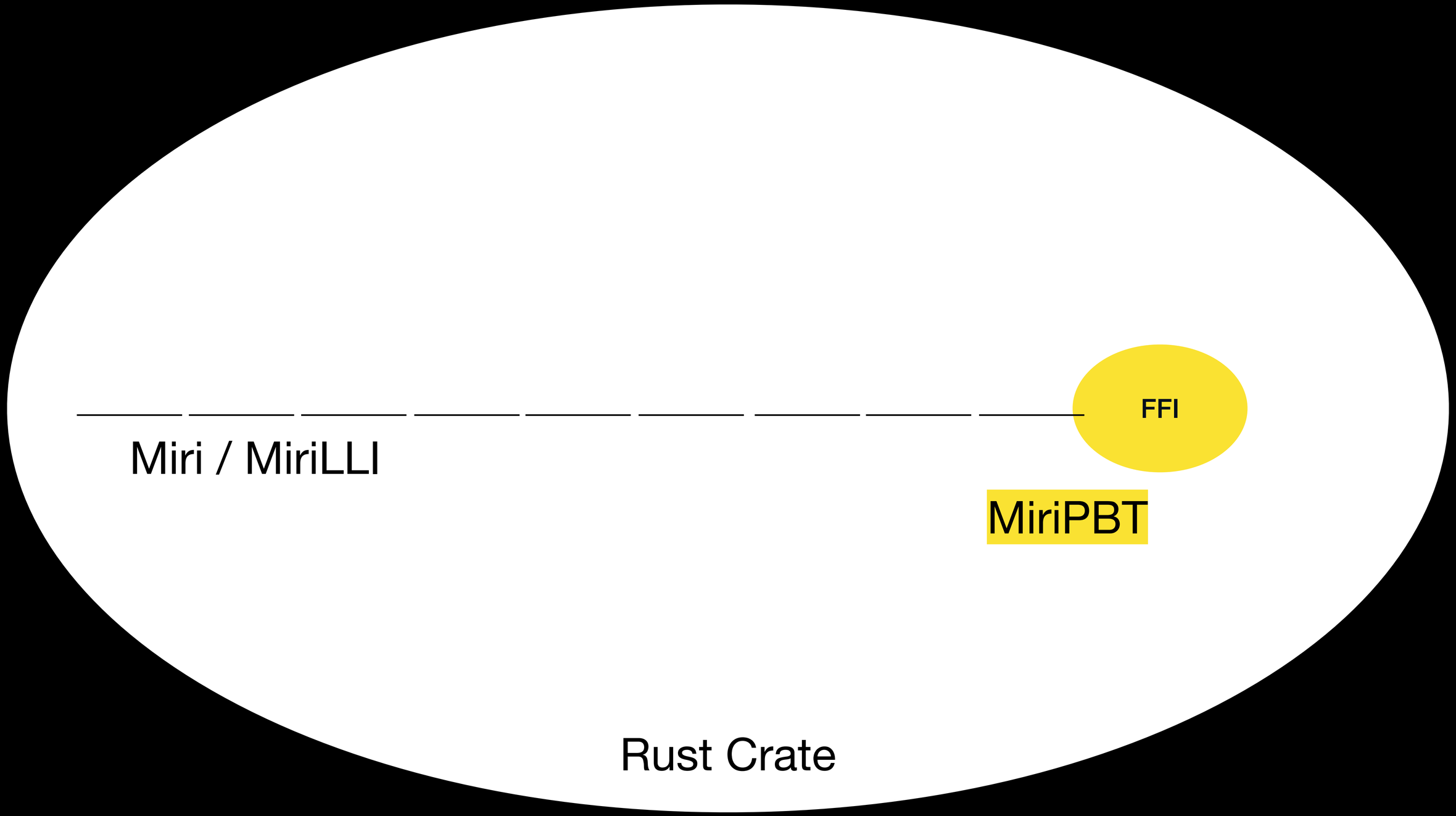
```
unsafe extern "C" {
    fn add_one(i: &i32);
}

#[proptest]
fn test_add_one(i: &i32) {
    let x = *i;
    add_one(i);
    assert_eq!(x, *i);
}
```

