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

IAN MCCORMACK, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA

The Rust programming language restricts aliasing and mutability to provide static safety guarantees, which developers rely on to write secure and performant applications. However, Rust is frequently used to interoperate with other languages that have far weaker restrictions. These languages support cyclic and self-referential design patterns that conflict with Rust's aliasing model, representing a potentially significant source of undefined behavior that has remained understudied due to limitations in tooling. We created MiriLLI: a tool which can detect this type of undefined behavior by using Rust and LLVM interpreters to jointly execute multi-language applications. We used our tool in a large-scale study of Rust libraries that call foreign functions, and we found 45 instances of undefined or undesirable behavior. These include four bugs from libraries that had over 10,000 daily downloads on average, one from a component of the GNU Compiler Collection (GCC), and one from a library maintained by the Rust Project. Most of these errors were caused by incompatible aliasing and initialization patterns, incorrect foreign function bindings, and invalid type conversion. The majority of aliasing violations were caused by unsound operations in Rust, but they occurred in foreign code. The Rust community must invest in new tools for validating multi-language programs to ensure that developers can easily detect and fix these errors.

CCS Concepts: • **Software and its engineering → Interpreters**; **Software maintenance tools**; *Software verification and validation*; *Software defect analysis.*

Additional Key Words and Phrases: Rust, LLVM, C, C++, Interoperation, Foreign Function Calls, Aliasing, Mutability, Undefined Behavior

1 INTRODUCTION

The Rust programming language has been increasingly popular due to its static safety guarantees, which provide security benefits comparable to garbage-collected languages without additional run-time overhead [Pereira et al. 2017; Stack Overflow 2023]. However, Rust is frequently used to interoperate with other languages that lack its restrictions on aliasing and mutability. Foreign function calls are one of several `unsafe` features that developers can enable to bypass these restrictions when they become burdensome. It is difficult to use `unsafe` features correctly, as they have been a significant source of security vulnerabilities for the Rust ecosystem [Xu et al. 2021]. Foreign function calls present unique challenges in this area, since Rust's operational semantics inhibit the use of certain aliasing idioms that can be used freely in other languages.

We seek to determine whether these differences in semantics are causing undefined behavior in real-world programs. We created MiriLLI, a tool which extends two pre-exising Rust and LLVM interpreters to jointly execute programs and detect violations of Rust's aliasing model. We conducted a large-scale evaluation of our tool on all compatible test cases from Rust libraries that called foreign

*This material is based upon work supported by the Department of Defense and the National Science Foundation under Grant Nos. CCF-1901033, DGE1745016, and DGE2140739. Results presented in this paper were obtained using CloudBank [Norman et al. 2021], which is supported by the National Science Foundation under Grant No. CNS-1925001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of Defense, the National Science Foundation, or CloudBank.

Authors' addresses: Ian McCormack, Carnegie Mellon University, USA, icmccorm@cs.cmu.edu; Joshua Sunshine, Carnegie Mellon University, USA, sunshine@cs.cmu.edu; Jonathan Aldrich, Carnegie Mellon University, USA, jonathan.aldrich@cs.cmu.edu.

functions. We found 9130 tests from 957 libraries that executed foreign code with our tool, and we identified 45 instances of undefined or undesirable behavior from 35 of these libraries.

Our results indicate that Rust's restrictions on cyclic and self-referential patterns make it easy to inadvertently introduce undefined behavior through safe operations in the encapsulation of foreign libraries. Tree Borrows violations were the most common type of bug that we detected, followed by memory leaks and uses of uninitialized memory. Developers can take immediate steps to avoid these errors by auditing their use of certain types at foreign function boundaries. However, the Rust Project must invest in new tools to ensure that these errors can be easily detected.

Overview. In Section 2, we compare current and past models of Rust's operational semantics with C and C++ and describe the resources and best-practices that Rust developers can use to ensure correct interoperation with foreign libraries. In Section 6, we review prior work on Rust verification and interoperation. In Section 3, we document our methodology for sampling and evaluating test cases from Rust libraries that call foreign functions. We provide a formal model of our extension and describe how we account for differences between the interfaces and semantics of LLVM and Rust. In Section 4, we describe each of the unique bugs that we encountered. We discuss the implications of our findings in Section 5, and we conclude in Section 7.

2 BACKGROUND

Rust's safety restrictions are enforced by its borrow checker, which validates the Middle Intermediate Representation (MIR) of Rust programs before they are lowered into the LLVM intermediate representation [Klabnik and Nichols 2022]. These restrictions begin at the level of a value, which is valid for a particular scope. A value's type implements 'traits' that define its behavior. For example, all types have "move semantics" by default, denoting that each value has a unique owner. However, values of types that implement the `Copy` trait have no owner and can be copied freely.

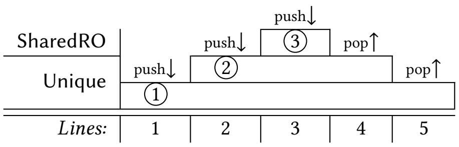
Ownership can be transferred through assignment or *borrowed* by creating a reference. References are either mutable, taking the form `&mut T`, or immutable, with the form `&T`. Mutable references have move semantics, but immutable references can be copied. Each reference has a "lifetime," which is the portion of the program over which it will be used. The borrow checker asserts that the lifetime of a reference cannot exceed the scope or lifetime of the value that it refers to. A value can only have a single mutable reference or many immutable references that are active within a given context, but not both at the same time [Crichton 2020].

These restrictions statically eliminate data races and use-after-free errors, but they can be overly conservative for certain design patterns. When developers need to bypass Rust's safety restrictions, they can use the `unsafe` keyword to enable a set of additional features that are not restricted by the borrow checker within a limited context. Developers can use these features to access raw pointers and the fields of union types, modify static mutable state, implement `unsafe` traits, and call `unsafe` functions—including those written in other languages [Klabnik and Nichols 2022]. To avoid introducing security vulnerabilities, developers are encouraged to mitigate the risks of using `unsafe` code by keeping it local, minimal, and well-encapsulated [Qin et al. 2020].

The Rust compiler assumes that programs satisfy the constraints of the borrow checker. These assumptions can be leveraged to apply behavior-preserving transformations that may lead to significant performance gains [Ghiya et al. 2001]. However, if a program violates these assumptions using `unsafe` code, then compile-time optimizations may break its semantics, leading to security vulnerabilities and differences in functionality. Determining what should constitute undefined behavior is not a trivial task; Jung [2021] frames undefined behavior as a "double-edged sword" for compiler developers, who must balance performance and usability. If every possible behavior is valid, then the compiler cannot assume much, but the definition of defined behavior is too

Fig. 1. A valid Rust code snippet (left) and its corresponding borrow stack (right). The first column of the table on the righthand side indicates the type of permission represented by each of the blocks, which correspond to the permissions created by each annotated location in the snippet. Permissions are pushed (push↓) and popped (pop↑) at each of the lines in the snippet indicated by the last row.

```
1  let mut x = 0; ①
2  let y = &mut x; retag y; ②
3  let z = &*y; retag z; ③
4  *y = 1
5  x = 0;
```



restrictive, then it may be difficult to write correct programs. For example, Yodaiken [2021] outlines how the ISO standard of C has become near-unusable for operating systems development, since its overeager optimization leads to unexpected behavior in trivial situations.

The Stacked Borrows [Jung et al. 2019] and Tree Borrows [Villani et al. 2023] models of Rust's operational semantics attempt to address this problem. Both rely on the notion that the origin, or *provenance* of a pointer can be used to determine if a memory access is valid [Jung 2024]. A pointer's provenance value must be associated with the location being accessed and it must grant permission for the type of access taking place. The precise meaning of these conditions varies under each model, but if either is unmet, then a memory access is considered to be undefined behavior.

2.1 Stacked Borrows

The Stacked Borrows [Jung et al. 2019] model of Rust's operational semantics follows from the observation that the lifetimes of references begin and end in a manner similar to objects being pushed and popped from a stack. In this model, every pointer consists of an address and a "borrow tag," which is associated with a permission "item." Items are contained in a stack associated with each location. Borrowing a location pushes an item onto the stack, and when a reference is no longer valid, its permission is popped from the stack. Every read, write, allocation, and deallocation operation will validate and update the borrow stack for the location being accessed.

Figure 1 provides a minimal example that demonstrates the stack discipline of Rust's reference types. When the variable `x` is declared on Line 1, a new stack is created containing a Unique item with the tag ①. The item Unique denotes unique, mutable access to a location. Mutable references are Unique, mutable pointers have shared, mutable access (SharedRW), and both immutable pointers and immutable references have shared, read-only access (SharedRO). Variables start as either Unique or SharedRW, depending on their mutability.

New permissions are created and pushed into the stack by the **`retag`** operation. This is not present in written code, but it can be emitted by the Rust compiler for error checking. A **`retag`** is inserted every time a reference is created, assigned, or cast into a raw pointer. We include retags in Figures 1 and 2d, but elide them elsewhere. In Figure 1, the variable `x` is mutably borrowed on Line 2 and then immutably reborrowed on Line 3. Each borrow is followed by a **`retag`**, pushing new items onto the stack with tags ② and ③. When `y` is used for a write access on Line 4, the tag ③ is popped from the stack. This enforces the uniqueness of mutable references by making it an error for ② and ③ to be used interchangeably within the same context. Similarly, writing through `x` on Line 5 invalidates ②, since a location cannot have more than one mutable reference or alias.

The program in Figure 1 does not use **`unsafe`** features, so Rust's borrow checker can statically prove that it has defined behavior. However, raw pointers have few static restrictions, so it is possible to invoke undefined behavior in **`unsafe`** contexts by accessing locations using permissions

Fig. 2. Minimal examples representing the distinct categories of Stacked Borrows and Tree Borrows violations that we consider in our evaluation.

(a) A code snippet containing an *invalid range* error, which is only undefined behavior under Stacked Borrows.

```
1  let mut point = (0, 0);
2  let x: &mut i32 = &mut point.0;
3  unsafe {
4    *((x as *mut i32).offset(1)) = 1; UB!
5  }
```

(b) A code snippet that produces an *insufficient permission* error under both Stacked Borrows and Tree Borrows.

```
1  let x: i32 = 0;
2  mutate(&x as *const _));
3  fn mutate(x: *const i32) {
4    unsafe {
5      *(x as *mut _) = 5; // UB!
6    }
7  }
```

(c) A code snippet that produces an *expired permission* error under Stacked Borrows but *not* under Tree Borrows.

```
1  let mut x: i32 = 0;
2  let y = &mut x as * mut _;
3  let z = &mut x;
4  unsafe { *y = 0; } // UB!
```

(d) A code snippet that produces a *framing* error under both Stacked Borrows and Tree Borrows.

```
1  unsafe fn free(x: & mut u8, layout: Layout) {
2    retag x;
3    dealloc(x as * mut u8, layout); // UB!
4  }
```

that are insufficient or expired. There are four categories of undefined behavior under the Stacked Borrows model which are relevant to our investigation. First, a reference cannot be used to access locations outside of the range it originally borrowed, even if the reference still points to the same allocation. This is illustrated by the example in Figure 2a, where the first field of the tuple `point` is borrowed to create a reference that is offset into the second field. Reading through this new reference is undefined behavior, since it originally borrowed from the first field. We refer to this type of violation as an *invalid range* error.

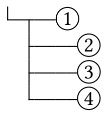
If an access is within bounds, the next step is to find the "granting item," [Jung et al. 2019] or permission, associated with the reference's tag. If that permission no longer exists in the stack, then the access is undefined behavior. As shown in Figure 1, items are be "popped" from the stack to assert that mutable permissions have unique access to a location during their lifetime. In the following example, `x` is mutably borrowed to create a new permission, which is then cast into a raw pointer (with "`as *mut _`") and assigned to `y`. This process involves two retags: once to create a unique mutable permission for `&mut x`, and then again to create a shared mutable permission when this expression is cast into a raw pointer. Before creating new permissions, retags also performs a simulated read or write access using the newly created permission. When `x` is mutably borrowed on Line 3, a third **`retag`** performs a write access through `x`, popping the mutable SharedRW permission held by `y`. This makes the write access on Line 4 undefined behavior, since the tag for `y` no longer exists in the stack. We refer to this type of error as an *expired permission* error.

Once a valid permission has been located, it must also permit the type of access taking place. In the following example, the variable `x` is immutably borrowed on Line 2 to create a reference that is cast into a **`const`** pointer and passed to the function `mutate`. On Line 5, in the body of `mutate`, it is cast into a **`mut`** pointer and mutated. Despite having a mutable type, this pointer retains a shared, read-only permission from when `&x` was cast into `*const i32` on Line 5, so it cannot be used for writing. We refer to this type of violation as an *insufficient permission* error

Fig. 3. An example illustrating the structure and semantics of permissions in Tree Borrows. Each cell in the table on the righthand side illustrates the state of the permissions associated with each circled borrow tag, before and after the write access on Line 5

```
1  let mut w: i32 ① = 0;
2  let x =  &mut w ② as *mut _;
3  let y =  &w; ③
4  let z =  &mut w; ④
5  unsafe { *x = 1; }
```

Active		
Reserved	→	Active
Frozen	→	Disabled
Reserved	→	Disabled



After finding a granting permission, the stack is modified based on the type of access taking place. A write access may cause tags to be popped from the stack, as shown in prior examples. However, certain permissions may be associated with a "protector". Protected permissions are created by special "function entry" retags, which apply to every reference-type argument at the beginning of the body of a function. It is undefined behavior for a protected tag to be popped from the stack before the associated call returns, at which point the protectors are removed. This asserts that the lifetime of each reference-type argument must be at least the duration of the body of the function. In Figure 2d, deallocating `x` destroys any stacks associated with its memory. However, the function-entry **`retag`** protects the permission associated with `x`, so destroying this permission is undefined behavior. We refer to an access that pops a protected tag as an *framing* error, since this mechanism is analogous to enforcing the "framing rule" from Smans et al. [2012]'s separation logic with implicit dynamic frames.

A memory access is *defined* behavior under Stacked Borrows if a stack exists for the location being accessed, a granting permission is present in the stack, and the access does not invalidate any protected permissions.

