Verifying Rust Standard Library

Kosumi Chan
The University of North Carolina at Chapel Hill
kosumi@cs.unc.edu

Hu Guo
The University of North Carolina at Chapel Hill
huhu8@cs.unc.edu

***Index Terms*—Verus, Rust, Standard Library, Memory Safety, Verification**

I. Abstract

Rust's standard library (std) is a cornerstone of its ecosystem, providing essential features while extensively employing `unsafe` code for performance and low-level control. Despite Rust's compile-time memory safety guarantees, the presence of `unsafe` code in std necessitates rigorous verification to ensure its correctness and prevent vulnerabilities that could be triggered even by safe API usage. This paper explores the application of deductive verification to formally prove the functional correctness of the Rust standard library's implementation. We propose and demonstrate an incremental and modular approach using the Verus verifier. Our methodology involves developing formal specifications based on informal documentation, starting with simple safety properties and progressing to high-level functional correctness, and supplementing proofs where automated verification is insufficient. We present case studies, including the formal verification of the `push` method of the `BinaryHeap` data structure in std, illustrating the effectiveness of our approach. The developed methodology is applicable to other data structures within the standard library. This work highlights how formal specification and automated verification can enhance the public interface and overall reliability of the Rust standard library, moving beyond manual auditing and testing to provide stronger guarantees.

II. Introduction

Rust[20] is a modern programming language with an affine type system that checks memory safety at compile time. Its memory safety guarantees, fearless concurrency, and zero-cost abstractions are compelling for systems programming, where safety and performance are both critical. For code whose safety cannot be statically checked by the compiler, Rust provides the `unsafe` keyword to bypass checks.

Some common idioms include:

1) **Minimize and Isolate**: Keep `unsafe` blocks small and confined to modules or functions that perform low-level operations.
2) **Encapsulate Unsafe in Safe Abstractions**: Hide `unsafe` code behind a public, safe API(Application Programming Interface).
3) **Document Invariants and Assumptions**: Clearly state the assumptions(such as pointer validity or aliasing rules).

The Rust standard library(std) is a prime example of these idioms in action. Internally, it employs many unsafe operations not only for low-level control but also for performance. All public and private APIs of std are well documented. Usually a user only needs to interact with the safe public API of std.

Because std underpins the entire Rust ecosystem, ensuring its safety and correctness is paramount. Manual auditing by library maintainers, dynamic testing and incomplete static analysis may reveal the existence of bugs, but they are insufficient for providing strong guarantees. On the other hand, deductive verification can prove that the implementation refines the specification with completeness.

In the paper, we explore how deductive verification can formally prove the functional correctness of the implementation of the Rust standard library. We take an incremental and modular approach: write formal specifications for a module based on its informal documentation, starting from simple safety properties to high-level functional correctness, and supplement proofs when the automated verifier gets stuck. Specifically, we use the state-of-the-art Rust verifier Verus[16].

A. Contributions

We make the following contributions:

- We propose an incremental and modular approach to specifying and proving properties of the standard library code.
- We formally verify the functional correctness of the `push` method implementation of `BinaryHeap` data structure in std. The methodology we develop applies to other data stuctures as well.
- We explore how formal specification and automated verification can enhance and enrich the public interface of std.
- This project is at https://github.com/KaminariOS/Verustd.

III. State of the Art for Verifying Rust Program

Formal verification of Rust code is an active research area. However, the majority of existing automated verifiers[2, 7] are unable to reason about unsafe code.

Kani[24] is a model checker for Rust code. It translates Rust program into an intermediate representation(IR) for verification and then invokes CBMC(C Bounded Model Checker)[15], which checks the IR for property violations via symbolic execution and SAT solving. Kani emphasizes soundness over completeness. It requires minimal user annotations and enables push-button verification of single-threaded safety properties.

However, it has limited specification expressiveness compared to deductive verification.

RefinedRust[8] aims to produce foundational proofs of Rust code in the Coq proof assistant. It is a semi-automated deductive verifier that removes most of Rustc from the TCB(Trusted Computing Base). Its frontend automatically translates the Rust code into **Radium**, a formalization of the operational semantics of Rust in Coq. Verification of a simplified `Vec` implementation takes 22 minutes CPU time on an Apple M1 Max processor due to blow-up caused by the lowering from surface Rust to MIR by Rustc. Currently, it does not support concurrency.