RO



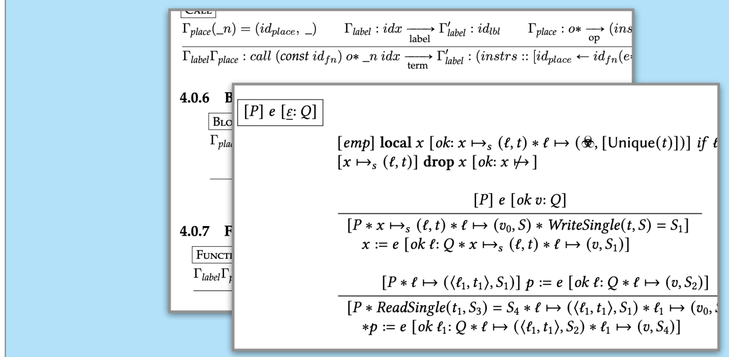
1. Use type information to generate a PBT.
2. Use MIRI to guide the execution to the FFI call.
3. “Fuzz” the FFI call using MiriLLI to check for type invariants violations.


FFI
Miri / MiriLLI
MiriPBT
Rust Crate

Borrows DSL

- instantiate
- validate
- compile

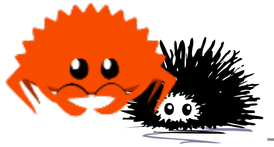
Tree / Stack Borrows + Incorrectness Separation Logic ⊢ Infer