2.2 Tree Borrows

Tree Borrows [Villani et al. 2023] is an improved iteration on the Stacked Borrows model. Instead of items in a stack, permissions are nodes in a tree. Borrowing creates a permission that forms a new branch, which becomes the root of a subtree. For each subtree, a memory access is a *child access* if requires the root permission or any of its descendants; otherwise, it is a *foreign access.*

Tree borrows takes a lazy approach to managing permissions that allows several behaviors which are considered undefined under Stacked Borrows. Instead of creating a stack for each newly borrowed range of locations, every allocation has a single tree of permissions. Any permission in a tree can be used for any location within the bounds of its associated allocation. The example shown in Figure 2c is no longer undefined behavior under Tree Borrows, since the permission created by `&point.0` is lazily extended to apply to `point.1`. Also, instead of performing simulated reads and writes on retags, Tree Borrows only simulates read accesses, allowing multiple mutable references to coexist until the first one is used. The write access on Line 4 in Figure 2c is no longer undefined behavior under Tree Borrows, since the borrow on Line 3 no longer performs a write access.

Tree Borrows also uses different types of permissions, which are illustrated by the example in Figure 3. The variable `w` is borrowed multiple times before being mutated through `x`. The tree diagram on the right-hand side illustrates the permissions for each labelled reference before and after the write access on Line 5. The tree for `w` is initialized with an Active permission, which allows writes and is the default state for mutable locations. Each mutable borrow on Lines 2 and 4 creates a new branch in the tree with a Reserved permission, which is the initial, read-only phase of mutable references in this model. The immutable borrow assigned to `z` creates a branch with a Frozen permission, which is equivalent to SharedRO under Stacked Borrows. Since retags no longer perform write accesses, Tree Borrows allows each of the references assigned to `x`, `y`, and `z` to coexist up to the first write access through `x` on Line 5, at which point permission ② transitions from Reserved to Active, asserting unique access over the location of `x`. This is a *foreign* write access relative to ③ and ④, so the permissions for each of these tags become Disabled, which cannot be used for any memory access.

Unlike Stacked Borrows, new pointers or references of certain types will receive the same tag as the value that they were derived from. This includes raw pointers, mutable references to types covered by `Pin<T>`, and shared references to types covered by `UnsafeCell<T>`, which opts out of Rust's aliasing XOR mutability guarantee. Other characteristics of Tree Borrows remain the same as in Stacked Borrows. Reference types are given a protector on entry to each function, and a *framing* error occurs when a protected tag transitions to Disabled. Writing through a pointer with a Frozen permission is an *insufficient permission* error, and using a Disabled permission for any type of access is an *expired permission* error. As such, the examples in Figures 2b and 2d are both still undefined behavior under Tree Borrows.

2.3 Rust vs. C and C++

Rust shares several other categories of undefined behavior with C [ISO/IEC 2017] and C++ [ISO/IEC 2023] standards, such as accessing memory that has been freed, accessing beyond the bounds of an allocation, creating a data race, reading uninitialized memory, and accessing a value at an address that is not divisible by the alignment of its type. However, C and C++ have different rules related to initialization, aliasing, mutability, and provenance that may introduce undefined behavior when Rust is used in multi-language applications.

2.3.1 Initialization. In both C and C++, accessing an object in memory before it has been initialized produces an "indeterminate value", which is undefined behavior [ISO/IEC 2017, 2023]. There are a few exceptions, however; static and thread local variables are initialized immediately at the beginning of their lifetime, and both unsigned characters and bytes can soundly store indeterminate values. It is also safe to create a reference to a value that is uninitialized. The initialization of a value is not encoded into the type system; developers must manually reason about the state of memory or rely on instrumentation to detect uninitialized reads.

Rust's borrow checker prevents variables from being used until they are initialized, and no primitive types tolerate indeterminate values in safe contexts. However, Rust's standard library provides the `MaybeUninit<T>` struct, which represents an instance of `T` that may or may not be initialized. It is undefined behavior to produce a safe reference to uninitialized memory, so the macros `addr_of!` and `addr_of_mut!` must be used to obtain raw pointers to the interior of an instance of `MaybeUninit<T>` without borrowing against it. Once an instance has been fully initialized, the `unsafe` function `MaybeUninit<T>::assume_init()` will unwrap the outer struct to produce the value inside. It is undefined behavior to call `assume_init()` if `T` is not fully initialized.

2.3.2 Aliasing & Mutability. Neither C nor C++ statically restrict aliasing, but the standards for each language allow implementations to make type-directed assumptions about aliasing. In both languages, it is considered undefined behavior for a pointer to one type to refer to a value of another type if the two types differ, with only few exceptions [ISO/IEC 2017, 2023]. Type differences include qualifiers; for example, if a function takes two parameters of types `int *` and `const int *`, it is undefined behavior for these two pointers to alias. If two pointer types are the same, the compiler cannot make any immediate aliasing assumptions. The C standard defines the `restrict` type qualifier, which indicates to a compiler that two pointer parameters with equal types will never alias [ISO/IEC 2017]. However, `restrict` is not enforced and compilers can ignore it.

In Rust, both variables and references have distinct, connected capabilities for mutation. Variables are immutable by default, only mutable variables can be borrowed mutably, and only mutable references can be used for mutation. It is always undefined behavior to mutate memory through a pointer or reference with immutable provenance unless the memory is covered by an `UnsafeCell<T>`. Both C and C++ invert this pattern. By default, variables are mutable and the capability to mutate memory through a pointer is entirely determined by the mutability of the object being pointed to, regardless of the pointer's type [ISO/IEC 2017, 2023]. If a variable was declared as mutable, then any pointer to it—even a **`const`** pointer—can be cast into a mutable pointer and used to write to it. Note that this is *not* equivalent to the semantics of Rust's `UnsafeCell<T>`, which allows mutation through a shared reference. Both C and C++ allow multiple mutable aliases for the same location, while `UnsafeCell<T>` only allows mutation through multiple *immutable* references.

2.3.3 Provenance. Both C and C++ allow pointers to be converted to and from integers, which must be at least as large as the word size of the current architecture. Otherwise, behavior is "implementation defined" [ISO/IEC 2017, 2023]. Compiler implementations may track the "provenance," or origin of pointers to inform compile-time optimizations [Feather 2004], but neither of the current standards for C or C++ define if or how provenance should be preserved across these conversions [ISO/IEC 2017, 2023]. The Rust project has settled on the notion of pointers having provenance, but the specifics of Rust's provenance model are still being decided [Jung 2024].

Miri is an interpreter for Rust programs that can validate **`unsafe`** code against both the Stacked and Tree Borrows aliasing models [Rust Community 2024c]. It has multiple methods of handling provenance across pointer to integer conversions. Its provenance values contain an identifier for an allocation and a borrow tag. By default, Miri uses a "permissive provenance" model, which matches Memarian et al. [2019]'s "PVNI-wildcard" model. When a pointer is converted into an integer, its tag is added to a set of tags that have been "exposed" by pointer to integer conversion. When integers are converted back into pointers, their allocation ID is reconstructed from their address, but they receive a "wildcard" provenance value. When a memory access occurs under Stacked or Tree Borrows using a wildcard tag, Miri's model eagerly interprets this tag as being equivalent to any tag in the stack or tree that will allow the access to take place. Alternatively, developers can enable "strict provenance", which treats any form of pointer to integer conversion as an error.

2.4 Interoperation

Developers can access functions and static variables from foreign libraries using Rust's foreign function interface. Declarations, or "bindings", for each of these objects are written in **`extern`** blocks, which take an optional qualifier to indicate the application binary interface (ABI) used by the foreign library. By default, **`extern`** is equivalent to **`extern`** `"C"`, but Rust has many additional ABIs for various languages, compilers, and architectures.

Only a subset of Rust's types are guaranteed to be compatible with foreign ABIs. For example, Rust's string type `&str` is not null-terminated, so it is not safe to use as a `*`**`const char`**. Structs and enums must be annotated with the `#[repr(C)]` attribute to ensure that their layout in memory is compatible what C expects. The Rust compiler enforces the use of ABI-compatible types via lints, but they can be manually disabled for individual contexts or entire applications. A future iteration of Rust may provide support for declaring opaque types in **`extern`** declarations [Cann and Community 2017]. Opaque types are used to reference a foreign type in Rust without having to define its structure. The current best practice is to declare a struct `#[repr(C)]` with data field with an empty array type `[0;u8]` and an additional field with an instance of `PhantomData<T>` for some type `T`. The Rust compiler treats a value of type `PhantomData<T>` as if it were an instance of type `T`, but in reality, it has no size and is erased during compilation. The correct choice of `T` depends on the aliasing and thread-safety properties required for the foreign type to be used correctly.

Rust does not validate foreign bindings against their definitions, so manually writing bindings can be error-prone. However, several tools can automatically generate foreign bindings from existing codebases and user-provided specifications. Bindgen [Rust Community 2023a] analyzes C header files and creates equivalent, architecture-specific Rust declarations of their API. Similar tools exist for other languages, such as PyO3 [Rust Community 2024d] for Python and CXX [Rust Community 2024a] for C++. In particular, CXX allows developers to create a "bridge module" in Rust where they can import and export items across the foreign boundary, and it provides Rust encapsulations for key data structures from the C++ standard library. Other tools are targeted at creating bindings for Rust APIs for use in other languages. CBindgen [Rust Community 2023b] does the reverse of Bindgen, generating C APIs for Rust functions. Both Diplomat [Rust Community 2024b] and UniFFI [Mozilla 2024] have somewhat similar roles to has CBindgen, but they require different levels of configuration and each is capable of generating bindings in multiple additional languages.

Several dynamic analysis tools are also compatible with multi-language Rust applications. Each is available as a plugin for Cargo, Rust's build tool. Valgrind [Nethercote and Seward 2007] is capable of flagging spatial and temporal memory errors, as well as incompatibilities with size and alignment. Many of the LLVM Project's sanitizers are also compatible with Rust [Rust Community 2024e]. Miri can call foreign functions from natively compiled shared libraries using `libffi`[Green 2024], but only functions that take integer or floating-point arguments are supported.

3 METHODOLOGY

Our goal was to determine whether the differences between Rust and other languages lead to undefined behavior in practice. First, we analyzed Rust libraries from crates.io to find the subset with test cases that produce LLVM bitcode for C or C++ libraries during their build process; we describe this stage in Section 3.1. We developed MiriLLI to analyze runs of these test cases for undefined behavior. It combines Miri, a Rust interpreter that can detect undefined behavior, with LLI, an LLVM interpreter provided within the LLVM toolchain [LLVM Project 2024b]. MiriLLI uses each interpreter to jointly execute programs defined in both LLVM and Rust's intermediate representations. We describe the architecture of our tool in Section 3.2, and we account for several changes that were necessary to resolve differences in semantics between Rust and LLVM. We also include a formal model of our translation layer, which converts typed values between the representations used by each interpreter. In Section 3.3, we report the run-time characteristics of the test cases that we executed in our final evaluation. We describe our method for deduplicating test outcomes to identify unique instances of undefined behavior and our approach for ethically reporting bugs in open-source projects.

3.1 Sampling

We evaluated our design on all compatible Rust libraries with test cases that call foreign functions. Rust libraries are referred to as "crates," and they can be published for public use at crates.io. We used a snapshot of the crates.io database taken on September 20th, 2023. It contained 125,804 unique crates, of which 96% (121,015) had at least one version that had not been unpublished, or "yanked", from the repository. Of all valid crates, 67% (84,106) compiled without intervention, and of these crates, 36% (44,661) had unit tests and 9% (11,120) produced LLVM bitcode files, leaving 3% (3785) crates with both unit tests and LLVM bitcode available. Our sample only includes crates that statically link to foreign code in their default configuration. This sample does not include crates that default to dynamic linking or statically link in non-default configurations. However, the 3785 crates we identified were more than enough to evaluate our research questions.

We ran all unit tests from this subset of crates to determine which tests called foreign functions. First, we natively compiled each crate to obtain a complete list of unit tests. Then, we executed each test using an unmodified version of MIRI with a timeout of 5 minutes. Of the 88,520 test cases from crates that compiled natively, 53% (47,189) passed, 42% (36,766) failed, 4% (3869) timed out, and 1% (1054) had been manually disabled. Tests can be disabled using the `#[ignore]` attribute or with conditional compilation directives. Of the tests that failed in MIRI, 63% (23,116) terminated due to foreign function calls, making them potentially viable for use with our tool.

We executed each potentially viable test under both Stacked Borrows and Tree Borrows using an initial build of our tool to determine which tests called foreign functions that we could execute. Out of all 23,116 potentially viable test cases, 39% (9130) encountered a foreign function defined in LLVM that we could execute under one or both aliasing models. These tests came from 25% (957) of all crates with both test cases and bitcode. We used this subset of viable test cases to guide the development of our extension and to conduct our final evaluation.

3.2 Implementation

Differences in value representation between Rust and other languages prevent Miri from supporting foreign function calls in its current state. To validate Rust's aliasing models, Miri represents each pointer as a product of its address and additional provenance metadata, which is checked before each memory access. Either instrumentation or interpretation is necessary to track and update provenance, and there is no de facto implementation of this for foreign libraries. Other languages may also use different calling conventions than Rust; these depend on the current architecture and whether the foreign library is represented in an intermediate format or as native code. The `libffi` library currently used by MIRI can reconcile these differences at a native level, but it still requires users to implement a high-level mapping between the types of values supported by each language [Green 2024]. We address each of these problems by reusing MIRI's instrumentation to interpret LLVM bytecode and by implementing a conservative, type-directed method for translating arguments passed across foreign boundaries.

3.2.1 Execution. For our tool to function, CLANG [LLVM Project 2024a] must be set as the default C and C++ compiler and configured to emit LLVM bitcode files. Otherwise, using our tool is identical to using an unmodified version of MIRI. Before interpreting the program, we find all LLVM bitcode files in the root directory and eagerly link them into a single module. Execution proceeds as normal up until the point where MIRI encounters a foreign function call. Normally, MIRI would terminate with an error. Our tool looks for a corresponding function definition in the LLVM module. If a definition is found, MIRI passes each of the function's arguments through a translation layer, which converts them into values that are compatible with the signature of the definition.

MIRI is single-threaded, but it supports concurrency by non-deterministically stepping through multiple simulated "threads" of execution. LLI did not originally support any form of multithreading, but we modified it to be compatible with MIRI's implementation so that MIRI can step through instructions in both MIR and LLVM IR. After arguments are converted to LLVM, the current Rust thread is set to join on a new LLVM thread which executes the function. After the LLVM thread terminates, its return value is passed back through the translation layer and given to the Rust thread, which continues to execute. A similar process is used when an LLVM thread calls a Rust function. LLVM-typed arguments are passed through our translation layer and converted into Rust arguments, and the current LLVM thread is set to join on the Rust thread. We support calling foreign functions from parallel Rust threads, we do not support multithreading within LLVM. Only 12 test cases in our sample executed foreign functions concurrently.

