A Study of Undefined Behavior Across FFI Boundaries in Rust Libraries

Muhammad Hassnain

April 22nd, 2025

Recap

- FFI is the mechanism by which a program written in one programming language can call and use services written or compiled by another programming language.
- Undefined Behaviour refers to situations in a programming language where the language standards imposes no requirement on what should happen.

Some Rust Rules



- Each value has a unique owner.
- Ownership can be transferred through assignment or borrowed by creating a reference.
- Immutable references can be copied while mutable references have move.
- Lifetime of a references must not exceed the scope of the value.
- A value can have a single mutable reference multiple immutable references in one lifetime but not both.

Playground Link

Stacked Borrow



- A proper usage of mutable reference follows a stack discipline.

1. Use a stack to store pointer identifiers.
2. On each reborrow a new identifier is pushed to the stack.
3. A pointer can be used if its identifier is in the stack.
4. Using a pointer pops everything above it.

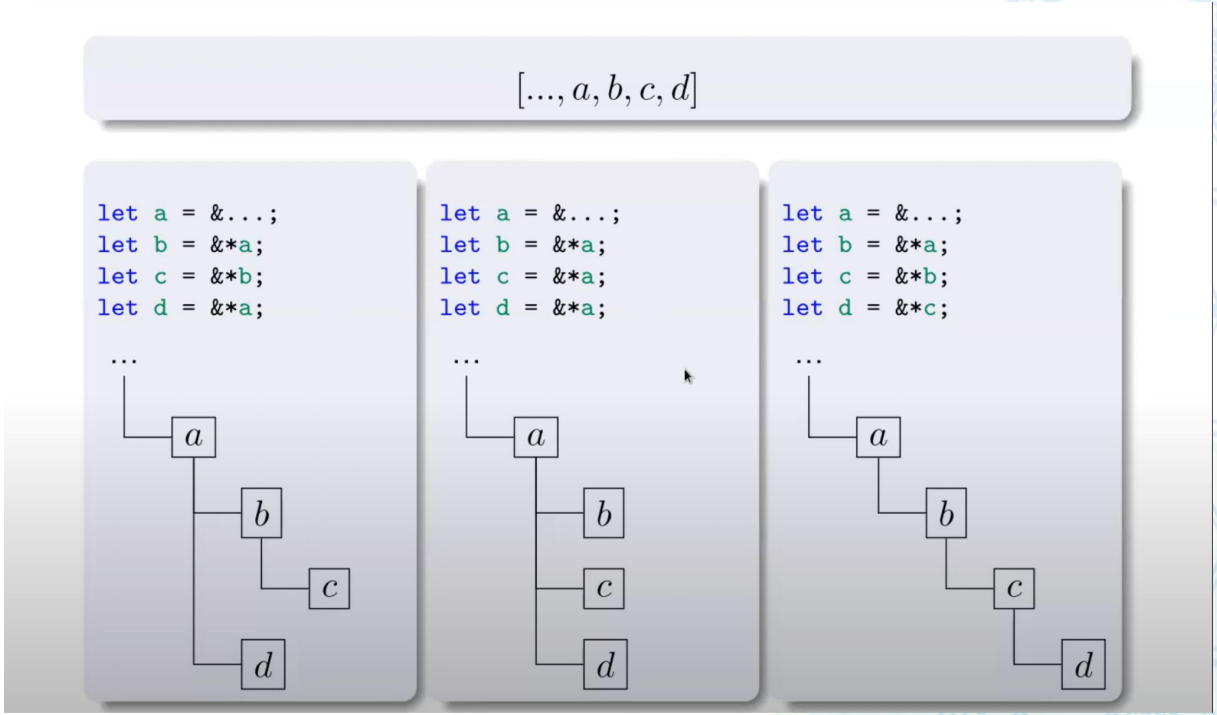
An example : [Playground Link]

Stacked Borrow

Information is lost in the stack!!!

Stack borrow cares more about the order in which we created the pointer rather than their exact relationship with each other.

….We can extend it into a tree to encode this additional dimensional information.