IV. Background

A. Rust

1) Ownership policy: Rust employs an ownership model enforced by an affine type system, meaning each memory resource must be exclusively owned by exactly one variable at a given time. Ownership can be temporarily shared through references ("borrowing"), subject to strict rules: at any moment, Rust allows at most one mutable reference or multiple immutable references. The ownership policy is checked by the borrow checker in rustc at compile time.

When the ownership policy upholds, Rust ensures spatial and temporal memory safety. For operations that may break the ownership policy but cannot be checked by the compiler, the programmer has to use the `unsafe` keyword to mark such code block and take on the responsibility to ensure that the unsafe code does not violate Rust's safety invariants(unsound). Otherwise unsound code may cause undefined behaviors(UB).

2) The Rust Standard Library: Rust's standard library provides essential foundational features and abstractions required for building Rust applications across different environments. It is structured hierarchically into several layers, each designed for different environments and capabilities:

- **core** The `core` crate is the foundation of Rust. It contains the minimal set of language features and abstractions that work in any environment-even on bare metal without an operating system.
- **alloc** The `alloc` crate builds on `core` by adding dynamic memory allocation support. It provides heap-based data structures such as `Box`, `Vec`, `String`, and reference-counted pointers (`Rc`/`Arc`).
- **std** The `std` builds on `core` and `alloc` to provide OS-dependent features (threads, I/O, networking).

The hierarchical architecture of the Rust standard library, positions it as the foundational pillar of the Rust ecosystem, enabling cross-environment compatibilityfrom resource-constrained embedded systems to feature-rich application development. Due to its role in managing low-level system interactions and stringent performance requirements, std extensively employs unsafe code blocks. It encapsulates unsafe implementations with a safe public interface as much as possible. However, in the past there have been multiple reported Common Vulnerabilities and Exposures (CVEs)[22] that can be triggered by invoking a safe function from std. For unsafe code in std, the documentation describes the safety invariants required and internally runtime assertions are put at certain program points, as shown in Table I.

Metric	Count
Safety conditions in std comments	2475
Runtime assertions	9089
Debug assertions	678
Unwinding conditions in std comments	397

TABLE I
ASSERTION COUNTS AND INFORMAL FUNCTION CONTRACTS IN THE STANDARD LIBRARY

B. Automated Dedutive Verification

Automated deductive verification is a formal method that employs logical inference and an automated theorem prover to mathematically establish the correctness of software, hardware, or protocols against specified properties. Unlike testing, which samples executions, or model checking, which explores finite state spaces, deductive verification employs logical deduction base on Hoare logic to prove system adherence to formal specifications exhaustively. It requires annotating code with preconditions, postconditions, and invariants. An automated deductive verifier generates verification conditions(VC) from the code and annotations automatically and invokes an SMT(satisfiability modulo theories) solver to check the VCs.

C. Verus

Verus[16] is the first deductive Rust verifier that trusts the borrow checker of the Rust compiler and leverages the ownership rules to simplify SMT encoding of verification conditions. Together with other optimizations, Verus generates VCs that are an order or two of magnitude smaller[9] in size than its precursor Linear Dafny[18] for complex systems. Similar reduction in verification time is also observed.

Verus's specification and proof language is also based on Rust itself, so it is easier for a Rust programmer to read and write Verus expressions. Ghost code including specifications and proofs is written alongside executable code but erased after compilation. Ghost code can interleave with executable code, convenient for verifying properties at any program point.

Verus introduces linear ghost types, inspired by the Iris concurrent separation logic[10] and the affine type system of Rust. Verus does not directly support verification of unsafe code, but it has ghost state and invariant types that can be used to reason about interior mutability, memory permissions and concurrency. In scenarios where multiple parties share mutable access to the same resource in a coordinated manner-thus creating potential unsafe conditions, Verus provides the VerusSync framework, which models these situations as a tokenized state machine.

Verus supports a comprehensive subset of Rust's language features and has demonstrated scalability in large-scale, complex systems[25].

1) vstd: Vstd is the "standard library" for Verus. It consists of commonly used specifications, proofs and runtime functionality. Vstd serves as the infrastructure for verification, in the same way that std serves as the infrastructure for execution.

V. Threat Model

We assume that while an adversary is restricted to writing safe code, they can invoke any safe public standard functions with arbitrary arguments. Although Rust's borrow checker already prevents many common memory safety errors (such as double-free and use-after-free), it does not guarantee functional correctness or invariants related to data structure behavior. The domain-specific safety property of a downstream program may depend on the functional correctness of std. Consequently, our verification targets a class of attacks in which incorrect library logic might be leveraged to corrupt data or cause unexpected behavior.