Fig. 4. Syntax for types and values from Rust and LLVM

🟥 Rust 🟦 LLVM

$$m, n \in \mathbb{N} \setminus \{0\} \qquad b \in \textsc{Bytes} \qquad l \in \textsc{Locations} \qquad t \in \textsc{Tags}$$

$$\begin{array}{lcll lcll}
\tau & ::= & \mathsf{int}(n) \mid \mathsf{ptr} \mid \overline{\tau} & \textit{(LLVM types)} & v_b & ::= & \overline{b} \mid \mathsf{Pointer}(\ell, \varrho) & \textit{(Base Values)} \\
\tau & ::= & \mathsf{int}(n) \mid *\tau \mid \tau^p & \textit{(Rust types)} & v & ::= & v_b \mid \langle \overline{v} \rangle & \textit{(LLVM Values)} \\
\tau^p & ::= & \overline{\langle \tau, n \rangle}^m & \textit{(Rust products)} & v & ::= & v_b & \textit{(Rust Values)} \\
\tau & ::= & \tau \mid \tau & \textit{(Types)} & \varrho & ::= & t \mid * \mid \cdot & \textit{(Provenance)}
\end{array}$$

Miri takes over each of LLI's operations for accessing memory, calling external functions, and converting between pointers and integers. We are able to detect aliasing violations without instrumenting LLVM bitcode due to the semantics of Rust's **`retag`** operation. Only references are retagged; raw pointers retain the same tag across function calls. All other mechanisms of Stacked and Tree Borrows are instrumented automatically as part of Miri's mechanisms for accessing memory. By treating each LLVM pointer as equivalent to a raw pointer and replacing each of Miri's operations, we get Miri's error-detecting capabilities for free.

Initialization. Miri tracks which bytes are initialized in each allocation and reports an error when uninitialized bytes are read. Even with `MaybeUninit<T>`, it is impossible to set a single bit within an uninitialized byte without reading the entire byte first, which is undefined behavior. However, we frequently observed uninitialized reads in LLVM due to this pattern. LLVM minimally updates memory in order to preserve `undef` and `poison` values, which represent the results of indeterminate or invalid computation and apply at the bit-level [Lee et al. 2017; McIver 2022]. If a single bit needs to be set, instead of writing an entire byte, LLVM will load the existing, potentially uninitialized byte, set the individual bit, and then write it back.

We treat these uninitialized reads as false positives. To avoid them, we equipped MiriLLI with two "memory modes." In the "Uninit" mode, we allow load operations to read uninitialized bytes. This eliminates this category of false positives, but it prevents us from detecting when LLVM reads uninitialized values from Rust-allocated memory. We addressed this by implementing the "Zeroed" mode, which treats all uninitialized reads as errors but zero-initializes LLVM-allocated stack and heap memory. This ensures that any uninitialized reads reported in foreign code are caused by uninitialized bytes being copied from Rust-allocated memory. Test cases that read uninitialized memory must be run twice (once in each mode) to ensure a complete evaluation, since running in "Zeroed" prevents Miri from detecting when Rust reads uninitialized memory allocated by LLVM. Overall, uninitialized reads occurred in 25% (2328) of all viable tests.

3.2.2 Translation. Rust's MIR uses different calling conventions and represents values differently than the LLVM IR emitted by Clang, so we cannot rely on a function's definition to have the same number or types of parameters as its binding. Our translation layer can handle nearly every ABI difference we observed while still being capable of detecting incorrect bindings. Here, we introduce a formal model of our translation layer. Sections A, B, and C of our Appendix include a complete definition of this model and proofs for each of the theorems we introduce in this section.

Syntax. We model the types and values defined by Rust and LLVM's intermediate representations using the syntax shown in Figure 4. Every type has a size, which we define with the relation $\mathsf{sizeof}(\tau)$ in Section A.7 of our Appendix. Both LLVM and Rust share a set of base types and representations for integer and floating pointer values. We represent each of these using the type $\mathsf{int}(n)$, where n is its size in bytes. An integer value of type $\mathsf{int}(n)$ is a string of n bytes $\overline{b}^n$. Pointers take the form $\mathsf{Pointer}(\ell, \varrho)$, where ℓ is the address of the pointer and ϱ is its provenance. Every address can be converted to a byte string of length n_{ptr}, which is architecture dependent. Provenance values ϱ are either concrete borrow tags t, wildcard tags $*$, or '$\cdot$', which is an empty provenance value. Pointer types are different in each language; LLVM pointers are opaque and written as ptr, while Rust's pointers and references take the form $*\tau$ for some valid τ.

Products have different types and representations in each language. Rust's product types are lists of fields $\langle\tau, n\rangle$, where τ is the type of value contained in the field and n is the number of padding bytes. A product value is a pointer to its first field; subsequent fields can be accessed using the metafunction fields defined in Section A.7 of our Appendix; it offsets the pointer by the sum of the size and padding of preceding fields. LLVM has no notion of padding, so we represent its product types as lists of the form $\overline{\tau}$. Similarly, LLVM product values are tuples $\langle\overline{v}\rangle$. Only vectors are represented as pointers in LLVM; both arrays and structs are lists of adjacent values in memory. Rust products can have individual fields nested at any arbitrary depth, while LLVM types are flattened. For example, the Rust product $\langle\langle\mathsf{int}(1), 0\rangle, 0\rangle$ would be represented in LLVM IR as `{i1}`, which is equivalent to $\mathsf{int}(1)$ in our formalism. Typed values that follow each of these principles are well-formed, which we define as $\vdash v : \tau$ in Section A.8 of our Appendix.

State. Our value conversion judgements rely on operations that modify a store μ and a set of provenance values σ. The store represents the shared memory accessible by each interpreter, and it is a mapping from locations ℓ to tuples of bytes with provenance values. A value v can be written to the store μ at location ℓ using the judgement $\mu \vdash \mathsf{write}(\ell, v) \dashv \mu'$, which produces an updated μ' mapping ℓ and its adjacent locations to the bytes and provenance associated with v. As a prerequisite, we assume that each ℓ is a valid location within μ. The set σ contains borrow tags exposed by integer-to-pointer conversion. The function $\mathsf{expose}(\sigma, \varrho) = \sigma'$ adds ϱ to σ if it is a concrete borrow tag. The judgement for reading values is type-directed: $\mu; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'$. Reading a value never changes the store, but if the bytes of a pointer are read as an integer of type $\mathsf{int}(n_{ptr})$, then σ' is updated to expose its provenance. After writing a typed value to the store, it can be read from the same location:

Lemma 3.1. For all well-formed, typed scalar values $v : \tau$ and all valid heap locations ℓ, we have:

$$\mu \vdash \mathsf{write}(\ell, v) \dashv \mu' \quad \Rightarrow \quad \mu'; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'$$

Proof (Sketch). By inversion. We cannot prove the converse, since each byte of an integer written to the store will contain the empty provenance value, but any series of n bytes can be read as an integer of size n, regardless of their provenance. □

Conversion. Our value conversion judgement takes the form $\mu; \sigma \vdash v : \tau \leftrightsquigarrow v : \tau \dashv \mu'; \sigma'$, which reads: "under the store μ and tag set σ, the typed Rust value $v : \tau$ is interconvertible with the typed LLVM value $\tau : v$, producing the updated store μ' and tag set σ'." This is an adaptation of Scherer et al. [2018]'s model of interoperation, which uses a type compatibility relation $\simeq$ and a heterogeneous value conversion relation $\leftrightarrow$. We reason about typed values at run-time, so $\leftrightsquigarrow$ has the semantics of both of these relations combined.

Figure 5 shows all bidirectional conversion rules. The rule C-Int indicates that integers with the same size are interconvertible, while C-Product requires product types to have the same number of fields, which must also be interconvertible. The remaining conversion rules are unidirectional, with one set for LLVM and another set for Rust. Figure 6 shows the rules for converting Rust values to LLVM values; rules for the other direction are listed in Section A.5 of the Appendix.

Multiple ABI differences prevented us from defining every form of value conversion as a bidirectional rule. Rust's product values are pointers and LLVM's product values are lists, so converting values from Rust to LLVM involves reading from the store, while converting in the other direction

Fig. 5. Bidirectional value conversion rules.

$$\textsc{C-Int}\quad \frac{}{\mu;\sigma \vdash \overline{b}^{n} : \mathsf{int}(n) \leftrightsquigarrow \overline{b}^{n} : \mathsf{int}(n) \dashv \mu;\sigma}$$

$$\textsc{C-Product}\quad \frac{\mathsf{fields}(\mathsf{Pointer}(\ell,\varrho) : \overline{\tau^{p}}^{n}) = \overline{v : \tau^{p}}^{n} \qquad \forall i \in [1,n].\ \mu_{i-1};\sigma_{i-1} \vdash v : \tau^{p} \leftrightsquigarrow v : \tau_i \dashv \mu_i;\sigma_i}{\mu_0,\sigma_0 \vdash \mathsf{Pointer}(\ell,\varrho) : \overline{\tau^{p}}^{n} \leftrightsquigarrow \langle\overline{v}\rangle : \overline{\tau}^{n} \dashv \mu_n;\sigma_n}$$

requires writing to the store. Additionally, we encountered 526 test cases from 70 crates required pointers to be converted to and from integers at foreign boundaries. Converting a pointer into an integer requires exposing its provenance, as defined in the rule C-PointerToInt. However, when we convert an integer into a pointer, we assign it the wildcard provenance value.

We also allow any Rust value represented as a pointer to be converted into an opaque pointer in LLVM, as defined by the Rule C-Product. However, LLVM's product types are always passed by value, so the equivalent rule in the other direction enforces the Rust type to be some $*\tau$, instead of τ. The rules C-FieldToScalar and C-ProductToInt handle LLVM's lack of a distinction between the value of a field and its padding. The rule FieldToScalar indicates that if a Rust field has no padding, then is equivalent to the value it contains. Padded fields are handled by ProductToInt, which allows products with any number of fields to be converted into integers with an equivalent size, which includes padding bytes.

Values are only convertible if they have the same size. This is too strict to handle every valid ABI difference. However, we did not observe any valid differences in size in our set of test cases, and maintaining this requirement was necessary to ensure that we would always detect bindings with integer types that had incorrect widths.

Theorem 3.1. For all well-typed Rust values $v : \tau$ and LLVM values $v : \tau$, if either value is convertible to the other, then $\mathsf{sizeof}(\tau) = \mathsf{sizeof}(\tau)$.

Proof (Sketch). By induction. Integers and pointers retain their width when crossing foreign boundaries. All other types are products of these types. If a field of a Rust product value has padding, then it can only be converted into an integer with a width equal to the size of the value and the padding. Otherwise, the padding is zero, and the size of types match on each side of the boundary by induction. □

Scherer et al. [2018]'s conversion and compatibility relations are functional and deterministic; every typed value in one language is uniquely convertible into a typed value in the other language. This does not hold for our relation; LLVM integers can be converted to pointers or products, and Rust products can be passed by reference as pointers or by value as either integers or tuples. The strongest property we can prove of our relation is that it is semi-functional:

Theorem 3.2 (Conversion is semi-functional)**.** For all well-typed values $v : \tau$ and $v : \tau$:

$$\mu;\sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'\sigma \Rightarrow \mu';\sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu'\sigma'$$

Proof (Sketch). By inversion and application of Lemma 3.1. Writing a value implies that it can be read, and conversions from LLVM write values into Rust memory, so conversions from Rust to LLVM can read the same values back. □

Calling Conventions. If a foreign binding and its definition have the same number of parameters, then passing arguments to a foreign function involves a straightforward lifting of the value conversion judgement to lists of typed values. If the judgement gets "stuck" due to differences in

Fig. 6. Unidirectional value conversion rules from Rust to LLVM. Rules from LLVM to Rust have a similar structure and labelling, with "From" instead of "To".

C-AnyToPointer

$$\frac{}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau \rightsquigarrow \mathsf{Pointer}(\ell,\varrho) : \mathsf{ptr} \dashv \mu;\sigma}$$

C-PointerToInt

$$\frac{\ell \triangleq \overline{b} \qquad \mathsf{expose}(\sigma,\varrho) = \sigma'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau \rightsquigarrow \overline{b} : \mathsf{int}(n_{ptr}) \dashv \mu;\sigma'}$$

C-FieldToScalar

$$\frac{\begin{array}{c}\mu;\sigma \vdash \mathsf{read}(\ell,\tau) = v_b \dashv \sigma'' \\ \mu;\sigma'' \vdash v_b : \tau \rightsquigarrow v_b : \tau \dashv \mu;\sigma'\end{array}}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \langle\tau,0\rangle \rightsquigarrow v : \tau \dashv \mu;\sigma'}$$

C-ProductToInt

$$\frac{\begin{array}{c}\mathsf{sizeof}(\tau^p) = q \\ \mu;\sigma \vdash \mathsf{read}(\ell,\mathsf{int}(q)) = \overline{b} \dashv \sigma'\end{array}}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau^p \rightsquigarrow \overline{b} : \mathsf{int}(q) \dashv \mu;\sigma'}$$

size or breadth, we terminate with an error. However, LLVM supports variadic functions, while Rust does not. Rust can declare foreign bindings with variable arguments, but functions defined in Rust must have a fixed set of arguments. When a function is variadic, we attempt to convert the first n arguments corresponding to the number of fixed parameters of the function. If each of the remaining variable arguments are pointer or integers, we convert them into equivalent types in LLVM, with Rust pointers becoming opaque. We do not support passing products as variable arguments since they do not have a canonical form. We use a similar mechanism for handling arguments from LLVM to Miri's "shim" implementations of system calls. Opaque pointers become pointers to single bytes, unless they point to stack or static memory; in those cases, we make the pointee type an integer equivalent to the size of the allocation.

We observed that 2% (156) of our viable test cases from 15% (140) of all crates expected to receive individual product values through multiple parameters, with one parameter for each field. This only occurred with homogeneous aggregates: product types where every field is identical and lacks padding. To accommodate this pattern, we treat each field in a homogeneous aggregate as a separate argument if the entire aggregate cannot be passed as a single parameter. We only attempt to apply this transformation if the number of fixed parameters is greater than or equal to the number of remaining arguments plus the number of fields in the aggregate. Since bindings may be incorrect, this transformation is not guaranteed to be sound. However, it was rarely necessary in practice, and combined with our additional restriction on size and breadth, we feel that it is a reasonable compromise to achieve higher test coverage. We describe each of these transformations using pseudocode in Section C of our Appendix.

