# Verifying Rust's Standard Library with Verus

Presenter: Kusomi, Hu Guo

Github repo: https://github.com/KaminariOS/Verustd

# Motivation & Background

Rust Programing Language

**Rust's safety promise:**

Rust's affine type and ownership system enforces memory safety at compile time

**The "unsafe" escape hatch:**

Rust allows opting out of these checks via unsafe code blocks, trading safety for flexibility

**Unsafe in the standard library, including but not limited to:**

- Raw pointer dereferencing
- Concurrency and Atomics
- Any memory safety requirement that cannot be proved by the compiler

**Risk:**

Bugs in unsafe code can lead to undefined behavior since the usual guarantees don't apply. Manual auditing and testing may miss such subtle bugs.

Rust ownership example

```rust
fn main() {
    // s represent ownership of a memory object
    // Here, s is a string on the heap
    let mut s = String::from("hello"); // s owns the String

    // the ownership can be borrowed temporarily
    borrow_string_mut(&mut s);

    // s is moved into sink, no longer in current scope
    sink(s)
    // println!("{}", s);          // ERROR: use of moved value "s"
}

// Immutable borrow
fn borrow_string(s: &String) {
    // ERROR: cannot mutate
    // s.pop();
}

// Mutable borrow
fn borrow_string_mut(s: &mut String) {
    s.pop();
}

fn sink(s: String) {
    // s goes out of scope and its destructor gets called automatically
    // Memory leak, double free, dangling pointer: impossible
}
```

# Why Verify the Rust Std Library?

**Foundation of Rust programs:**

Virtually all Rust programs depend on std, alloc, or core libraries. Any unsafe implementation in std can affect many downstream programs. Its safety and functional correctness are both important.

The safety of downstream program may depend on the functional correctness of std.

**Significant unsafe footprint:**

Rust's core library contains ~7k unsafe functions, and std has ~7.5k unsafe functions

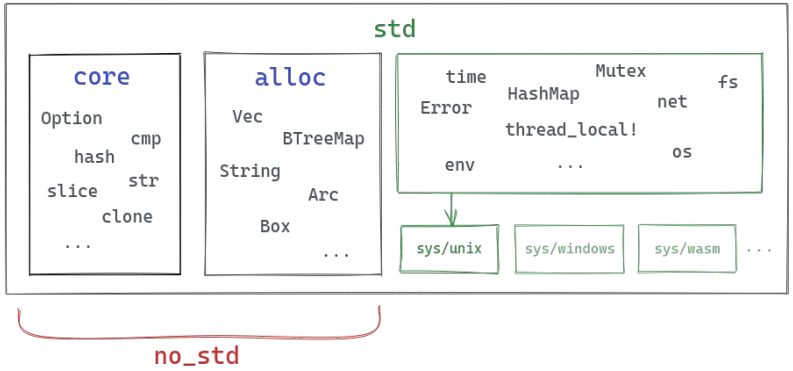
**History of soundness bugs:**

Dozens of memory safety issues have been found in the std library (57 soundness issues in the last 3 years, with 20 leading to CVEs)