- **Safety Conditions** By default, Verus rules out simple errors including number overflow/underflow, division by zero etc. Unsafe code in std usually has its safety conditions documented alongside the source code. Incorrect usage of unsafe code may lead to UB.
- **Data Structure Invariant Violations** Since a data structure may process an arbitrary number of elements controlled by the attacker, we need to prove that it is impossible to cause a safety or invariant violation by invoking a safe method of the target data structure.
- **Potential Bugs from Concurrency or interior mutability** While we do not cover the formal verification of concurrent or interior mutable code in this paper, Verus does supports them. The Verus source code repository contains verification examples of several simplified concurrent or interior mutable data structures in std. Currently, Verus only supports sequentially consistent memory ordering.

We do not address hardware-level attacks, compiler bugs, or misuse of `unsafe` Rust blocks external to the data structures being verified. These lie outside our verification scope, as our project focuses on proving correctness within safe Rust and a subset of unsafe Rust by modeling them with axioms. However, ensuring strong functional invariants is an essential step in making Rust standard libraries more robust against adversarial misuse.

VI. Approach

A. Specification

To verify data structures, we will introduce a combination of preconditions, postconditions, loop invariants, and ghost variables directly into their Rust implementations. For example, `BinaryHeap::push(value)` must preserve order and maintain a conceptual binary search tree structure to ensure correct behaviors. To capture invariants beyond the native checks of Rust, we rely on ghost variables and lemmas, which allow us to encode properties that do not exist at runtime but guide the verifier in proving functional correctness. In any concurrent scenarios, these same ghost annotations can help formalize atomic invariants or concurrency-related postconditions.

```
    /// Returns a reference to an element or
   subslice, without doing bounds checking.
   /// For a safe alternative see [`get`].
   /// # Safety
   /// Calling this method with an out-of-bounds
    index is *[undefined behavior]*
   /// even if the resulting reference is not
   used.
   pub unsafe fn get_unchecked<I>(&self,
   index: usize) -> &I::Output
   {
       // Raw pointer read
       ...
   }
```

Listing 1. a simplified example of the safety condition of an unsafe public function in std

B. SMT-Based Deductive Verification

We embed proof hints (such as function specifications, loop invariants, and ghost variables) directly into the Rust code. Verus then translates both the implementation and our annotations into verification conditions for an SMT solver. Although the solver automates many steps, developers must provide detailed specifications and invariants to guide the verification. This approach differs from so-called push-button tools (e.g., certain model checkers), as Verus does not attempt exhaustive state-space exploration by default. Instead, it relies on user-supplied proof elements to discharge correctness conditions.

Once the solver confirms that all conditions are satisfied, we gain confidence that the annotated invariants hold under valid executions.

C. Challenges and Difficulties

1) Unsupported features: Verus supports a large subset of Rust language features. However, some commonly used features are still missing. For example, the reasoning of mutable references requires **prophecy variables**[13], which is a work in progress. Most code that involves unsupported features can be rewritten while preserving the original semantics. If rewriting is not feasible, annotating the item with `#[verifier::external_body]` tells the verifier to ignore it. The absence of these features does not stem from any inherent limitations within Verus. Rather, these features have yet to be implemented as Verus is currently under development.

2) std is special: To the Rust compiler, std is a special crate. It contains std-only language features, compiler intrinsics and other unstable library features that Verus is unlikely to ever support. The oddness of std necessitates code rewriting. Normalizing std would be beneficial for integrating formal verification into the development and building process of std.

```
    #[verifier::external_body]
    pub fn get_unchecked<I>(&self, index: usize
    ) -> (res: &I::Output)
    // usize is unsigned integer so implicitly
    index >= 0
    requires index < self@.len(),
    ensures *res == self@[index as _]
    {
        // Raw pointer read
        ...
    }
```

Listing 2. a simplified example of converting an unsafe function into a conditionally safe function. We do not need the `unsafe` keyword in the function signature if the caller is also verified. The `external_body` annotation makes the verifier ignore the function body. The safety condition in the comment is also replaced with a precondition. The `@` operator is a syntactic sugar that returns the abstract representation of the operand.

3) Unsafe code: Verus does not have direct support for unsafe code, although vstd provides useful abstractions for some unsafe code. Unsafe code is unsafe because the correct usage of it needs to satisfy some unchecked safety conditions. The general approach to tackling unsafeness is to model its safety conditions as preconditions, semantics as postconditions and convert unsafe code into conditionally safe code. Listing 1 shows an example of an unsafe function, and its safety condition in the comments, and the list 2 is the formally specified version of it.