3.3 Evaluation

We used MiriLLI to execute each of the 9130 viable test cases that we identified in Section 3.1. We used a consistent configuration throughout this process. We overrode each crate to use version 1.74 (nightly-2023-09-25) of the Rust toolchain with the latest compatible version of Miri (1a82975). We set the global C and C++ compilers to version 16 of Clang, and we configured it to disable optimizations, enable debugging information, and save all temporary bitcode files. We collected data using Amazon EC2 on-demand instances provisioned through CloudBank [Norman et al. 2021]. We used `c6a.2xlarge` instances during the initial stages of data collection, and then we switched to using `c6a.xlarge` instances when using MiriLLI. Miri is single-threaded, so fewer cores were necessary to complete this task efficiently. All commands were executed on each instance in a Docker container running Ubuntu 23.04.

We executed each viable test natively and in MiriLLI under both Stacked Borrows and Tree Borrows. When using Tree Borrows, we treat all values of types derived from `Unique`, such as

`Vec<T>`, as having the semantics of a mutable borrow. Without this configuration option, only `Unique` values in the interior of `Box<T>` are treated this way. We disabled isolation to allow executing non-deterministic operations that depend on the current state of the hardware or operating system. We also enabled symbolic alignment checking. By default, Miri will report an error if the address of a pointer is not a multiple of the requested alignment of the type of value being read or written through it. However, the base address of an allocation is not guaranteed to be a multiple of the requested alignment, so it is possible for a misaligned pointer to be "aligned" by chance. Symbolic checking avoids these false negatives by ensuring that the pointer's offset from the base address of the allocation is a multiple of the alignment of the value being read, and that the alignment associated with the allocation is greater than the alignment of the value. We ignored unaligned accesses in LLVM to prioritize detecting Rust-specific errors.

Deduplicating Errors. We reason about bugs in terms of test outcomes, where an outcome includes the results of a single test under both Stacked Borrows and Tree Borrows. It was typical for several tests in a given library to exhibit the same outcome. We deduplicated outcomes based on exit codes, stack traces, and descriptive information from error logs to avoid redundant reports to developers. Prior to deduplication, we modified error logs to remove unnecessary elements, such as references to specific allocation identifiers, addresses, and borrow tags. We also included more or less detail in stack traces depending on the location of an error. For foreign errors, we used the subset of the stack trace up to the Rust boundary to avoid deduplicating errors which appeared to be identical but were caused by mistakes at different callsites. For Rust errors, we only used the line where the error occurred, since we did not observe errors with a root cause in LLVM.

Reporting Errors. Not all errors detectable by Miri are undefined behavior, and not all instances of undefined behavior are readily exploitable. However, we still attempted to follow ethical vulnerability disclosure practices by reporting bugs privately via email before creating public reports. When we detected a bug in a published crate, we attempted to find contact information for its maintainers by examining the `Cargo.toml` file in the root of the crate and GitHub profile of its repository's owner. If we were unsuccessful, we logged an issue publicly. We also logged public issues if we had not received a response after 1 month or if the type of issue did not appear to be exploitable. For example, memory leaks are not considered undefined behavior in Rust, so we did not report them privately. Our private reports and issues included a representative test case, the output from MiriLLI, and a minimal example written entirely in Rust where applicable. When the fix was trivial, we filed a pull request. Tables 3 and 4 of the Appendix include links to our open-source contributions.

4 RESULTS

On average across each memory mode, 61% of tests terminated due to an unsupported operation, 19% passed, 10% timed out, 1% failed, and 9% had a potential bug, which includes both undefined and undesired behaviors. Table 5 of the Appendix includes test outcomes under each configuration. After deduplication, we had 393 errors to investigate. We manually reviewed every error and found 45 bugs in 35 Rust crates. Stacked Borrows violations were fairly common, but most were not violations of Tree Borrows. Multiple factors lead to the number of bugs being significantly smaller than the number of deduplicated test outcomes; our method of deduplication was conservative, we ignored errors from crates that had been unpublished since the start of our investigation, and we only reported Stacked Borrows violations that were also Tree Borrows violations.

Figure 1 shows the number of bugs for each error type. Bugs occurred just as frequently on each side of the foreign boundary; 23 occurred in LLVM, 16 occurred in Rust, and 6 were related to incorrect bindings. Of these bugs, 33 were found in crates with less than 100 average daily

Table 1. Counts of each unique error grouped by category and location. Locations are either in Rust, LLVM, or in Rust bindings for foreign functions. The "Fix" column indicates the location where a change would be needed to fix the bug, while the "Error" column indicates the location where the bug was reported by Miri.

Location		**Category**		
Fix	*Error*	*Ownership*	*Typing*	*Allocation*
Binding	Binding	-	6	-
	LLVM	3	-	-
LLVM	LLVM	3	1	-
	Rust	-	2	-
Rust	LLVM	**15**	-	1
	Rust	1	3	10
	Total:	22	12	11

downloads in the 6 months prior to our snapshot of crates.io. However we discovered 3 Tree Borrows violations in separate crates which each had more than 10,000 average daily downloads, as well as an access out-of-bounds error in `libdecnumber`, which is part of the GNU Compiler Collection (GCC). The most common type of bug that we discovered were Tree Borrows violations. The majority were caused by incorrect encapsulations or argument types in Rust, but the actual aliasing violations occurred in LLVM.

The bugs we discovered fall into three categories. We describe *Ownership* errors in Section 4.1, which include all violations of Tree Borrows and access out-of-bounds errors. *Allocation* errors include both memory leaks and cross-language deallocation, which we describe in Section 4.2. We describe *Typing* errors in Section 4.3, which include incorrect foreign function bindings and values that were invalid for a given type due to errors in alignment or initialization. We use minimized code snippets to illustrate each bug, but we provide additional details in Tables 3 and 4 of the Appendix. At this point, 24 of the bugs we reported have been fixed, and none have been identified as security vulnerabilities.

4.1 Ownership

We discovered 22 Ownership bugs, which include 16 Tree Borrows violations and 6 accesses out-of-bounds. The majority were caused by incorrect arguments or encapsulations in Rust that triggered invalid memory accesses in foreign code.

Comparison with Stacked Borrows. We found 90 tests from 37 crates that had Stacked Borrows violations. However, 66% (59) of these tests did not have a Tree Borrows violation reoccur at the same location. This is primarily since 76% (45) of these tests had invalid range errors, which are no longer considered undefined behavior under Tree Borrows. Table 2 shows these test outcomes after being deduplicated. The count before a "/" is the number of unique errors, while the count after is the number of crates where they occurred. The columns under "Error or Failure Under Tree Borrows" indicate whether representative test cases also had a Tree Borrows violation.

In addition to range violations, Tree Borrows' handling of retags also had a significant effect on our results. Recall that under Stacked Borrows, a `retag` that creates a unique, mutable permission will perform a write access as a side effect using the permission it was derived from, which may invalidate other mutable permissions. However, under Tree Borrows, retags are always read accesses, regardless of the type of permission being created. Accordingly, we observed that a subset of expired permissions errors caused by unique retags were no longer undefined behavior under Tree Borrows. Insufficient permission and invalid framing errors that did not reoccur under Tree Borrows were also caused by writes that were triggered as side-effects of retags.

Table 2. Counts of deduplicated tests with Stacked Borrows errors and their crates (tests / crates) as well as outcomes for the same tests under Tree Borrows.

		Error or Failure Under Tree Borrows			
SB Error Type	**Count**	*None*	*Non-TB*	*TB, Same Loc.*	*TB, Diff. Loc.*
Expired (Unique Retag)	18/12	2/2	1/1	11/9	4/1
Expired (Write)	4/1	-	-	4/1	-
Insufficient	20/13	5/3	-	15/10	-
Invalid Framing	3/2	-	-	1/1	2/1
Out of bounds	45/10	27/8	2/2	-	16/3

Const-Correctness. There were 9 Tree Borrows errors caused by incorrect casts from immutable references to mutable raw pointers. The following snippet is a minimal example of Bug 41, which used an explicit cast:

```rust
pub fn as_ffi(&self) -> *mut T {
    self.data.as_ref() as *const T as *mut T
}
```

The type `T` encapsulated a heap object containing state for a foreign library. The function `as_ffi` was used to access a raw pointer to this object, which would be passed to a foreign function. The expression `data.as_ref()` produces an immutable reference to `data`, which is then cast to a **`const`** pointer and again to a **`mut`** pointer. However, it still retains the Frozen permission derived from the immutable reference, so mutating this structure across the FFI caused an insufficient permission error. This error is equivalent to the example shown in Figure 2b. We found 5 bugs with equivalent errors, which we label as "`&T as *mut T` in Table 4 of the Appendix.

We found instances where a pointer was incorrectly declared as **`const`** in a foreign function binding. In each case, the underlying function definition was correct, but the maintainer had made an error in writing the binding by hand. The mechanism for these errors is the same as in the definition of `as_ffi` above, but the cast occurred implicitly when the pointer crossed the foreign boundary. These bugs are labelled "Incorrect **`const`**" in our Appendix.

Several of these errors can be detected by Clippy using the lint `as_ptr_cast_mut`, but it is still in development, so it is disabled by default [Rust Community 2022]. Clippy may also produce false positives in Rust libraries that require foreign bindings to match their definitions. C and C++ developers do not always declare types with **`const`** qualifiers where applicable [Davidoff 2019], so it may be necessary and sound to cast an immutable reference into a raw pointer if a foreign function call will only use it to read memory.

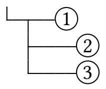
Self-Reference. In Rust, it is not possible to implement a struct where one field contains a reference to another field without using **`unsafe`**; allowing this could lead to use-after-free errors depending on the order in which each field is dropped. It is possible, but non-trivial, to implement this pattern using **`unsafe`** operations [Goregaokar 2022]. Bug 36 demonstrates how a self-referential pattern can be implemented incorrectly. We found this bug in an encapsulation of a C library that implements a minimal file system for embedded applications. Its API used separate structs to represent the state of the file system and its configuration. The configuration held a mutable reference to the state, but both objects were contained in a single parent struct representing the entire file system. Rust held a single mutable reference to the parent struct, which was invalidated when the state was mutated through the configuration during a foreign function call.

We model this with a minimal example in Figure 7. The parameter `x` is declared as a mutable reference to a tuple of type `&mut State` on Line 2. The type `State` is an alias for a tuple of type

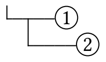
Fig. 7. A minimal example of Bug 35: a Tree Borrows violation caused by an incorrect implementation of a self-referential pattern. The tables on the righthand side corresponds to the subtree of permissions for location of the field "`x.0`" before (to the left of the arrow) and after (to the right of the arrow) the write access on Line 5. The table on top corresponds directly to the snippet on the left, while the table on the bottom describes the tree after we apply our fix, which replaces the first field of `State` with `UnsafeCell<i32>`.

```
1  type State = (i32, *mut i32);
2  fn ub(x: ① &mut State) {
3    x.1 = ② &mut x.0 as *mut _;
4    let r = ③ &mut *x;
5    unsafe{ (*r.1) = 1; }
6    drop(r);
7  }
```

Active		
Reserved	→	Active
Reserved	→	Disabled



Reserved*	→	Active
Reserved*		



(`i32, *mut i32`) representing the self-referential struct; the first field represents the state, and the second field represents the configuration's reference to the state. The second field `x.1` is initialized on Line 3 with a raw pointer cast from a mutable reference to the first field. Since this pointer was originally a mutable reference, it holds the Reserved permission indicated by ②. On Line 4, `x` is reborrowed to create a new mutable reference that is assigned to the variable `r`, which represents the mutable permission held by Rust prior to the foreign function call. Similar to `x.1`, this also receives a reserved permission indicated by ③.

The table at the top of the righthand side of Figure 7 represents the state of the borrow tree before and after Line 5. Multiple Reserved permissions are present at the same depth in the tree, so whichever is used first for a write access will invalidate the other. In this case, `x.1` is written through, causing permission ② to become Active and disabling permission ③. Any further use of `r` is an insufficient permission error. We use `drop` as a placeholder for the function that triggered the error with a function-entry **`retag`**, which attempted to perform a read access using ③.

We fixed this bug by wrapping relevant parts of state in an `UnsafeCell`. In this example, it would be equivalent to changing the first field of `State` to be `UnsafeCell<i32>` and replacing the expression being assigned to `x.1` on Line 3 to `x.0.get()`, which immutably borrows through `UnsafeCell<i32>` to create a raw pointer. The table at the bottom of the righthand side of Figure 7. illustrates how this change affects the tree. References to `UnsafeCell<T>` inherit the permission being borrowed against, so the Active permission ① now applies to both `x` and `x.1`. Mutably borrowing a location covered by `UnsafeCell<T>` creates a special Reserved* permission that can tolerate foreign writes. Consequentially, the permission held by `r` is no longer disabled by the write on Line 7, so the read access on Line 6 is no longer undefined behavior.

The fix we applied for Bug 35 will not work for every self-referential pattern, since `UnsafeCell` only allows shared mutable state through immutable references. The library in question used a mutable borrow to represent unique, *read* access to the parent struct from the Rust side of the boundary, and it was the only active mutable borrow against this location for the duration of the program. Even with the fix that we applied, mutably borrowing from the location referenced by `x` in Figure 5 would invalidate that pointer held by `x.1`, leading to another expired permission error.

Shared Mutability. It was more typical for each side of the boundary to assert mutable access through a single, unique permission. This bug pattern begins when a foreign function receives a raw pointer with a Reserved permission, indicating that it has been cast from a mutable reference. Unbeknownst to the caller, this pointer is copied by the foreign function and stored in memory

Fig. 8. A minimal example of the cyclic aliasing pattern that we observed in Bugs 33 and 36.

```rust
#[repr(C)]
struct Stream {
  state: *mut State,
}
#[repr(C)]
struct State {
  strm: *mut Stream,
}
struct Compression {
  strm: Box<Stream>
}
fn initialize() -> Compression {
  let mut strm = Box::new(zeroed());
  unsafe { ffi:init(strm.as_mut()); }
  Compression { strm }
}
```

```c
void init(Stream* strm) {
  State* s = malloc(sizeof(State);
  s->strm = strm;
  strm->state = s;
}
```

that remains valid after the call returns. Then, the location is mutably borrowed again on the Rust side of the boundary. Similar to Bug 35, there are two adjacent Reserved branches in the tree; one held on each side of the boundary. When one of these permissions is used to write memory, the other is invalidated, leading to an expired permission error when the invalid permission is used.