**Limitations of testing:**

Traditional tests or fuzzing can miss edge-case bugs

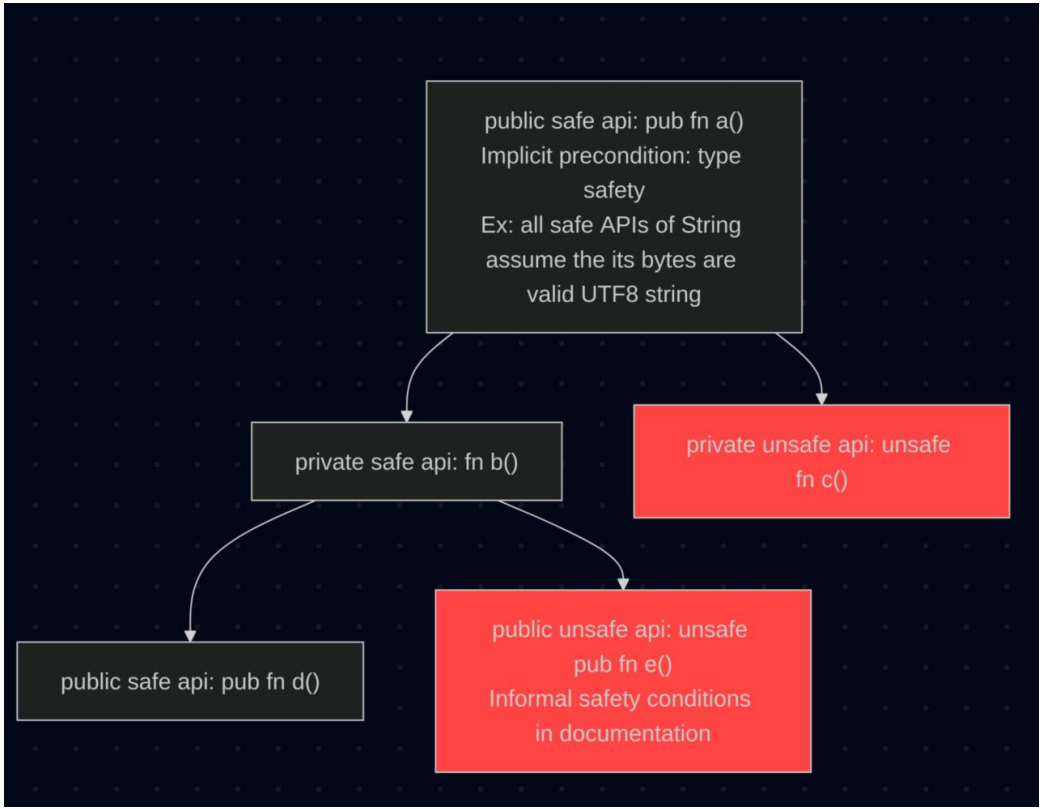
https://blog.timhutt.co.uk/std-embedded-rust/index.html




Unsafe example in std

```
/// # Safety
///
/// The caller must guarantee that `pos < self.len()`.
unsafe fn sift_up(&mut self, start: usize, pos: usize) -> usize
```

# Conceptual call graph of a Rust library function




Loss of logical conditions across function boundaries

| 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

Info loss example in raw_vec

```rust
fn grow_amortized(
    &mut self,
    len: usize,
    additional: usize,
    elem_layout: Layout,
) -> Result<(), TryReserveError> {
    // This is ensured by the calling contexts.
    debug_assert!(additional > 0);
```

# Challenges and Obstacles

- Unsupported features: Verus supports a large subset of Rust language features. However, some commonly used features are still missing.
- 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.

We work around the obstacles above by code re-writing.

- Formalizing functional correctness is non-trivial
- For code outside of our verification scope but used by our verification target, we need to provide external specifications(like stubs) for them.

# Approach Overview: Formalization and Proof

**Deductive verification:**

We write formal specifications (preconditions, postconditions, and invariants) for std library functions and then prove that the implementation meets those specs.

**Formalizing safety conditions and functional correctness**:

We translate the informal safety requirements into formal contracts. For example, if an unsafe function requires the caller to uphold some invariant, we encode that as a requires clause. We add ghost variables and invariants to track the abstract state.

Formalizing functional correctness is more tricky. There are more than one way to model a given data structure and we need to find the one that is proof-friendly.

**Proof**

The most challenging part is finding loop invariants.

Specification of the push method of BinaryHeap

```
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()
```

Verification example in std

```
/// # Safety
///
/// The caller must guarantee that `pos < self.len()`.
unsafe fn sift_up(&mut self, start: usize, pos: usize) -> (res: usize)
    requires pos < old(self).spec_len(),
    start == 0, // all calls to this function have start == 0
    old(self).well_formed_to(pos as _)
    ensures
    self.spec_len() == old(self).spec_len(),
    self.well_formed_to((pos + 1) as _),
    old(self)@.to_multiset() =~= self@.to_multiset()
```

# Verus: the State of the Art Verifier for Rust

**Rust-native proofs:**

Verus allows writing proofs in Rust itself, using Rust's syntax and leveraging its ownership model

**Linear types & permissions:**

Verus introduces linear ghost types to mirror Rust's ownership rules in the logic

**SMT-backed automation:**

The tool translates our Rust code + specs into logical conditions for an SMT solver

**Supports unsafe reasoning**:

Verus can handle certain unsafe patterns.