4) Incomplete External Specifications for std in vstd: Modules in std often import traits, functions, and types from other modules. Ideally, all modules get verified and expose public specifications for other modules to use. In practice, we limit our verification scope to the current module and make assumptions(external specifications) about other modules.

Vstd provides external specifications for a basic subset of std. For std items not in vstd, we need to come up with our own external specifications. Not all external items can be handled this way. For example, reasoning of types with interior mutability, raw pointer memory access, and concurrency require adding some linear ghost state to the function signature.

VII. Evaluation

A. Case Study I: `raw_vec`

```
/// Matches the conditions here: <https://doc.
    rust-lang.org/stable/std/alloc/struct.Layout.
    html>
pub open spec fn valid_layout(size: usize, align
    : usize) -> bool {
    is_power_2(align as int) && size <= isize::
    MAX as int - (isize::MAX as int % align as
    int)
    // This condition is currently missing in
    vstd but needed for proving pointer arithmics
    . See <https://github.com/verus-lang/verus/
    issues/1570>.
    pub fn push(&mut self, item: T)
    requires old(self).well_formed()
    ensures
    self.well_formed(),
    self@.to_multiset() =~= old(self)@.push(item
    ).to_multiset()
    {...}
```

Listing 4. The specification of the `push` method of `BinaryHeap`.

```
    && size % align == 0
}
```

Listing 3. The formal specification of a valid layout in vstd

Our first verification target is the `raw_vec` module in the `alloc` crate. A `RawVec` is a growable memory buffer, which serves as the basis for other container data structures like `Vec` and `VecDeque`. The problem is the logic in `raw_vec` is trivially simple and its functionality largely depends on the system memory allocator, which is out of our verification scope. One simple safety property that we verify is the absence of numeric overflow/underflow. Another safety property is the validity of the type layout (size and alignment), as specified in the listing 3.

The developer left notes in the documentation and used defensive programming(assertions and comments) to prevent an overflow/underflow. With ghost code, this property is explicitly expressed in the preconditions and machine-checked, so comments or defensive programming are no longer needed.

B. Case Study II: BinaryHeap

Data structures like `BinaryHeap` are ideal targets for formal verification. It has high-level properties and its internal implementation is non-trivial.

Same as `raw_vec`, we begin with proving simple safety properties: absence of numeric overflow and safety conditions in the comments. This step is straightforward and requires little verification expertise. From the perspective of a programmer, Verus extends Rust with a descriptive language for expressing static constraints. Therefore, this layer of verification has the potential to scale to the whole Rust ecosystem. At least library authors can incrementally verify simple properties of their own code with minimum efforts.

Verifying the functional correctness is another level of endeavor. The common steps for verifying any data structures are:

1) Select an abstract model of the target data structure. For example, although `BinaryHeap<T>` is conceptually a binary search tree, its actual runtime representation is a `Vec<T>` that be represented as `Seq<T>`(an abstract data type from vstd).
2) Define a well-formedness specification in terms of the abstract model. For example, in a BinaryHeap, both left and right children are less than or equal to their parent. For all public mutable API like `pub fn x(&mut self,`

`...`), the precondition and postcondition should both include well-formedness.

- There are usually more than one way to define well-formedness: bottom-up, top-down, iterative, recursive etc. Need to find one that is easy for Verification.

3) Add preconditions and postconditions to methods of the target struct to describe the semantics of the target method. For example, see listing 4.
4) Write proofs to establish the logical connection from preconditions to postconditions. The executable code in the function body can be considered as part of the proof. It is also possible to have proofs without executable code: proof functions in Verus. The most challenging part is finding and proving loop invariants.

Following these steps, we formally verify the `push` method of the `BinaryHeap`. Other methods can also be verified similarly. The proof-to-code ratio is about 1:1 II:

Spec	Proof	Exec
131	199	195
131	199	195

TABLE II
LINE COUNT

1) Handling of Unsafe Code: The `BinaryHeap` in std is based on the `Vec` data structures, which are external to our verification. The Verus standard library `vstd` provides external specifications and proofs for `Vec`. In principle, `BinaryHeap` can be implemented completely in safe Rust. However, the actual implementation leverages unsafe raw pointer access to avoid unnecessary memory copy operations. In general, verifying raw pointer operations requires the application of linear ghost permissions like `PointsTo`. In the case of `BinaryHeap`, since all raw pointer code sections are small and well defined, we choose to trust them and model the semantics with assumptions.