Textual

ULLBC

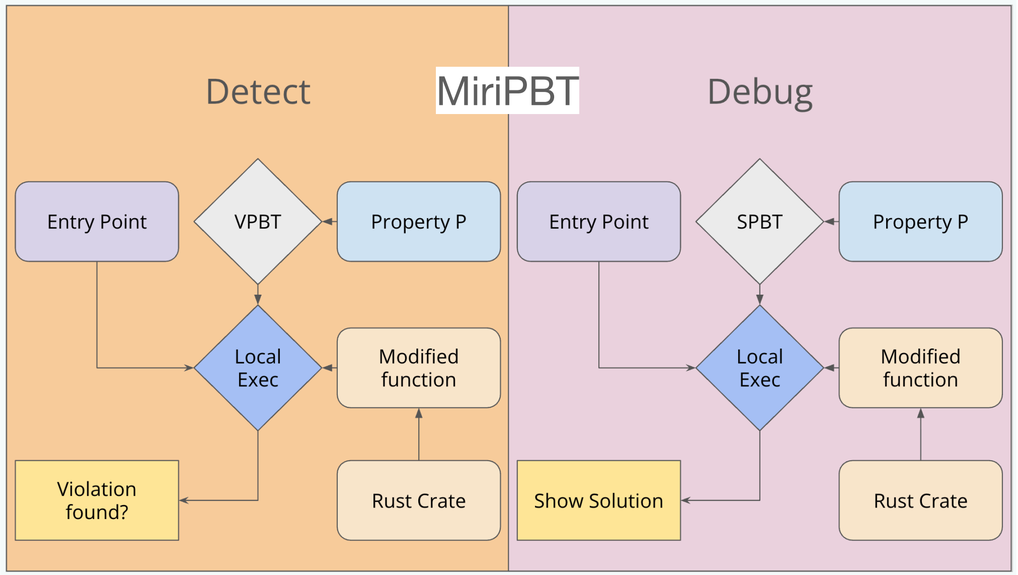
Charon



Source → AST → HIR → MIR → LLVM → Machine code

Source → AST → LLVM

Miri MiriPBT MiriLLI

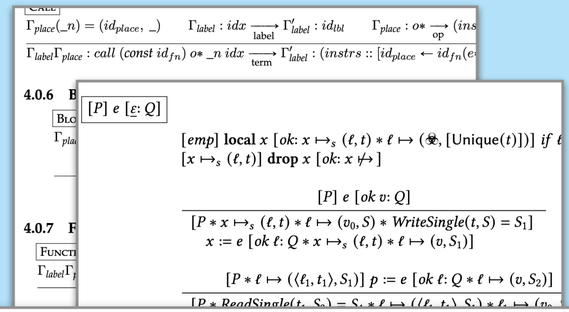


Arjan Seijs *(MSc TU Delft)*
Matei Mirica *(MSc TU Delft)*
Julius de Jeu *(MSc TU Delft)*
Glenn Weeland *(MSc TU Delft)*
Darius Foo *(PhD, National University of Singapore)*
Taha Kahya *(BSc, Sabancı University of Istanbul)*
Jana Dönszelmann *(Rust T-compiler member)*
David Pichardie *(Meta, ENS Rennes)*

Borrows DSL

- instantiate
- validate
- compile

Tree / Stack Borrows + Incorrectness Separation Logic ⊢ Infer

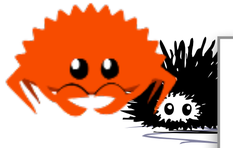


Textual

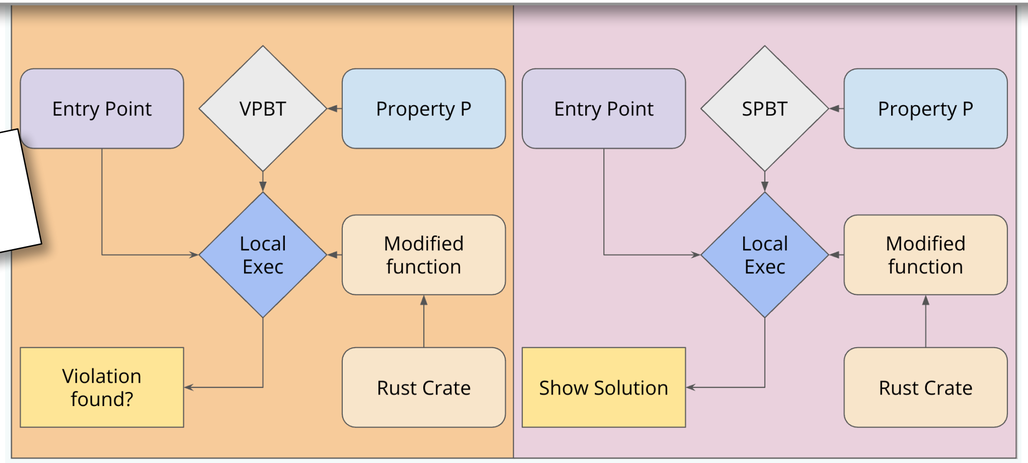
ULLBC

Source

AST



Detect and debug UB in Rust–C interoperability.



Thank You!

Arjan Seijs *(MSc TU Delft)*
Matei Mirica *(MSc TU Delft)*
Julius de Jeu *(MSc TU Delft)*
Glenn Weeland *(MSc TU Delft)*
Darius Foo *(PhD, National University of Singapore)*
Taha Kahya *(BSc, Sabancı University of Istanbul)*
Jana Dönszelmann *(Rust T-compiler member)*
David Pichardie *(Meta, ENS Rennes)*

Summary

Highlight potential vulnerabilities in Rust–C interoperability.

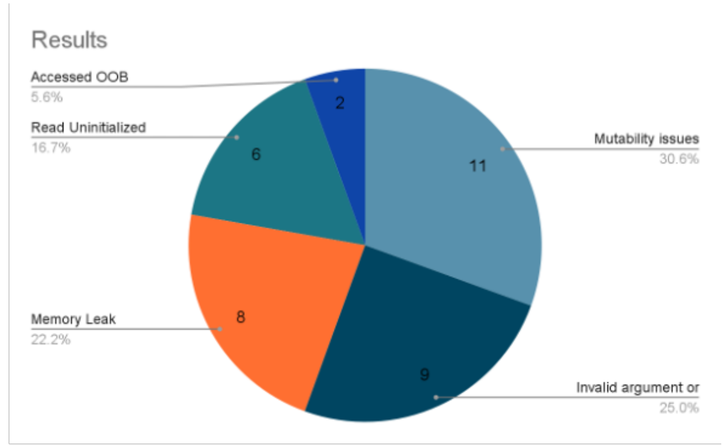
- Design a hybrid approach for detecting UB across Rust–C boundaries by:
  - leverage Rust type system to reduce the search space,
  - automate detection,
  - reduce human effort,
  - aim detection, not verification.