We observed this type of bug in 4 separate libraries. The most notable examples were Bug 33 from `bzip2` and Bug 36 from `flate2`, which are both popular compression libraries. In particular, `flate2` is actively maintained by the Rust project and had more than 130,000 downloads per day on average during our observation period. Both libraries created a cyclic aliasing pattern between a struct representing the data stream and a struct representing the current state of the compression or decompression algorithm. Figure 8 shows a minimal example of the initialization pattern that lead to this error in each crate. The Rust function `initialize` creates a `Compression` object by allocating its memory through `Box` and passing a mutable reference to the `Box` across the FFI to `init`: a foreign function exposed by the C API. This function initializes the `Compression` object with a pointer to a newly-allocated `State` instance. It also and stores the pointer it received to the allocation of the `Compression` object within the `State` instance, forming a cyclic structure. This makes it unsound to mutably borrow the `Compression` object for the duration of the program, lest it invalidate the reference held by `State`.

The solution for each of these errors was to unwrap the `Box` using `into_raw()` and store a raw pointer to the allocation within the `Compression` struct. We modified the API so that any access to `Compression` would use a raw pointer with the same permission as the pointer held within `State`, ensuring that the cyclic structure remains valid. Since the raw pointer was exposed, we also had to modify the implementation of `Drop` for the parent struct to rewrap this pointer using `Box::into_raw()`, ensuring that it would be deallocated when `Compression` leaves scope. Raw pointers can be copied freely without creating new permissions, so this ensured that each side of the boundary mutated the memory using the same capability. Regrettably, this increased the amount of `unsafe` code on the Rust side, since we could not convert this pointer into `&mut Compression` without invalidating the pointer held by C.

Bug 37 did not have a cyclic structure, but both the cause of the error and our solution were identical. However, Bug 34 required a different approach. It was found in the same file system library as Bug 36, which avoided use of heap allocations as a constraint of the embedded application context. Instead of a `Box`, the Rust encapsulation stored a mutable reference to a stack allocation. The equivalent "unwrapping" operation for a reference to stack memory is the `addr_of_mut!` macro, which will also produce a raw pointer to the root permission of the tree.

Framing. We found one framing violation that was equivalent to the example in Figure 2d. In Bug 26, a Rust function attempted to grow a heap-allocated array of values using a foreign function call. The Rust function received a mutable reference to an allocation of type `&'static mut [T]`, which was converted into a raw pointer to the first element of the array and passed to the foreign function. However, this reference was given a protected, Reserved permission by a function-entry `retag`. Reallocation is equivalent to deallocation under Tree Borrows, so when this pointer was passed to the foreign function and used to reallocate the array, its protected permission was disabled, which is undefined behavior. It was fixed by changing the function to receive a raw pointer, which is not retagged on function entry, instead of a reference.

PhantomData. Rust's `PhantomData<T>` is typically treated as equivalent to an instance of `T`, but this does not currently apply when `T` is equivalent to `UnsafeCell<U>` for some type `U`. Tree Borrows considers any part of a struct to be interior mutable if only one field is interior mutable, but this field must be a concrete instance of `UnsafeCell<T>`; it cannot be `PhantomData<UnsafeCell<T>>` [Jung 2020]. Bug 32 involved an instance where involved an instance where `PhantomData<UnsafeCell<T>>` was incorrectly assumed to provide interior mutability. It was found in a library which providing an API for encapsulating foreign, opaque types in Rust. This included the following definition of the type `Opaque`, which was used to derive references to opaque foreign types.

```
struct Opaque(PhantomData<UnsafeCell<*mut ()>>);
fn as_ptr(&self) -> *mut Self::CType;
```

Since `Opaque` did not contain a concrete instance of `UnsafeCell`, the pointer produced from `&self` in the implementation of `as_ptr` did not have interior mutability, so all writes through it were undefined behavior. The relationship between `PhantomData` and `UnsafeCell` is not yet settled, so it is possible that this will become defined behavior in a future version of Tree Borrows.

Access Out-of-Bounds. We found 6 access out-of-bounds errors in unique crates. Bug 20 was found in two separate Rust encapsulations for libdecnumber, which is part of the GCC toolchain. This library often used the following pattern to iterate over arrays of integers:

```
for (; *current_element==0 && current_element+3<end_of_array;) current_element+=4;
```

The first term in the conditional will execute before the second term, but if the second term is false, then the first term is an access out-of-bounds. This was fixed by swapping the order of each term. Bug 25 was found in a disassembler which incorrectly implemented several instructions, leading to an access out-of-bounds into adjacent static arrays. This error was fixed once the correct semantics were implemented.

Neither of these bugs meaningfully involved Rust. However, the remaining 3 errors were caused by Rust encapsulations that did not adequately enforce the preconditions required by foreign function calls. Bug 21 involved the following API, which redirected its arguments to foreign functions based on the value of `key_size`:

```
fn key_expansion(expanded_key: &mut [u8], key: &[u8], size: KeySize)
```

For example, `KeySize::K128` would call a foreign function expecting `expanded_key` and `enc_key` to have 128 bytes. However, this function did not validate whether the length of each key matched the provided `KeySize`. This made it possible to cause an access out-of-bounds error by passing a `KeySize` that was larger than the length of the keys.

Bug 24 involved a foreign function that expected a pointer to a string with 6 characters. This was allocated with `CString::new(Vec::with_capacity(6))`, exposed as a raw pointer using `into_raw()`, and correctly deallocated after the call using `from_raw(..)`. The implementation of `CString::new` appends a null-terminator the the vector and then converts it into a value of type `Box<[u8]>` using `into_boxed_slice()` which shortens the capacity of the vector to be equal to its length. While its capacity is 6, its length is 1. The encapsulation did not validate that the length of the `CString` instance was correct, leading to an access out-of-bounds when the C codebase attempted to read beyond the first character of the string.

4.2 Allocation

Issues related to memory leaks or incorrect deallocation were the second most common type of bug that we encountered. We found 10 memory leaks and 1 new cross-language deallocation bug.

Memory Leaks. Several Rust types that encapsulate heap allocations, such as `Box<U>` and `CString`, have similar APIs. An instance of one of these types is produced with `T::new(..)`, which creates an allocation that is deallocated by `T::drop()` when the value leaves scope. A raw pointer to the heap allocation can be obtained using `T::into_raw(..)`, which consumes an instance of `T` to produce a raw pointer. To avoid a memory leak, this pointer must be reconstituted in its wrapper type using `T::from_raw(..)`. There were 4 leaks caused by calling `into_raw(..)` on either `Box<T>` and `CString` without later calling `from_raw(..)`. In each case, the type was used to allocate memory for a foreign function call. These bugs were easily fixed by adding calls to `T::from_raw` in the appropriate locations. We also encountered 6 bugs involving leaks of memory allocated by C. This was typically due to neglecting to call a destructor function which had already been exposed by the C API and bound in Rust. However, in Bug 11, the C API did not expose a destructor; it had been designed with the assumption that consumers would be able to call `free`.

Cross-Language Deallocation. We found one example of invalid cross-language deallocation. In Bug 4, a pointer to a heap-allocated string was returned to Rust by a foreign function call and stored as a `Cow<&'static [u8]>`, which will lazily clone its data when mutated. This wrapper type is an enumeration with two variants; `Cow::Borrowed` receives a reference to a value, while `Cow::Owned` takes ownership of a value, deallocating it when it goes out of scope. When this `Cow` was dropped, it deallocated the C heap memory using Rust's allocator. There was not an immediate fix for this bug, since the C API did not expose a destructor.

4.3 Typing

We encountered 11 bugs from 5 crates involving values which were invalid for a given type due to partial initialization or invalid alignment. Most of these bugs were caused by invalid use of `MaybeUninit<T>`, but several involved incorrect foreign function bindings.

Incomplete Initialization. There were 4 bugs involving partial initialization of `MaybeUninit<T>`. Bugs 43 and 44 were each found in the same library. In each case, an uninitialized instance of a struct `T` containing an array of values was created using `MaybeUninit<T>` and passed across the FFI to be initialized. We provide a minimal example of this bug in Figure 9. The foreign call only initialized the first element of the array to 0. The remaining uninitialized elements would never be read by C, since every iteration stopped at the null terminator. However, `MaybeUninit<T>` requires every byte of `T` to be initialized, or else calling `assume_init()` is instantaneous undefined behavior. Miri reported these initialization patterns as an error, and we fixed each bug by zero-initializing the entire array in Rust.

Bug 45 was found in a library that used a similar pattern for initializing foreign structs. Each struct had several padding fields which were never initialized on either side of the boundary. Unlike Bugs 43 and 44, zero-initializing the padding fields in Rust prior to the foreign call did not fix the issue. After examining the LLVM bytecode for the initialization function, we found that all writes to the padding fields had been removed, even with optimizations disabled.

Fig. 9. A minimal example of Bugs 43 and 44.

```
let mut d = MaybeUninit::<[u32;2]>::uninit();
let d = unsafe {
  ffi::init(d.as_mut_ptr() as *mut u32);
  d.assume_init()
};
```

```
void init(uint32_t * x) {
  *data = 0
}
```

Fig. 10. A minimal example of Bug 45. The function `reset` is implemented in C (left) and in LLVM IR (right).

```
1  i32 reset(state_t* s) {
2    memset(s, 0, sizeof(*s));
3    return 0;
4  }
```

```
1  define i32 @reset(ptr %0) {
2    %1 = alloca %struct.state_t
3    call void @memset(ptr %1, i8 0, i64 88)
4    call void @memcpy(ptr %0, ptr %1, i64 80)
5    ret i32 0
6  }
```

Figure 10 shows minimal implementations of the initialization function `reset` in C and LLVM IR. An instance of the struct `state_t` is passed by reference as the first argument and zero-initialized with `memset`. The width of `state_t` is 88 bytes, but the last 8 bytes correspond to its padding fields. On the lefthand side, the C implementation appears to fully initialize its argument using `memset` on Line 2. When lowered to LLVM, the call to `memset` on Line 3 still fully initializes all 88 bytes of `state_t` stored in register `%1`. However, only 80 bytes are copied out of this struct on Line 4; the padding fields are left uninitialized. If `%0` contains a pointer to an instance of `MaybeUninit<state_t>`, then calling `assume_init()` on this instance after `reset` returns is undefined behavior. We fixed Bug 45 by zero-initializing the padding fields after `reset` returned and before calling `assume_init()`.

Incorrect FFI Bindings. We encountered 6 crates with one or more incorrect foreign function bindings. All bindings from each crate had been written manually. Three of these crates had bindings with missing return types. In Bug 6, a binding was declared in a unit test without its 32-bit integer return type. The C implementation used this to return a status code indicating if one of its integer parameters was within bounds. The remaining two bugs involved bindings with incorrect integer types, which lead to incomplete initialization. In Bug 7, a function was declared to return a 32-bit integer, but its C implementation returned a 1-byte boolean value. In Bug 8, the last parameter of a function was declared as a 32-bit integer, but the C implementation expected a `size_t`. The width of `size_t` is architecture dependent, so this binding was only correct for 32-bit architectures.

5 DISCUSSION

Our findings indicate that it is necessary to be aware of the aliasing patterns of foreign libraries to correctly encapsulate them in Rust. Errors that we identified can be prevented through adequate testing, linting, and auditing. However, it is likely that many will persist, undetected, until the Rust community develops a production-ready method for detecting undefined behavior across foreign function boundaries. We provide the following recommendations:

For Rust Developers. Awareness is key to avoiding improper use of `unsafe` features. Developers who depend on foreign code either directly or indirectly should validate their tests cases with language-agnostic bug-finding tools such as LLVM's sanitizers or Valgrind, since several of the bugs we identified involving memory leaks (e.g Bug 11) and accesses out-of-bounds (e.g. Bug 25) can also be detected using those tools. Developers who have written foreign function bindings should consider that the infrequent cost of generating and committing bindings may be preferable to writing them by hand without assistance, which evidently is error-prone. When calling foreign functions, developers must be aware of where each object was created, whether it has one or many active references, whether these references are on each side of the foreign boundary, and which capabilities they must have. Each of these attributes can dramatically influence the correct design for an encapsulation. For example Bug 35 was fixable with an `UnsafeCell<T>` since write accesses only occurred in C, while Bugs 36 and 33 required a different approach, since each side of the boundary needed write access.

For The Rust Project. The Rust community is in dire need of a production-ready solution for validating Rust's aliasing model across foreign function boundaries. Though our approach was capable of finding useful results, it is not likely to scale due to the requirement of implementing shims for system calls and the lack of a formal specification of the ABIs implemented by Rust and Clang. Instrumenting a shared, intermediate format is likely to be the most effective approach. The Rust community should invest resources into completing a prototype implementation of Krabcake [Klock and Garza 2023] to evaluate on real-world applications. However, Valgrind [Seward and Nethercote 2005] can incur significant runtime overhead, so it may also be worthwhile to develop an equivalent approach using LLVM's Sanitizer API. Targeted static analysis will also help developers avoid aliasing violations at foreign boundaries. Binding generation tools would benefit from static analysis to provide recommendations on how to create correct encapsulations. Functions that initialize memory could be audited to ensure correct use of `MaybeUninit<T>`, and warnings could be issued for functions that retain copies of the pointers that they receive.

6 RELATED WORK

Foreign function calls are a common use case for **`unsafe`** code, but they have been overlooked in empirical studies and tool-building efforts due to the complexity of interoperation.

Surveys of the Rust Ecosystem. Studies that examined **`unsafe`** code in Rust libraries have consistently found that foreign function calls are a significant use-case for **`unsafe`** code [Astrauskas et al. 2020; Evans et al. 2020]. In particular, Evans et al. [2020] surveyed all published crates in September, 2018 and found that 22.5% of all **`unsafe`** function calls were to foreign functions. Rust developers also view foreign function calls as a central use case for **`unsafe`** code. Fulton et al. [2021] interviewed 16 Rust developers and surveyed 178 developers, finding that nearly half of participants interoperated with C, C++ and Python. C was seen as easier to interoperate with than C++, which participants found had complex aliasing patterns. Höltervennhoff et al. [2023] focused specifically on Rust developers who use **`unsafe`** code and found that all but one of 26 interview participants had used the FFI. Four participants found that it was typically easy to encapsulate foreign calls, but two noted that memory management was difficult across foreign boundaries. McCormack et al. [2024] conducted a mixed-method study of developers who engaged with **`unsafe`** code. Most of their interview participants struggled with differences between Rust and foreign memory models, and few survey participants avoided creating references to foreign memory.