C. Case Study III: Specification-Argumented Function Signature

Aside from usual preconditions and postconditions, Verus has additional function signature clauses that specify properties useful for ensuring safety.

1) Exception Safety: When a Rust program encounters an error that cannot be handled, the thread panics and a panic handler is invoked. The panic handler unwinds the execution stack and calls destructors of every object on the stack in order. Ideally, every object on the stack is well-formed and its destructor is ready to run. However, it is possible that when the thread panics an object is in a not-well-formed immediate state(in the middle of a series of unsafe operations) and calling its destructor leads to undefined behavior. Exception safety means the program does not unwind or unwinds safety.

Verus already prevents common source of unwinding such as division by zero. It also provides a function signature clause `no_unwind when {boolean expression in the input arguments}` for specifying unwinding conditions. An unwinding condition works similarly to a postcondition and needs to be proved in the function body. To rule out exception safety violations, Verus requires `no_unwind` on all destructor methods and forbids unwinding when an invariant is "open".

```
// CVE-2020-36317: a panic safety bug in String::retain()
pub fn retain<F>(&mut self, mut f: F)
where F: FnMut(char) -> bool
{
        // A while loop with raw pointer operations and calls the function f
        // f may panic and leave self in an inconsistent state
        ...
}
```

Listing 5. An example of exception safety violation

Currently, std specifies unwinding conditions in doc comments of the function. Table I shows the number of those. Rudra[3], a static analyzer of Rust code, reveals an exception safety violation bug[6] in std, as shown in the list 5. The `retain` method takes a function pointer as an argument and calls it in the middle of unsafe operations. The problem is the signature of the function pointer `f` exposes no information about its runtime properties. Even if we disallow unsafe code and ensure `f` is purely safe, it is not enough. It may panic, never terminate, or overflow the stack. In the above example, if `f` panics, the memory content of `self` may not be a valid UTF8 string, which could lead to a memory safety breach because other string APIs operate under the implicit assumption that the same string is encoded in UTF8.

While static analysis tools like Rudra can find safety bugs in Rust at the ecosystem scale automatically, we argue that std is a safety-critical and high-value verification target so preventing bugs in the development phase is more desirable than finding bugs after release.

There are multiple approaches to this problem. The official fix adds guarding code to ensure that `self` is always well-formed(a valid UTF8 string) at all program points. The well-formedness of `self` can be expressed as a type invariant(valid at any program point) or `LocalInvariant`(valid when not open) in Verus.

Another solution, restrictive but straightforward, is to prove `retain` does not unwind by requiring `f` to be `no_unwind`. This only works if the caller is also verified and may be too restrictive for certain use cases that `f` may unwind. Requiring a function pointer to be `no_unwind` is not yet supported by Verus, although there is no inherent limitation that blocks this features.

```
    fn fib(n: u32) -> u32
        decreases n
    {
        match n {
```

```
            0 => 0,
            1 => 1,
            _ => {
                    let fib_1 = fib(n - 1);
                    let fib_2 = fib(n - 2);
                    // Add this to prevent overflow
                    if fib_1 as u64 + fib_2 as u64 > u32::MAX as u64 {
                        return u32::MAX
                    }
                    let res = fib_1 + fib_2;
                    res
                },
        }
    }
```

Listing 6. decreases example

```
tracked struct Gas(nat);
impl Gas {
    #[verifier::external_body]
    proof fn new(num: nat) -> (tracked res: Self)
        ensures res.0 == num
    {
        unimplemented!()
    }

    #[verifier::external_body]
    proof fn consume_loop(tracked &mut self)
        requires old(self).0 > 0,
            ensures old(self).0 == self.0 + 1
    {}

    #[verifier::external_body]
    proof fn consume_func(tracked &mut self)
        requires old(self).0 > 0,
            ensures old(self).0 == self.0 + 1
    {}
}

#[verifier::loop_isolation(false)]
fn binary_search(v: &Vec<u64>, k: u64, Tracked(gas): Tracked<&mut Gas>) -> (r: usize)
    requires
        forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v[j],
        exists|i: int| 0 <= i < v.len() && k == v[i],
        old(gas).0 >= v.len()
    ensures
        r < v.len(),
        k == v[r as int],
        // A conservative approximation of gas consumption
        old(gas).0 <= gas.0 + v.len()
{

    proof {
        gas.consume_func();
    }
    let mut i1: usize = 0;
    let mut i2: usize = v.len() - 1;

    while i1 != i2
        invariant
            i2 < v.len(),
            exists|i: int| i1 <= i <= i2 && k == v[i],
            // A single line of proof added for proving gas consumption
            gas.0 + v.len() - (i2 - i1) >= old(gas).0,
        decreases i2 - i1,
    {
        proof {
            gas.consume_loop();
        }
        let ix = i1 + (i2 - i1) / 2;
        if v[ix] < k {
            i1 = ix + 1;
        } else {
            i2 = ix;
        }
    }
    i1
}
```