[Tree Borrows: An aliasing Model for Rust]

Tree Borrow



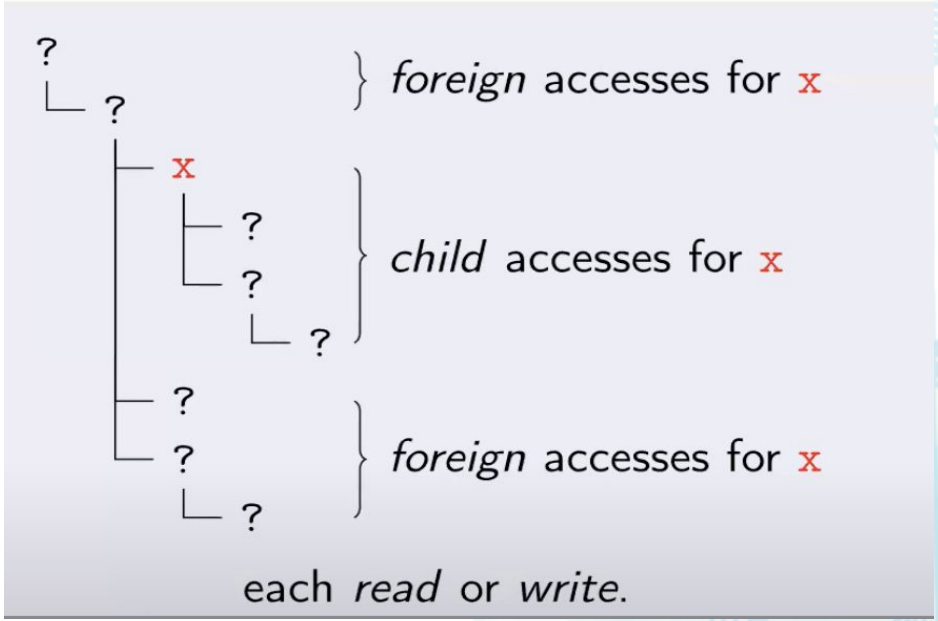
- A proper usage of all pointers follows a tree discipline.

1. Use a tree to store pointer identifiers.
2. On each reborrow a new identifier is added as leaf of the tree.
3. Each pointer has permissions.
4. A pointer can be used if its permissions allows it.
5. Using a pointer makes incompatible pointers lose information.

Tree Borrow : Types of Access

A tree borrow keeps track of permissions in two ways.

It will update the permission based on the type of access.



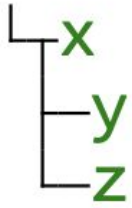
[Tree Borrows: An aliasing Model for Rust]

Aliasing Violations in Tree Borrows

1. **Expired Permission Error:** Writing to a memory location causes other permissions to expire. (transition to disabled). Using these expired permissions is undefined behavior.

```
1   let mut x = 0;
2   let y = &mut x;
3   let y = y as *mut _;
4   let z = &mut x;
5   unsafe { *y = 1; }
6   *z = 0;
```

Active		
Reserved	→	Active
Reserved	→	Disabled



Aliasing Violations in Tree Borrows

2. **Insufficient Permission Error:** Using a reference with a read only permission to write to memory.

```
1   let x = 0;
2   ub(&x);
3   fn ub(x: *const u8) {
4    let x = x as *mut _;
5    unsafe { *x = 1 };
6   }
```

Aliasing Violations in Tree Borrows

3. **Protected Permission Error:**

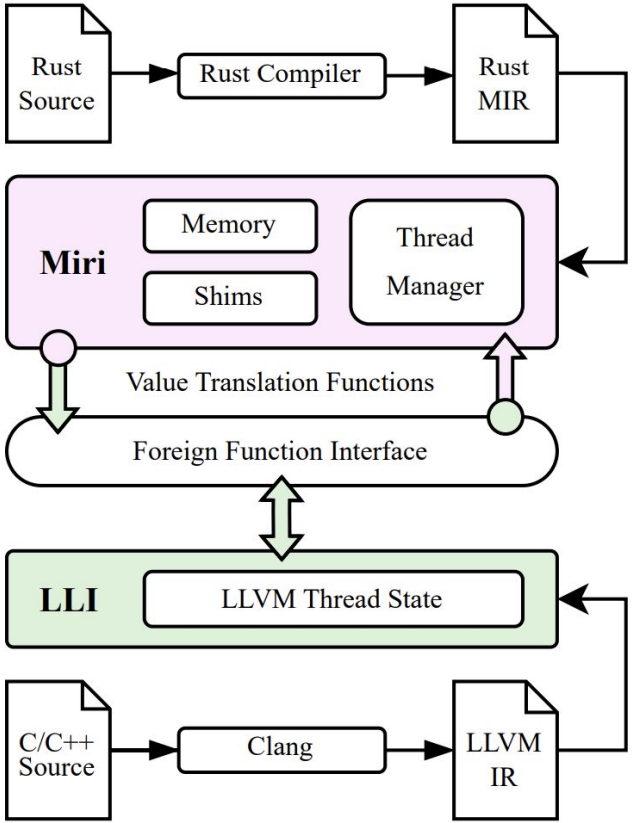
   This happens when the permissions of a pointer obtained as an argument is transitioned into disabled!

```
1   unsafe fn free(x: &mut u8, layout: Layout) {
2     dealloc(x as *mut _, layout);
3   }
```

Overview

9130 test cases.