Types of Bugs & Undefined Behavior. Prior studies have examined bug reports, CVEs, and comments to categorize the errors that occur when **`unsafe`** code is used incorrectly. Most considered foreign function calls to be out-of-scope, and none described aliasing violations in terms of Rust's Stacked Borrows [Jung et al. 2019] or Tree Borrows [Villani et al. 2023] models. Both Qin et al. [2020] and [Xu et al. 2021] examined bug and vulnerability reports from prominent, open source projects and found examples of allocation and typing errors that are similar to the ones that we identified. We mirror Qin et al. [2020]'s approach to categorizing bugs based on their "Cause" and "Effect" with our definitions of "Fix" and "Error." Xu et al. [2021] did encounter errors related to FFI use, but they were primarily "straightforward" issues related to either application-specific invariants, layout, and alignment. Cui et al. [2023b] evaluated a taxonomy of 19 safety properties required by `unsafe` design patterns, but they made it an explicit goal to "disregard" the the FFI to "avoid unnecessary complexities."

Rust-specific Bug-finding Tools. Abstract interpretation [Cousot and Cousot 1977] has proven effective at detecting a wide variety of memory and thread-safety issues related to improper use of `unsafe` code, such as panics, ownership violations, invalid frees, and unsound use of generic types [Bae et al. 2021; Cui et al. 2023a; Li et al. 2021; Qin et al. 2020]. However, each of these approaches targeted Rust's MIR. Several Rust bug-finding tools have supported multi-language applications by translating each language into a shared intermediate format. Li et al. [2022] applied dataflow analysis on LLVM IR and detected several memory leaks and cross-language deallocation errors in Rust libraries. Hu et al. [2022] modified several existing Rust analysis tools, including Miri, to analyze programs defined in a custom intermediate representation based on Rust's MIR. However, they did not evaluate their approach on any real Rust libraries, and aliasing violations were not included in their evaluation. [Xia et al. 2023] took a similar approach using WebAssembly as a shared, intermediate language; their evaluation did not use Miri. The Krabcake project [Klock and Garza 2023] is developing an extension to Valgrind [Seward and Nethercote 2005] that will add support for validating natively-compiled applications against Rust's aliasing models. However, it is not yet usable, and considerable differential testing [McKeeman 1998] will be necessary to ensure that it produces equivalent results to Miri.

Verification. Few approaches to formally verifying Rust have support for any `unsafe` features. Deductive verifiers [Astrauskas et al. 2019; Denis et al. 2022] and refinement type checkers [Lehmann et al. 2023] have targeted its Rust's safe subset, leveraging the guarantees of the borrow checker to reduce annotation burden. Additionally, Yanovski et al. [2021] and Lattuada et al. [2023] have used linear ghost types to support validating a subset of `unsafe` patterns, such as raw heap allocations and `UnsafeCell<T>`. Recent approaches based on the RustBelt [Jung et al. 2017] model of Rust's operational semantics have used compositional verification [Élie Ayoun et al. 2024] and automated proof search [Gäher et al. 2023] to directly support `unsafe` features. However, neither can validate `unsafe` code against Tree Borrows.

Formal Semantics. Matthews and Findler [2007] first explored static soundness for multi-language applications and introduced the concept of "embedding" values from one language at a different type in another language using boundary forms. Scherer et al. [2018] use a modified form of this approach to create a sound composition of ML and a linear language. Patterson et al. [2022]'s approach models modern compilation strategies by relying on the presence of an intermediate representation, analogous to LLVM IR, that is shared by each language. We considered modeling our formalism after this approach, but we lack an intermediate language between LLVM and Rust, and treating LLVM as both a target language and an intermediate language would be cumbersome.

7 CONCLUSION

We combined Miri, a Rust interpreter, with LLI, an LLVM interpreter, to create MiriLLI: a tool which can jointly execute programs defined in each language to detect undefined behavior across foreign function boundaries. We conducted a large-scale evaluation of our tool on Rust libraries that call foreign functions, and we identified 45 unique bugs from 35 libraries. Bugs were related to ownership, initialization, and typing, but the most common types of bugs were violations of Rust's Tree Borrows aliasing model. These were typically caused by unsound patterns in Rust that triggered errors in foreign code. Developers can take immediate steps to avoid these errors by documenting the aliasing patterns of foreign libraries and auditing their use of reference types at foreign boundaries. To ensure that these errors are easy to detect, the Rust project should invest in new approaches to static and dynamic analysis that can accommodate multi-language applications.

DATA-AVAILABILITY STATEMENT

MiriLLI is available on GitHub[1].

REFERENCES

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. https://doi.org/10.1145/3428204

V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). *Proc. ACM Program. Lang.* 3, OOPSLA, 147:1–147:30. https://doi.org/10.1145/3360573

Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

Andrew Cann and Rust Community. 2017. extern_types - RFC #1861. https://rust-lang.github.io/rfcs/1861-extern-types.html

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Will Crichton. 2020. The Usability of Ownership. *CoRR* abs/2011.06171 (2020). arXiv:2011.06171 https://arxiv.org/abs/2011.06171

Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023a. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (may 2023), 21 pages. https://doi.org/10.1145/3542948

Mohan Cui, Suran Sun, Hui Xu, and Yangfan Zhou. 2023b. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. arXiv:2308.04785 [cs.SE]

Sergey Davidoff. 2019. New lint: '_.as_ptr() as *mut _' is Undefined Behavior. https://github.com/rust-lang/rust-clippy/issues/4774

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 90–105.

Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2024. A hybrid approach to semi-automated Rust verification. arXiv:2403.15122 [cs.PL]

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

Clive D.W. Feather. 2004. DR260 - indeterminate values and identical representations. https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr%5F260.htm

Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Proceedings of the Seventeenth USENIX Conference on Usable Privacy and Security (SOUPS'21)*. USENIX Association, USA, Article 31, 20 pages. https://www.usenix.org/conference/soups2021/presentation/fulton

Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2023. RefinedRust: Towards high-assurance verification of unsafe Rust programs. Rust Verification Workshop. https://people.mpi-sws.org/~gaeher/slides/refinedrust%5Frw23.pdf Accessed: 2023-10-20.

Rakesh Ghiya, Daniel Lavery, and David Sehr. 2001. On the importance of points-to analysis and other memory disambiguation methods for C programs. *SIGPLAN Not.* 36, 5 (may 2001), 47–58. https://doi.org/10.1145/381694.378806

[1]https://github.com/icmccorm/mirilli

Manish Goregaokar. 2022. Not a Yoking Matter (Zero-Copy #1). https://manishearth.github.io/blog/2022/08/03/zero-copy-1-not-a-yoking-matter/

Anthony Green. 2024. libffi. https://sourceware.org/libffi/

Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. "I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2509–2525. https://www.usenix.org/conference/usenixsecurity23/presentation/holtervennhoff

Shuang Hu, Baojian Hua, Lei Xia, and Yang Wang. 2022. CRUST: Towards a Unified Cross-Language Program Analysis Framework for Rust. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 970–981. https://doi.org/10.1109/QRS57517.2022.00101

ISO/IEC. 2017. Information technology – Programming languages – C.

ISO/IEC. 2023. Working Draft, Standard for Programming Language C++.

Ralf Jung. 2020. Stacked Borrows: How precise should UnsafeCell be tracked? https://github.com/rust-lang/unsafe-code-guidelines/issues/236

Ralf Jung. 2021. Undefined Behavior deserves a better reputation. https://blog.sigplan.org/2021/11/18/undefined-behavior-deserves-a-better-reputation/

Ralf Jung. 2024. Rust Has Provenance. https://github.com/rust-lang/rfcs/pull/3559

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Steve Klabnik and Carol Nichols. 2022. The Rust Programming Language. https://doc.rust-lang.org/book/.

Felix S. Klock and Bryan Garza. 2023. Krabcake: A Rust UB Detector. Rust Verification Workshop. https://pnkfx.org/presentations/krabcake-rust-verification-2023-april.pdf Accessed: 2023-10-20.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. https://doi.org/10.1145/3586037

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. *SIGPLAN Not.* 52, 6 (jun 2017), 633–647. https://doi.org/10.1145/3140587.3062343

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (jun 2023), 25 pages. https://doi.org/10.1145/3591283

Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541

Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022 (Lecture Notes in Computer Science)*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). Springer Nature Switzerland, Cham, 680–700. https://doi.org/10.1007/978-3-031-17143-7_33

LLVM Project. 2024a. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

LLVM Project. 2024b. LLVM Documentation. https://llvm.org/docs/index.html

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. *SIGPLAN Not.* 42, 1 (jan 2007), 3–10. https://doi.org/10.1145/1190215.1190220

Ian McCormack, Tomas Dougan, Sam Estep, Hanan Hibshi, Jonathan Aldrich, and Joshua Sunshine. 2024. "Against the Void": An Interview and Survey Study on How Rust Developers Use Unsafe Code. arXiv:2404.02230 [cs.SE]

John McIver. 2022. [RFC] Load Instruction: Uninitialized Memory Semantics. https://discourse.llvm.org/t/rfc-load-instruction-uninitialized-memory-semantics/67481/1

William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (jan 2019), 32 pages. https://doi.org/10.1145/3290380

Mozilla. 2024. The UniFFI user guide. https://mozilla.github.io/uniffi-rs/.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, Katherine Yelick, Eric Van Dusen, and James Mitchell. 2021. CloudBank: Managed Services to Simplify Cloud Access for Computer Science Research and Education. In *Practice and Experience in Advanced Research Computing* (Boston, MA, USA) *(PEARC '21)*. Association for Computing Machinery, New York, NY, USA, Article 45, 4 pages. https://doi.org/10.1145/3437359.3465586

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 609–624. https://doi.org/10.1145/3519939.3523703

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

Rust Community. 2022. Clippy Lints - as_ptr_cast_mut. https://rust-lang.github.io/rust-clippy/master/index.html#/as%5Fptr%5Fcast%5Fmut

Rust Community. 2023a. The ‘bindgen‘ User Guide. https://rust-lang.github.io/rust-bindgen/.

Rust Community. 2023b. cbindgen User Guide. https://github.com/mozilla/cbindgen/blob/master/docs.md.

Rust Community. 2024a. CXX — safe interop between Rust and C++. https://cxx.rs/.

Rust Community. 2024b. The Diplomat Book. https://rust-diplomat.github.io/book/.

Rust Community. 2024c. Miri. https://github.com/rust-lang/miri

Rust Community. 2024d. The PyO3 user guide. https://pyo3.rs/v0.20.2/.

Rust Community. 2024e. The Rust Unstable Book. https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html

Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. Fabous Interoperability for ML and a Linear Language. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 146–162. https://doi.org/10.1007/978-3-319-89366-2_8

Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 2.

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 2 (may 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/.

Neven Villani, Derek Dreyer, and Ralf Jung. 2023. Tree Borrows. https://github.com/Vanille-N/tree-borrows/blob/master/full/main.pdf

Lei Xia, Baojian Hua, and Yang Wang. 2023. ACORN: Towards a Holistic Cross-Language Program Analysis for Rust. https://csslab-ustc.github.io/publications/2023/acorn.pdf

Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 3 (sep 2021), 25 pages. https://doi.org/10.1145/3466642

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP, Article 92 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473597

Victor Yodaiken. 2021. How ISO C became unusable for operating systems development. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 84–90. https://doi.org/10.1145/3477113.3487274

.1 Bug Classification & Reporting

Table 3. Download counts for each crate where a bug was found, sorted by "All-Time". The mean download count per day is aggregated across the 6 months between March 20th and September 20th, 2023.

Crate	Version	Mean ⬇ / Day	⬇ All-Time	Last Updated	Bug IDs
flate2	1.0.27	136,292	82,637,795	2023-08-15	36
foreign-types	0.5.0	88,416	64,223,888	2019-10-13	32
bzip2	0.4.4	23,862	13,447,830	2023-01-05	33
zmq	0.10.0	1,594	1,824,174	2022-11-04	18
lcms2	6.0.0	363	158,000	2023-09-02	39, 40, 9
dec	0.4.8	237	118,377	2022-02-05	38, 20, 43, 44
tectonic_engine_bibtex	0.2.1	32	15,680	2023-06-15	26
special-fun	0.2.0	28	15,176	2019-03-15	7
littlefs2	0.4.0	40	14,288	2023-02-07	34, 35
librsync	0.2.3	3	12,350	2023-03-10	37
libhydrogen	0.4.1	23	12,264	2021-05-16	19
bad64	0.6.0	10	6,071	2021-12-22	25
fluidlite	0.2.1	6	5,768	2021-08-21	16
sgp4-rs	0.4.0	10	5,680	2023-07-19	27
minimap2-sys	0.1.16	15	3,782	2023-09-12	42
minimp3_ex-sys	0.1.1	14	3,192	2020-12-19	15
libcmark-sys	0.1.0	1	2,396	2017-11-30	11
dec-number-sys	0.0.25	8	1,400	2022-11-28	22, 31, 5
xxhrs	2.0.0	3	1,390	2020-09-15	45
tetsy-secp256k1	0.7.0	3	1,202	2021-02-19	28
bchlib	0.2.1	1	739	2019-05-27	10
tree-sitter-svelte	0.10.2	2	718	2022-04-15	13
everrs	0.2.1	2	650	2020-04-12	3
x42ltc-sys	0.0.5	1	648	2020-09-05	14
quickjs_regex	0.2.3	3	637	2021-11-30	4, 23, 23
crypto_pimitives	0.1.1	2	535	2019-11-17	21
klu-rs	0.4.0	2	530	2022-09-15	41
ytnef	0.2.0	6	521	2021-11-14	17
tinyspline-sys	0.2.0	1	441	2020-06-09	12
ms5837	0.2.1	2	377	2022-07-30	30
spritz_cipher	0.1.0	1	324	2019-10-16	6
mseed	0.5.0	4	318	2023-08-24	24
jh-rs	0.1.0	1	206	2021-01-31	1
lsmlite-rs	0.1.0	2	52	2023-07-17	29

[1] 0.1.16+minimap2.2.26

Table 4. Unique bugs detected by our tool, sorted by the error type reported by Miri ("Miri Error Type") and our additional classification ("Annotated Error Type"). A "-" indicates that our classification is the same as reported by Miri. When a commit is listed in the last column, it indicates that the bug has been fixed. If multiple commits were used to fix a bug, we provide the last commit in the series.