Listing 7. Linear Ghost Type for Tracking Complexity

2) User-defined Linear Ghost State in the Signature: Even if we know that a function does not unwind, we have no idea whether it returns or overflows the stack, which may violate safety properties, especially when the execution environment is not isolated, for example, in the Linux kernel space or on bare metal.

In general, proving termination of arbitrary program automatically, i.e., the halting problem is undecidable. However, if for every loop and recursive function a ranking function is provided, we can prove termination. Verus already has a clause `decreases` for specifying ranking function, as shown in Listing 6. However, it is not flexible enough.

We demonstrate how linear ghost types can be leveraged to argument the function signature in Listing 7. The Verus keyword `tracked` marks the struct as a ghost type that is tracked by the borrow checker. In the function body we establish the relationship between gas consumption and input `i`. The precondition now specifies the approximate complexity of the computation inside the function body. The caller not only knows that the function terminates, but also has a rough idea of how running time grows with input. The `Gas` token serves as an abstract capability of CPU time if it is opaque to the user, and we can force all loops and function calls

```
proof fn address_add_align(addr: usize, size:
    usize, alignment: usize)
    requires
        alignment > 0,
        size % alignment == 0,
        addr % alignment == 0,
    ensures
        (addr + size) % (alignment as int) == 0,
{
    broadcast use lemma_mod_adds;
}
```

Listing 8. Proving pointer addition preserves alignment with modulo arithmetics

to consume gas. In practice, looping from zero to 2^{32} is acceptable, but to 2^{64} equals nontermination. We can prove that the binary search example consumes less than 2^{32} units of gas by introducing an implicit axiom of std: the length of a `Vec` is always less than 2^{32}, as documented in internal comments.

Similarly, we can define a linear ghost type for tracking max stack length. If we know the maximum size of a stack frame and the stack size, we can calculate the max stack depth.

Pure static analysis cannot handle cycles in the call graph or calls to external functions, but with a specification-rich function interface, we can prove certain unanalyzable runtime behaviors of the code. Although writing proofs is nontrivial, large language models(LLM) have the potential for proof auto-completion. Fine-tuning LLMs to aid in formal verification[5, 23] is an emerging research field.

The seL4[14] microkernel guarantees the impossibility of kernel stack overflow by disallowing hidden or dynamic stack allocations, and function calls through pointers or recursion. Rust itself has no native dynamic stack allocation, so its stack frames are fixed-size based solely on compile-time constants. seL4 leverages static analysis to calculate the worst-case size of the call stack.

In seL4, the translation from executable code to proof is manual, but in Verus it is automatic. Verus simulates execution with an abstract machine that contains more abstract metadata(ghost state) than the actual machine does. The verified source code contains more information thanks to the added specifications and proofs. Therefore, it is possible to enhance static analysis with ghost states. For example, static analysis can calculate the upper bound of the call stack size automatically if the range of input size, the related ranking function are formally specified for every recursive function.