- 61% unsupported.
- 19% passed.
- 10% timed out.
- 1% failed.
- 9% bug.



Results

46 bugs in 37 crates.

- 23 occurred in LLVM, 17 in Rust, 6 due to incorrect bindings.

32 bugs were found in crates with less than 100 average daily downloads.

3 bugs in crates with average 10000 daily downloads.

		Category			
Fix	**Error**	**Allocation**	**Ownership**	**Typing**	***Total:***
Binding	Binding	-	-	6	6
Binding	LLVM	-	3	-	3
LLVM	LLVM	-	3	-	3
Rust	LLVM	1	16	-	17
Rust	Rust	9	2	6	17
	Total:	10	24	12	46

Ownership Bugs

24 ownership bugs from 20 crates.

1. **Const Correctness:** insufficient permission errors caused by incorrectly casting immutable references to mutable raw pointers.
2. **Self Reference:** example on next page

Ownership Bugs

UnsafeCell can be mutated through shared references so, a and cache share the same permission.

```
1   // Rust Encapsulation
2   struct Alloc {
3   - cache: i32,
    + cache: UnsafeCell<i32>,
4     buffer: *mut i32,
5   }
6   fn open(a: &mut Alloc [1] (1)) -> i32 {
7   - let cache = &mut a.cache as *mut _; (2)
    + let cache = a.cache.get(); [1]
8     a.buffer = cache;
9     let b = &mut *a; [2] (3)
10    unsafe {
11      ffi::open(b.buffer [1] (2));
12    - b.cache (3)
      + *b.cache.get() [2]
13    }
14  }

15  // C API
16  void open(int *buffer [1] (2)) { // ffi::open
17    *buffer = 1;
18  }
```

(a) Tables illustrating the (simplified) subtree of permissions for location of the field "a.cache" before (to the left of the arrow) and after (to the right of the arrow) the write access on line 17.

Reserved	→	Active
Reserved	→	Active
Reserved	→	Disabled



Reserved*	→	Active
Reserved*		



Ownership Bugs

Multiple Mutable Aliasing

This is like an expired permissions error and occurs when pointers derived from mutable references were copied into foreign heap.

```
1   // Rust Encapsulation
2   impl Compression {
3     fn new() -> Self {
4       let mut stream = Box::new(Stream::default());
5     + let stream = Box::into_raw(stream);
6       unsafe {
7       - ffi::init(stream.as_mut()); ①
        + ffi::init(stream); [1]
8       }
9       Compression { stream }
10    }
11
12    fn compress(&mut self ② [2]) {
13      unsafe {
14      - self.stream.data = 0; ②
        + (*self.stream).data = 0; [1]
15      - ffi::compress(self.stream.as_mut()); ②
        + ffi::compress(self.stream); [1]
16      }
17    }
18  }

19  // C API
20  void init(Stream* s ① [1]) { // ffi::init
21    s->state = malloc(sizeof(State));
22    s->state->stream = s;
23  }
24  void compress(Stream* s ② [1]) { // ffi::compress
25    int data = s->state->stream->data; ① [1]
26  }
```

(a) Tables illustrating the (simplified) subtree of permissions fo location of the field `stream.data` before (to the left of the arrow and after (to the right of the arrow) the write access on line 14.

Reserved	→	Disabled	①		Active		[1]
Reserved	→	Active	②		Reserved	→	Disabled	[2]

Ownership Bugs

for (; *curr==0 && curr+3 < end ; curr+=4)

→ The first term in the conjunction will execute before the second term, but if the second term is false, then the first term is an access out-of-bounds or an uninitialized read. (This was fixed by swapping the order of each term.)

…..paper also has some other such examples.

Typing Bugs

There were mostly related to incorrect initialization or padding issues - I am skipping this.

Allocation

1. **Memory Leaks**
   a. They find 10 examples.
   b. 5 are because of into_raw and from_raw. (example)
   c. 5 leaks are because of memory allocated in C and C does not expose a destructor to Rust.
2. **Cross Language Deallocation**
   a. ‘Potentially’ undefined behavior to free a pointer allocated by C in Rust or vice versa.
   b. Ownership might be transferred to a Rust pointer after the the FFI call, and the borrow checker will try to deallocate the memory once the pointer goes out of scope. (which is deallocating C heap memory using Rust allocator).