- Much help needed.

Preliminary Eval

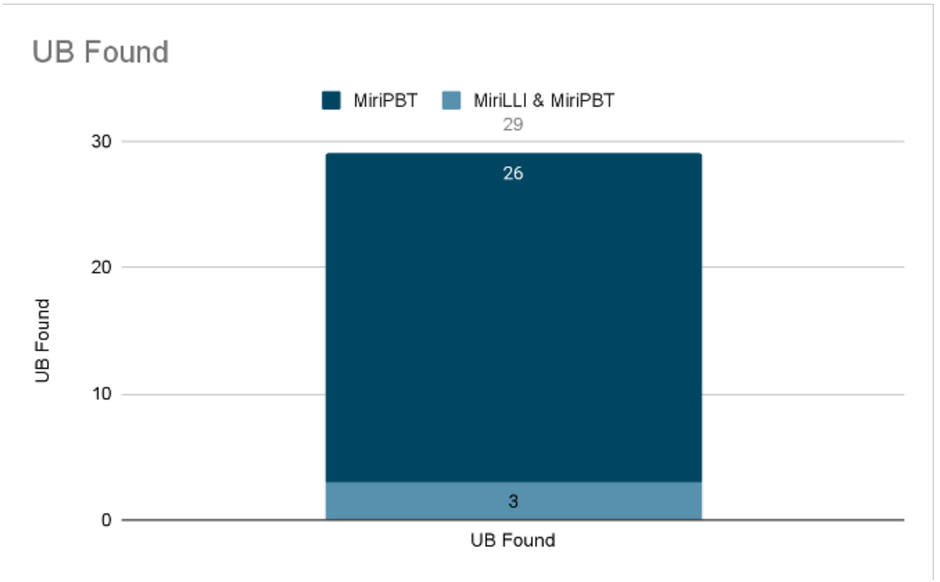
[dataset 1] 30 crates with 55 tests from the original MiriLLI dataset.

Results: MiriLLI and MiriPBT find the same number of bugs.



[dataset 2] Modified the C libraries used in the 30 crates to introduce synthetic bugs that only manifest on certain paths.

Results: MiriPBT finds more bugs than MiriLLI.



MiriPBT

Find UB (mem leak, UAF, DF, Aliasing XOR mutability) in Rust-C FFI.

Reduce the human effort required to write test cases that would cover all paths in the C call.

Developed as a MSc Research Project:
- **Julius de Jeu** (TU Delft),
- **Jana Dönszelmann** (Co-founder of Hexcat and Rust T-compiler member)

TODO:
- More aggressive evaluation on real crates.
- “Fuzz” the calling environment.

The Static Analysis way on Rust FFI

Goal: find memory safety violations in unsafe Rust and in Rust-C FFI.

Approach: Incorrectness Separation Logic for Rust.

- [unsafe Rust] **front-end for Rust in Infer/Pulse.**

Arjan Seijs *(MSc TU Delft),* ***Darius Foo*** *(National University of Singapore),*
Taha Kahya *(Sabancı University of Istanbul),* ***David Pichardie*** *(Meta, ENS Rennes)*

- [unsafe Rust] **extend ISL with stack/tree borrows.**

Matei Mirica *(MSc TU Delft),* ***Darius Foo*** *(National University of Singapore),*
David Pichardie *(Meta, ENS Rennes)*

- [Rust FFI] **compose Rust-ISL states with C-ISL, and with MiriPBT.**