Vstd defines a linear ghost type `PointsTo` that keeps track of the abstract metadata of a raw pointer including provenance, layout and memory content and exposes a safe API for reading and writing using raw pointers. We craft a pedagogical example in Listing 8 and Listing 9, which can serve as part of the proof of `Vec` in std.

```
fn write_to_raw_array<V>(first: V, second: V)
    requires
                    core::mem::size_of::<V>() !=
    0,
                    size_limit_for_valid_layout
    ::<V>(2)
{
    // This is an assumption that means
    // we trust the Rust compiler for the layout
    of type V to be valid
    layout_for_type_is_valid::<V>();

    let size = core::mem::size_of::<V>();
    let align = core::mem::align_of::<V>();

    // p is the actual pointer returned by the
    allocator while points_to_raw and dealloc are
     ghost.
    // points_to_raw represents the permission to
     access the memory region pointed to by p
    // It contains abstract metadata about the
    allocation such as provenance, size,
    alignment and memory value
    let (p, Tracked(points_to_raw), Tracked(
    dealloc)) = allocate(
                2 * size,
                align,
            );

    assume(p as usize + size <= usize::MAX);

    let tracked mut pointsToFirst;
    let tracked mut pointsToSecond;
    proof {
        // This should be included in the post
    condition of layout_for_type_is_valid
        assume(size % align == 0);
        let item_range = set_lib::set_int_range(
     p as int + size,
            p as int + 2 * size,
        );
        // We can split and merge memory
    permissions
        let tracked (a, b) = points_to_raw.split
    (item_range);
        pointsToFirst = b;
        pointsToSecond = a;

        address_add_align(p as usize, size, align
    );
    }
    let tracked mut pointsToFirst =
    pointsToFirst.into_typed::<V>((p as usize)
    as usize);
    // Now ptr write is conditionally safe
    ptr_mut_write(p as *mut V, Tracked(&mut
```

```
    pointsToFirst), first);
41  // Provenance is the abstract ID of a
    specific allocation
42  let provenance = expose_provenance(p);
43  let new_p: *mut V = with_exposed_provenance(
    p as usize + size, provenance);
44  let tracked mut pointsToSecond =
    pointsToSecond.into_typed::<V>((new_p as
    usize) as usize);
45  ptr_mut_write(new_p , Tracked(&mut
    pointsToSecond), second);
46 }
```

Listing 9. a pedagogical example showcasing how ghost memory permission converts unsafe raw pointer operation to safe code

All ghost code is erased after compilation, so ghost types have no run-time representation or cost.

VIII. Discussion

A. std-like crates

Although formal verification requires verification expertise and significant manual effort, it does not have to be scalable to the whole ecosystem to be useful. Code base that are safety-critical, complex and error-prone should be prioritized for verification.

Rust std is maintained by Rust experts and well funded so it is possible to integrate formal verification into development. There are other std-like crates like Rust for Linux[1] kernel crate and Rex[12] kernel crate that are suitable for formal verification.

B. Language-based Safety

Even without verification, the Rust compiler can prove memory safety of safe Rust code. Isolation of subsystems based on Rust language safety guarantees[4, 21, 17, 12] is an ongoing research topic.

1) Challenges in ensuring the safety of adversarial safe Rust code: Existing work on isolation with Rust follows a non-adversarial model because the safety of Rust does not account for transient execution attacks and the compiler toolchain contains unsound bugs.

C. Verification-based Safety

It is an open research question how formal verification can strengthen the safety guarantees of Rust. Ideally we will have a formally verified Rust compiler like CompCertC so that compiler bugs are eliminated and we only rely on the hardware model. Or at least we need an assembly verifier that proves the generated assembly code retains the memory safety property of safe Rust source code.

We may also require untrusted code to carry proofs of its safety. Verus has a no-cheating mode that forbids assumptions. With no-cheating mode enabled, the untrusted code must provide proof of its safety to pass verification. Proof-carrying code may be able to squeeze more performance and flexibility out of existing isolation mechanisms and minimize the extralingual runtime required for safety.

Sometimes, the safety of a system depends on some hidden state machine. The code must follow some kind of protocol to be safe. For example, some functions need to be invoked in a specific order, or some MMIO registers of a device must be accessed following the specification in the datasheet, or after modifying page table entries, the TLB cache needs to be flushed. Out of 240 CVEs of device drivers in the Linux kernel, 72 are due to protocol violation[19]. Type-state pattern and session types[11] are proposed for encoding such protocols at zero runtime overhead. They can be seemed as a limited form of ghost state. Verus ghost state is a more general and straightforward approach to keeping track of abstract metadata of the system.

IX. Conclusion

In this paper, we have presented an incremental deductive verification approach to formally demonstrate the functional correctness of key components of the Rust standard library using the Verus verification tool. Starting from the informal documentation of each module, we systematically derive pre-conditions, postconditions, loop invariants, and ghost state annotations to bridge the gap between safe Rust type-level guarantees and the deeper semantic properties required for highassurance systems.

Our case studies show that, while initial annotation and proof development requires expertise, Verus scales to non-trivial data structure implementations with manageable annotation overhead. By eliminating informal comments and runtime assertions in favor of machinechecked contracts, we gain stronger confidence in library invariants and expose previously undocumented safety conditions to both implementers and clients.

By establishing a clear methodology and demonstrating its effectiveness in the core Rust std data structures, this work serves as a precursor for high-assurance Rust libraries that harden the languages safety with the rigor of formal verification.