ID	Crate	Version	Miri Error Type	Annotated Error Type	Location	Issue(s)	Pull(s)	Commit(s)
1	jh-rs	0.1.0	Alignment	Invalid Transmutation	Rust ⟶ Rust	#1		
2	quickjs_regex	0.2.3	Cross-Language Free	-	Rust ⟶ LLVM	#2		
3	everrs	0.2.1	Incorrect Binding	Incorrect Integer Width	Binding ⟶ Binding	#1		
4	quickjs_regex	0.2.3	Incorrect Binding	Incorrect Integer Width	Binding ⟶ Binding		#1	
5	dec-number-sys	0.0.25	Incorrect Binding	Missing Return Type	Binding ⟶ Binding		#2	
6	spritz_cipher	0.1.0	Incorrect Binding	Missing Return Type	Binding ⟶ Binding		#1	
7	special-fun	0.2.0	Incorrect Binding	Missing Return Type	Binding ⟶ Binding	#14	#13	ded37f8
8	secp256k1	0.28.0	Incorrect Binding	Out of Bounds Access	Binding ⟶ Binding	#669	#670	60a5e36
9	lcms2	6.0.0	Invalid Enum Tag	Logical Error	Rust ⟶ Rust			85218b6
10	bchlib	0.2.1	Memory Leaked	Missing C Destructor	Rust ⟶ Rust	#1		
11	libcmark-sys	0.1.0	Memory Leaked	Missing C Destructor	Rust ⟶ Rust	#3		
12	tinyspline-sys	0.2.0	Memory Leaked	Missing C Destructor	Rust ⟶ Rust	#1		
13	tree-sitter-svelte	0.10.2	Memory Leaked	Missing C Destructor	Rust ⟶ Rust	#46		
14	x42ltc-sys	0.0.5	Memory Leaked	Missing C Destructor	Rust ⟶ Rust	#1	#2	1c594f2
15	minimp3_ex-sys	0.1.1	Memory Leaked	Missing C Destructor	Rust ⟶ Rust		#5	33bea0d
16	fluidlite	0.2.1	Memory Leaked	Missing from Raw	Rust ⟶ Rust	#15		
17	ytnef	0.2.0	Memory Leaked	Missing from Raw	Rust ⟶ Rust	#1		
18	zmq	0.10.0	Memory Leaked	Missing from Raw	Rust ⟶ Rust	#387	#388	
19	libhydrogen	0.4.1	Memory Leaked	Missing from Raw	Rust ⟶ Rust	#11		bddff45
20	dec	0.4.8	Out of Bounds Access	-	LLVM ⟶ LLVM	#76		
21	crypto_pimitives	0.1.1	Out of Bounds Access	-	Rust ⟶ LLVM	#1		
22	dec-number-sys	0.0.25	Out of Bounds Access	-	LLVM ⟶ LLVM	#76		
23	quickjs_regex	0.2.3	Out of Bounds Access	-	Rust ⟶ LLVM		#1	
24	mseed	0.5.0	Out of Bounds Access	-	Rust ⟶ LLVM			0cfede1
25	bad64	0.6.0	Out of Bounds Access	-	LLVM ⟶ LLVM			6dbd961
26	tectonic_engine_bibtex	0.2.1	Tree Borrows	Freeing Through &`mut` `T`	Rust ⟶ LLVM		#1129	c64e524
27	sgp4-rs	0.4.0	Tree Borrows	Incorrect Integer Width	Binding ⟶ LLVM	#29		
28	tetsy-secp256k1	0.7.0	Tree Borrows	Incorrect `const`	Rust ⟶ LLVM	#3		

Continued on next page

Table 4 – continued from previous page

ID	Crate	Version	Miri Error Type	Annotated Error Type	Location	Issue(s)	Pull(s)	Commit(s)
29	lsmlite-rs	0.1.0	Tree Borrows	Incorrect `const`	Binding → LLVM		#5	2e0cf90
30	ms5837	0.2.1	Tree Borrows	Incorrect `const`	Rust → LLVM		#26	7be05c1
31	dec-number-sys	0.0.25	Tree Borrows	Incorrect `const`	Binding → LLVM	#1	#2	4a12cce
32	foreign-types	0.5.0	Tree Borrows	Phantom `UnsafeCell<T>`	Rust → LLVM	#24		
33	bzip2	0.4.4	Tree Borrows	Sharing `&mut T`	Rust → LLVM	#94		
34	littlefs2	0.4.0	Tree Borrows	Sharing `&mut T`	Rust → LLVM		#54	
35	littlefs2	0.4.0	Tree Borrows	Sharing `&mut T`	Rust → LLVM		#54	
36	flate2	1.0.27	Tree Borrows	Sharing `&mut T`	Rust → LLVM	#392	#394	0a584f4
37	librsync	0.2.3	Tree Borrows	`&T as *mut T`	Rust → Rust	#23		
38	dec	0.4.8	Tree Borrows	`&T as *mut T`	Rust → LLVM	#74	#2	ece7d84
39	lcms2	6.0.0	Tree Borrows	`&T as *mut T`	Rust → LLVM		#18	28626ed
40	lcms2	6.0.0	Tree Borrows	`&T as *mut T`	Rust → LLVM			5d3b648
41	klu-rs	0.4.0	Tree Borrows	`&T as *mut T`	Rust → LLVM		#1	c5e89d1
42	minimap2-sys	0.1.16	Uninitialized Memory	Erroneous Failure	Rust → Rust			2ac2a6d
43	dec	0.4.8	Uninitialized Memory	Incomplete Initialization	LLVM → Rust	#76	#77	3545623
44	dec	0.4.8	Uninitialized Memory	Incomplete Initialization	LLVM → LLVM	#76	#77	3545623
45	xxhrs	2.0.0	Uninitialized Memory	Uninitialized Padding	LLVM → Rust		#10	

Table 5. Test results across each of the three evaluation modes. In the “Zeroed” mode, all stack and heap memory from LLVM is zero-initialized. In the “Uninitialized” mode, LLVM is allowed to read uninitialized bytes without throwing an error.

	Zeroed		Uninitialized	
Error Type	*Stacked Borrows*	*Tree Borrows*	*Stacked Borrows*	*Tree Borrows*
Borrowing Violation	2.7% (245)	2% (184)	2.7% (245)	2% (183)
Using Uninitialized Memory	2.2% (197)	2.2% (202)	2.2% (200)	2.2% (205)
Other Error	5.4% (495)	5.5% (501)	5.2% (479)	5.4% (490)
Passed	18.7% (1706)	18.9% (1724)	18.6% (1695)	18.7% (1710)
Timeout	9.7% (890)	10.6% (968)	9.6% (873)	10.4% (953)
Unsupported Operation	61.3% (5597)	60.8% (5551)	61.8% (5638)	61.2% (5589)

[1] 0.1.16+minimap2.2.26

A FORMAL SEMANTICS

🟥 Rust 🟦 LLVM

A.1 Domains

b	$\in \textsc{Bytes}$	*(bytes)*
m, n	$\in \mathbb{N} \cup \{0\}$	*(sizes)*
ℓ	$\in \textsc{Locations} : \mathcal{P}(\textsc{Bytes})$	*(heap locations)*
t	$\in \textsc{Tags}$	*(access tags)*

A.2 Type Syntax

τ	$::= \mathsf{int}(n) \mid \mathsf{ptr} \mid \overline{\tau}$	*(LLVM types)*
τ	$::= \mathsf{int}(n) \mid *\tau \mid \tau^p$	*(Rust types)*
τ^p	$::= \overline{\langle \tau, n \rangle}^m$	*(Rust products)*
τ	$::= \tau \mid \tau$	*(Types)*

A.3 Value Syntax

v_b	$::= \overline{b} \mid \mathsf{Pointer}(\ell, \varrho)$	*(Base Values)*
v	$::= v_b \mid \langle \overline{v} \rangle$	*(LLVM Values)*
v	$::= v_b$	*(Rust Values)*
ϱ	$::= t \mid * \mid \cdot$	*(Provenance)*

A.4 Environments

$\mu \in \textsc{Mem} : \textsc{Loc} \longrightarrow (\textsc{Bytes} \times \textsc{Tag})$	*(memory)*
$\sigma \in \textsc{TagSet} : \mathcal{P}(\textsc{Tag})$	*(exposed tags)*

A.5 Conversion

$$\boxed{\mu; \sigma \vdash v : \tau \leftrightsquigarrow v : \tau \dashv \mu'; \sigma'}$$

"*Under the store μ and tag set σ, Rust values v of type τ and LLVM values v of type τ are interconvertible, producing the updated store μ' and tag set σ'.*"

$$\textsc{C-Int}\quad \frac{}{\mu; \sigma \vdash \overline{b}^n : \mathsf{int}(n) \leftrightsquigarrow \overline{b}^n : \mathsf{int}(n) \dashv \mu; \sigma}$$

$$\textsc{C-Product}\quad \frac{\mathsf{fields}(\mathsf{Pointer}(\ell, \varrho) : \overline{\tau^p}^n) = \overline{v : \tau^p}^n \qquad \forall i \in [1, n].\ \mu_{i-1}; \sigma_{i-1} \vdash v : \tau^p \leftrightsquigarrow v : \tau_i \dashv \mu_i; \sigma_i}{\mu_0, \sigma_0 \vdash \mathsf{Pointer}(\ell, \varrho) : \overline{\tau^p}^n \leftrightsquigarrow \langle \overline{v} \rangle : \overline{\tau}^n \dashv \mu_n; \sigma_n}$$

$$\boxed{\mu;\sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu;\sigma'}$$

"Under the store μ and tag set σ, Rust values v of type τ can be converted to LLVM values v of type τ, producing the updated tag set σ'"

C-AnyToPointer

$$\frac{}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau \rightsquigarrow \mathsf{Pointer}(\ell,\varrho) : \mathsf{ptr} \dashv \mu;\sigma}$$

C-PointerToInt

$$\frac{\ell \triangleq \overline{b} \qquad \mathsf{expose}(\sigma,\varrho) = \sigma'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau \rightsquigarrow \overline{b} : \mathsf{int}(n_{ptr}) \dashv \mu;\sigma'}$$

C-FieldToScalar

$$\frac{\mu;\sigma \vdash \mathsf{read}(\ell,\tau) = v_b \dashv \sigma'' \qquad \mu;\sigma'' \vdash v_b : \tau \rightsquigarrow v_b : \tau \dashv \mu;\sigma'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \langle\tau,0\rangle \rightsquigarrow v : \tau \dashv \mu;\sigma'}$$

C-ProductToInt

$$\frac{\mathsf{sizeof}(\tau^p) = q \qquad \mu;\sigma \vdash \mathsf{read}(\ell,\mathsf{int}(q)) = \overline{b} \dashv \sigma'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau^p \rightsquigarrow \overline{b} : \mathsf{int}(q) \dashv \mu;\sigma'}$$

$$\boxed{\mu;\sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu;\sigma'}$$

"Under the store μ and tag set σ, LLVM values v of type τ can be converted to Rust values v of type τ, producing the updated store μ'"

C-PointerFromPointer

$$\frac{}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau \rightsquigarrow \mathsf{Pointer}(\ell,\varrho) : \mathsf{ptr} \dashv \mu;\sigma}$$

C-PointerFromInt

$$\frac{\ell \triangleq \overline{b}}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,*) : \tau \leftsquigarrow \overline{b} : \mathsf{int}(n_{ptr}) \dashv \mu;\sigma}$$

C-FieldFromScalar

$$\frac{\mu;\sigma \vdash v_b : \tau \leftsquigarrow v_b : \tau \dashv \mu'';\sigma \qquad \mu'' \vdash \mathsf{write}(\ell,v) \dashv \mu'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \langle\tau,0\rangle \leftsquigarrow v_b : \tau \dashv \mu';\sigma}$$

C-ProductFromInt

$$\frac{\mathsf{sizeof}(\tau^p) = q \qquad \mu \vdash \mathsf{write}(\ell,\overline{b}) \dashv \mu'}{\mu;\sigma \vdash \mathsf{Pointer}(\ell,\varrho) : \tau^p \leftsquigarrow \overline{b} : \mathsf{int}(q) \dashv \mu';\sigma}$$

A.6 Store Operations

$\boxed{\mu(\ell) = \langle b, \varrho \rangle}$

"*The store μ maps the location ℓ to the byte b with provenance ϱ.*"

$$\textsc{Store}\quad \frac{\ell \mapsto \langle b, \varrho \rangle \in \mu}{\mu(\ell) = \langle b, \varrho \rangle}$$

$\boxed{\mu(\ell, m) = \overline{\langle b, \varrho \rangle}^{m}}$

"*Reading a value of size m from location ℓ produces a list of m pairs of bytes and provenance values.*"

$$\textsc{Store-Slice}\quad \frac{\mu(\ell), \ldots, \mu(\ell + m - 1) = \overline{\langle b, \varrho \rangle}^{m}}{\mu(\ell, m) = \overline{\langle b, \varrho \rangle}^{m}}$$

$\boxed{\mathsf{expose}(\sigma, \varrho) = \sigma'}$

"*Exposing the tag ϱ produces the updated tag set σ'.*"

$$\textsc{Ex-Tag}\quad \frac{}{\mathsf{expose}(\sigma, t) = \sigma \cup \{t\}} \qquad \textsc{Ex-Null}\quad \frac{}{\mathsf{expose}(\sigma, \cdot) = \sigma} \qquad \textsc{Ex-Wild}\quad \frac{}{\mathsf{expose}(\sigma, *) = \sigma}$$

$\boxed{\mu \vdash \mathsf{write}(\ell, v) \dashv \mu'}$

"*Writing the value v to the store μ at location ℓ produces the updated store μ'.*"

$$\textsc{W-Bytes}\quad \frac{\ell \in \mathsf{dom}(\mu_0) \qquad \forall i \in [0, n-1].\mu_{i+1} = \mu_i[\ell + i \mapsto \langle b_i, \cdot \rangle]}{\mu_0 \vdash \mathsf{write}(\ell, \overline{b}^{n}) \dashv \mu_n}$$

$$\textsc{W-Ptr}\quad \frac{\begin{array}{c}\ell \in \mathsf{dom}(\mu_0) \qquad \ell \triangleq \overline{b}^{n_{ptr}} \\ \forall i \in [0, n_{ptr} - 1].\mu_{i+1} = \mu_i[\ell + i \mapsto \langle b_i, \varrho \rangle]\end{array}}{\mu_0 \vdash \mathsf{write}(\ell, \mathsf{Pointer}(\ell, \varrho)) \dashv \mu_n}$$