X. Who Did What

- **Kosumi Chan:** Literature survey, project setup, verification, paper writing.
- **Hu Guo:** Study Verus' use of Rust's affine-type system, practice examples for implementation of the verification, inspected verification strategy for `BinaryHeap`, debugging, paper writing.

References

[1] URL: https://rust-for-linux.com/.

[2] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 2427, 2022, Proceedings*. Pasadena, CA, USA: Springer-Verlag, 2022, pp. 88–108. ISBN: 978-3-031-06772-3. DOI: 10.1007/978-3-031-06773-0_5. URL: https://doi.org/10.1007/978-3-031-06773-0_5.

[3] Yechan Bae et al. “Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 84–99. ISBN: 9781450387095. DOI: 10.1145/3477132.3483570. URL: https://doi.org/10.1145/3477132.3483570.

[4] Kevin Boos et al. “Theseus: an Experiment in Operating System Structure and State Management”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1–19. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/boos.

[5] Tianyu Chen et al. *Automated Proof Generation for Rust Code via Self-Evolution*. 2025. arXiv: 2410.15756 [cs.SE]. URL: https://arxiv.org/abs/2410.15756.

[6] *CVE-2020-36317*. Available from NVD, CVE-ID CVE-2020-36317. Dec. 2013. URL: https://nvd.nist.gov/vuln/detail/cve-2020-36317.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. “Creusot: A Foundry for the Deductive Verification of Rust Programs”. In: *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 2427, 2022, Proceedings*. Madrid, Spain: Springer-Verlag, 2022, pp. 90–105. ISBN: 978-3-031-17243-4. DOI: 10.1007/978-3-031-17244-1_6. URL: https://doi.org/10.1007/978-3-031-17244-1_6.

[8] Lennard Gäher et al. “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656422. URL: https://doi.org/10.1145/3656422.

[9] Travis Hance. “Verifying Concurrent Systems Code”. Available at https://www.andrew.cmu.edu/user/bparno/papers/hance_thesis.pdf. PhD thesis. Pittsburgh, PA: Carnegie Mellon University, Aug. 2024.

[10] Iris contributors. *Iris*. DOI: 10.5281/zenodo.595182. URL: https://github.com/SciTools/iris.

[11] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. “Session types for Rust”. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 13–22. ISBN: 9781450338103. DOI: 10.1145/2808098.2808100. URL: https://doi.org/10.1145/2808098.2808100.

[12] Jinghao Jia et al. *Safe and usable kernel extensions with Rax*. 2025. arXiv: 2502.18832 [cs.OS]. URL: https://arxiv.org/abs/2502.18832.

[13] Ralf Jung et al. “The future is ours: prophecy variables in separation logic”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371113. URL: https://doi.org/10.1145/3371113.

[14] Gerwin Klein et al. “seL4: formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[15] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. *CBMC: The C Bounded Model Checker*. 2023. arXiv: 2302.02384 [cs.SE]. URL: https://arxiv.org/abs/2302.02384.

[16] Andrea Lattuada et al. “Verus: Verifying Rust Programs using Linear Ghost Types”. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586037. URL: https://doi.org/10.1145/3586037.

[17] Amit Levy et al. “Multiprogramming a 64kB Computer Safely and Efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 234–251. ISBN: 9781450350853. DOI: 10.1145/3132747.3132786. URL: https://doi.org/10.1145/3132747.3132786.

[18] Jialin Li et al. “Linear types for large-scale systems verification”. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527313. URL: https://doi.org/10.1145/3527313.

[19] Zhaofeng Li et al. “Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel”. In: *2024 Annual Computer Security Applications Conference (ACSAC)*. 2024, pp. 548–562. DOI: 10.1109/ACSAC63791.2024.00054.

[20] Nicholas D Matsakis and Felix S Klock II. “The rust language”. In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[21] Vikram Narayanan et al. “RedLeaf: Isolation and Communication in a Safe Operating System”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[22] Qwaz. *rust-cve: CVEs for the Rust standard library*. https://github.com/Qwaz/rust-cve. GitHub repository, accessed on March 3, 2025. 2025.

[23] Aleksandr Shefer et al. *Can LLMs Enable Verification in Mainstream Programming?* 2025. arXiv: 2503.14183 [cs.SE]. URL: https://arxiv.org/abs/2503.14183.

[24] Alexa VanHattum et al. “Verifying Dynamic Trait Objects in Rust”. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2022, pp. 321–330. DOI: 10.1145/3510457.3513031.

[25] *Verus — Publications and Projects*. https://verus-lang.github.io/verus/publications-and-projects/. Accessed: 2025-03-04. 2025.