$\boxed{\mu; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'}$

"*Reading a rust value v of type τ from the store μ at location ℓ produces the updated tag set σ'.*"

$$\textsc{R-Int}\quad \frac{\begin{array}{c}\mu[\ell, n] = \overline{\langle b, \varrho \rangle}^{n} \\ \forall i \in [1, n].\mathsf{expose}(\sigma_{i-1}, \varrho_i) = \sigma_i\end{array}}{\mu; \sigma_0 \vdash \mathsf{read}(\ell, \mathsf{int}(n)) = \overline{b}^{n} \dashv \sigma_n} \qquad \textsc{R-Ptr}\quad \frac{\begin{array}{c}\mu[\ell, n_{ptr}] = \overline{\langle b, \varrho \rangle}^{n_{ptr}} \\ \ell' \triangleq \overline{b}^{n_{ptr}} \qquad \forall i \in [1, n_{ptr}].\varrho_i = \varrho'\end{array}}{\mu; \sigma \vdash \mathsf{read}(\ell, *\tau) = \mathsf{Pointer}(\ell', \varrho') \dashv \sigma}$$

A.7 Metafunctions

$\boxed{\text{sizeof}(\tau) = n}$

"*The type τ has size n.*"

TS-Int
$$\frac{}{\text{sizeof}(\text{int}(n)) = n}$$

TS-R-Ptr
$$\frac{}{\text{sizeof}(*\tau) = n_{ptr}}$$

TS-L-Ptr
$$\frac{}{\text{sizeof}(\text{ptr}) = n_{ptr}}$$

TS-R-Field
$$\frac{\text{sizeof}(\tau) + m = n}{\text{sizeof}(\langle \tau, m \rangle) = n}$$

TS-R-Prod
$$\frac{\Sigma_{i=1}^{m}(\text{sizeof}(\tau^{p}{}_{i})) = n}{\text{sizeof}(\overline{\tau^{p}}^{m}) = n}$$

TS-L-Prod
$$\frac{\Sigma_{i=1}^{m}(\text{sizeof}(\tau_i)) = n}{\text{sizeof}(\overline{\tau}^{m}) = n}$$

$\boxed{\text{scalar}(\tau)}$

"*The type τ is a scalar.*"

$$\frac{}{\text{scalar}(\text{int}(n))} \qquad \frac{}{\text{scalar}(*\tau)} \qquad \frac{}{\text{scalar}(\text{ptr})}$$

$\boxed{\text{fields}(v : \tau^{p}) = \overline{v : \tau}}$

"*The rust product value $v : \tau^{p}$ can be represented as a list of field values $v : \tau$*"

$$\frac{\forall i \in [1, n].o_i = \Sigma_{j=1}^{i-1}(\text{sizeof}(\tau^{p}{}_{j}))}{\text{fields}(\text{Pointer}(\ell, \varrho) : \overline{\tau^{p}}^{n}) = \overline{\text{Pointer}(\ell + o_i, \varrho) : \tau^{p}{}_{i}}^{n}}$$

$\boxed{\text{homogeneous}(\tau)}$

"*The type τ is a homogeneous aggregate.*"

$$\frac{}{\text{homogeneous}(\tau_b)} \qquad \frac{}{\text{homogeneous}(*\tau)}$$

$$\frac{\exists \tau.\forall \langle \tau', n \rangle \in \tau^{p}.n = 0 \land \tau' = \tau \land \text{homogeneous}(\tau)}{\text{homogeneous}(\tau^{p})}$$

$\boxed{\mathsf{equivalent}(\tau) = \tau'}$

"*The type τ is equivalent to the type τ'*"

$$\frac{}{\mathsf{equivalent}(\tau_b) = \tau_b}$$

$$\frac{}{\mathsf{equivalent}(*\tau) = \mathsf{ptr}} \qquad \frac{}{\mathsf{equivalent}(\mathsf{ptr}) = *\tau}$$

A.8 Well-formedness

$\boxed{\vdash v : \tau}$

"*The typed value $v : \tau$ is well-formed.*"

$$\text{WF-INT}\ \frac{}{\vdash \overline{b}^{n} : \mathsf{int}(n)} \qquad \text{WF-LLVMPTR}\ \frac{}{\vdash \mathsf{Pointer}(\ell, \varrho) : \mathsf{ptr}} \qquad \text{WF-RUSTPTR}\ \frac{}{\vdash \mathsf{Pointer}(\ell, \varrho) : *\tau}$$

$$\text{WF-RUSTPROD}\ \frac{}{\vdash \mathsf{Pointer}(\ell, \varrho) : \tau^{p}} \qquad \text{WF-LLVMPROD}\ \frac{\forall i \in [1, n] \vdash v_i : \tau_i}{\vdash \langle \overline{v}^{n} \rangle : \overline{\tau}^{n}}$$

B THEORY

Lemma B.1 (Canonical Forms)**.** For all values v, if $\vdash v : \tau$, then

(1) If $v \triangleq \overline{b}^{n}$, then $\tau \triangleq \mathsf{int}(n)$.
(2) If $v \triangleq \mathsf{Pointer}(\ell, \varrho)$, then τ is either $*\tau$ or τ^{p} in Rust, or ptr in LLVM.
(3) If $v \triangleq \langle \overline{v}^{n} \rangle$ then τ is an LLVM product type $\overline{\tau}^{n}$

Proof. By inspection of $\vdash v : \tau$. □

Lemma B.2 (Compatible Forms)**.** For all Rust typed values $v : \tau$ and LLVM typed values $v : \tau$, if $\mu; \sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu; \sigma'$, then the form of $v : \tau$ determines the possible forms of $v : \tau$.

(1) If $v : \tau \triangleq \overline{b}^{n} : \mathsf{int}(n)$, then $v : \tau$ is an LLVM integer type taking the same form.
(2) If $v : \tau \triangleq \mathsf{Pointer}(\ell, \varrho) : \tau$, then $v : \tau$ may be an opaque pointer of the form $\mathsf{Pointer}(\ell, \varrho) : \mathsf{ptr}$, an integer of the form $\overline{b}^{n} : \mathsf{int}(n)$, or an LLVM product type $\langle \overline{v} \rangle : \overline{\tau}$.

Similarly, if $\mu; \sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'; \sigma$ then the form of $v : \tau$ determines the possible forms of $v : \tau$.

(1) If $v : \tau \triangleq \overline{b}^{n} : \mathsf{int}(n)$, then $v : \tau$ must be an integer value of the same form, a pointer value of the form $\mathsf{Pointer}(\ell, *) : *\tau$ or a Rust product value $\mathsf{Pointer}(\ell, \varrho) : \tau^{p}$ stored at some valid location ℓ.
(2) If $v : \tau \triangleq \mathsf{Pointer}(\ell, \varrho) : \mathsf{ptr}$, then $v : \tau$ must be a Rust product value $\mathsf{Pointer}(\ell, \varrho) : \tau^{p}$ for some τ^{p} or a Rust pointer value $\mathsf{Pointer}(\ell, \varrho) : *\tau$ for some τ.
(3) If $v : \tau \triangleq \langle \overline{\tau} \rangle : \overline{\tau}$ then $v : \tau$ must be a Rust product value $\mathsf{Pointer}(\ell, \varrho) : \tau^{p}$ for some τ^{p}.

Proof. By Lemma B.1 and inspection of the syntax for the value conversion judgement. □

Lemma 3.1. For all well-formed, typed scalar values $v : \tau$ and all valid heap locations ℓ, we have:

$$\mu \vdash \mathsf{write}(\ell, v) \dashv \mu' \quad \Rightarrow \quad \mu'; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'$$

Proof. By inversion, guided by the structure of $v : \tau$. Since $v : \tau$ is a well-formed, scalar-typed value, we have $\vdash v : \tau$ and $\mathsf{scalar}(\tau)$. It follows that v is either a byte string $\overline{b}$ or a pointer $\mathsf{Pointer}(\ell', \varrho)$ to some location ℓ with some provenance ϱ.

Case 1: $v \triangleq \overline{b}^n$

By Lemma B.1 we have that $\tau \triangleq \mathsf{int}(n)$. By inversion of W-Bytes and for $i \in [0, n-1]$, the store μ' maps each location $\ell + i$ to the tuple $\langle b_{i+1}, \cdot \rangle$. By Store and Store-List, we have that

$$\mu'(\ell), \ldots, \mu'(\ell + n - 1) = \mu'(\ell, n) = \overline{\langle b, \cdot \rangle}^n$$

Exposing the null provenance of each byte leave σ unchanged (Ex-Null). We can now apply R-Int to read the original value $\overline{b}$ back from the store.

Case 2: $v \triangleq \mathsf{Pointer}(\ell, \varrho)$

By Lemma B.1 and since $\mathsf{scalar}(\tau)$, we have that τ is either $*\tau$ of ptr. Each are treated equivalently. We can implicitly convert the location ℓ into the byte string, $\overline{b}^n_{ptr}$, so we proceed as in the first case. However, instead of the null provenance, we have:

$$\mu'(\ell, n_{ptr}) = \overline{\langle b, \varrho \rangle}^{n_{ptr}}$$

Each ϱ_i is equivalent to the provenance ϱ of the pointer value. Now, we can apply R-Ptr to reach our goal by reading the original value $\mathsf{Pointer}(\ell, \varrho)$ back from the store.

□

Theorem 3.1. For all well-typed Rust values $v : \tau$ and LLVM values $v : \tau$, if either value is convertible to the other, then $\mathsf{sizeof}(\tau) = \mathsf{sizeof}(\tau)$.

Proof. By induction on value conversion, guided by Lemmas B.2 and B.1. Integers are interconvertible by C-Int and retain the same size on each side of the boundary. The size of a pointer is the size of its address, which is the same on each side of the boundary. Pointers can only be converted into integers of equal size. Size is preserved for product values by induction on their fields. Fields with non-zero padding are converted into integers with a size equal to the size of their value plus its padding. Fields without padding are treated as equal to the values they contain, so size is preserved by induction. □

Theorem 3.2 (Conversion is semi-functional)**.** For all well-typed values $v : \tau$ and $v : \tau$:

$$\mu; \sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'\sigma \Rightarrow \mu'; \sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu'\sigma'$$

Proof. By induction on value conversion. By Lemma B.1, $v : \tau$ can take the following forms:

Case 1: $v : \tau \triangleq \overline{b} : \tau_b$

By Lemma B.2, $v : \tau$ can take one of the following forms:

Subcase 1: $v : \tau \triangleq \overline{b} : \tau_b$

Both typed values are interconvertible by C-Int.

Subcase 2: $v : \tau \triangleq \mathsf{Pointer}(\ell, *) : \tau$

By inversion of C-PtrFromInt, we have $\ell \triangleq \overline{b}$. We can now apply C-PtrToInt to achieve our goal.

Subcase 3: $v : \tau \triangleq \mathsf{Pointer}(\ell, *) : \langle \tau, 0 \rangle$ and $\mathsf{scalar}(\tau)$

By inversion of C-FieldFromScalar, we have:

$$\mu; \sigma \vdash v_b : \tau \leftsquigarrow v_b : \tau \dashv \mu''; \sigma \qquad \mu'' \vdash \mathsf{write}(\ell, \overline{b}) \dashv \mu' \qquad \mathsf{scalar}(\tau)$$

By the induction hypothesis and Lemma 3.1, we have:

$$\mu' \sigma \vdash \mathsf{read}(\ell, \tau) \dashv \mu'; \sigma'' \qquad \mu'; \sigma'' \vdash \tau \rightsquigarrow v_b \dashv \mu'; \sigma'$$

Now we can apply C-FieldToScalar to reach our goal.

Subcase 4: $v : \tau \triangleq \mathsf{Pointer}(\ell, *) : \tau^p$

By inversion of C-ProdFromInt, we have:

$$\mathsf{sizeof}(\tau^p) = q \qquad \mu \dashv \mathsf{write}(\ell, \overline{b}, \dashv) \mu'$$

By Lemma 3.1, we have:

$$\mu' \sigma \vdash \mathsf{read}(\ell, \tau) \dashv \mu'; \sigma'$$

We can apply C-ProdToInt to reach our goal.

Case 2: $v : \tau \triangleq \mathsf{Pointer}(\ell, \varrho) : \mathsf{ptr}$

By Lemma B.2, $v : \tau$ must take the following forms:

Subcase 1: $v : \tau \triangleq \mathsf{Pointer}(\ell, \varrho) : *\tau$ for some τ

Both typed values are interconvertible by C-PointerFromPointer and C-AnyToPointer.

Subcase 2: $v : \tau \triangleq \mathsf{Pointer}(\ell, \varrho) : \langle \tau, 0 \rangle$ for some τ

Equivalent to Case 1, Subcase 3.

Case 3: $v : \tau \triangleq \langle \overline{v} \rangle : \overline{\tau}$

By Lemma B.2, $v : \tau$ must be equivalent to $\mathsf{Pointer}(\ell, \varrho) : \tau^p$ for some τ^p, which is interconvertible by C-Product and the induction hypothesis.

$\square$

C PARAMETER PASSING

Algorithm 1: Converting a list of LLVM arguments to Rust arguments.

```
// A list of typed values provided by Rust
$R \leftarrow [\overline{v, \tau}^n]$;
// A list of LLVM types.
$L \leftarrow [\overline{\tau}^n]$;
// A calling convention; either 'static' or 'variable'.
$C \leftarrow c$;
// The list of converted arguments
$A \leftarrow []$;
// The initial store and tag set.
$\mathcal{S} \leftarrow \mu; \sigma$;
while $R$ is not empty do
    $v_i : \tau_i \leftarrow \text{next}(R)$;
    if $L$ is not empty then
        $\tau_j \leftarrow \text{next}(L)$;
        if $\text{sizeof}(\tau_i) = \text{sizeof}(\tau_j)$ then
            $\mathcal{S} \leftarrow \mathcal{S}'$ where $\mathcal{S} \vdash v_i : \tau_i \rightsquigarrow v_j : \tau_j \dashv \mathcal{S}'$;
            $A \leftarrow A \mathbin{++} [v_j : \tau_j]$;
            continue
        else
            /* We only expand homogeneous aggregates when converting from Rust; in
               the other direction, we skip directly to an error.                 */
            if $\tau \triangleq \overline{\tau p}^n$ and homogeneous$(\tau)$ then
                if $\text{len}(L) \geq n + \text{len}(R)$ then
                    $R \leftarrow R \mathbin{++} [\text{fields}(v_i : \tau_i)]$;
                    continue
                end
            end
        end
    else
        if $L$ is empty and $C$ = variable and scalar$(\tau_i)$ then
            $L \leftarrow L \mathbin{++} [\text{equivalent}(\tau_i)]$; continue
        end
    end
    error()
end
```