**Anvil: Verifying Liveness of Cluster Management Controllers**

Authors:
Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, and Zicheng Ma, *University of Illinois Urbana-Champaign;* Tej Chajed, *University of Wisconsin-Madison;* Jon Howell, Andrea Lattuada, and Oded Padon, *VMware Research;* Lalith Suresh, *Feldera;* Adriana Szekeres, *VMware Research;* Tianyin Xu, *University of Illinois Urbana-Champaign*

***Awarded Best Paper!***

**VeriSMo: A Verified Security Module for Confidential VMs**

Authors:
Ziqiao Zhou, *Microsoft Research;* Anjali, *University of Wisconsin-Madison;* Weiteng Chen, *Microsoft Research;* Sishuai Gong, *Purdue University;* Chris Hawblitzel and Weidong Cui, *Microsoft Research*

***Awarded Best Paper!***

Two best papers of OSDI24
both uses Verus!

# Verus Workflow

**Modular verification:**

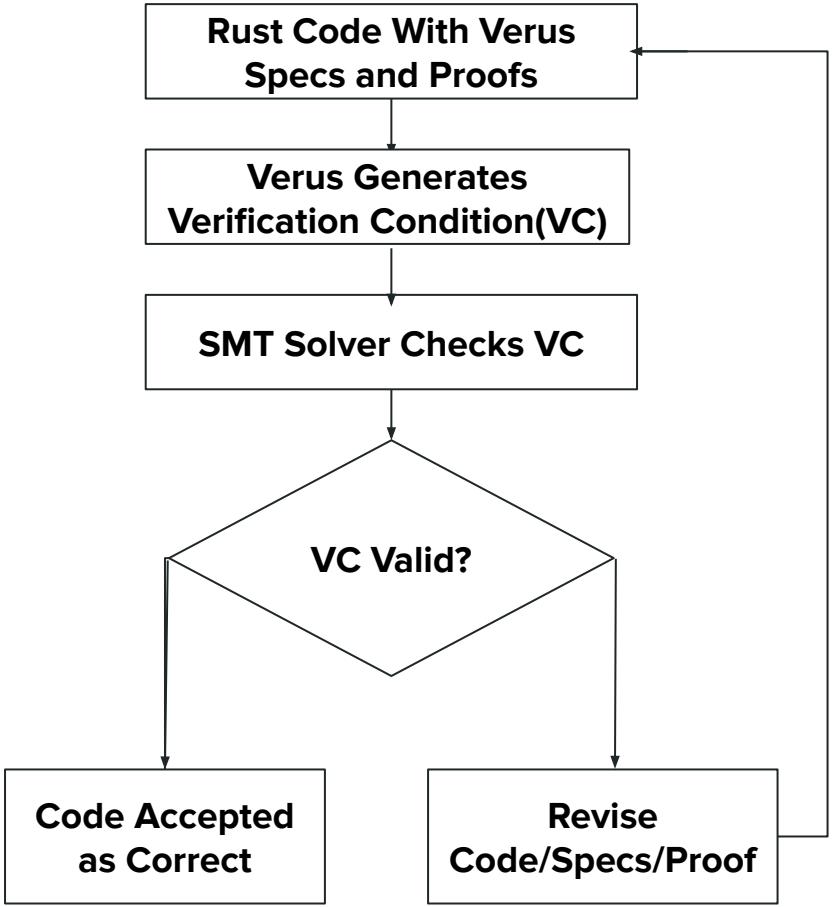
We verify pieces of the library in isolation. By proving simple functions correct, we can reuse those results when verifying higher-level abstractions.

**Workflow:**

Write/annotate Rust code with Verus specs.

Verus + SMT solver checks that all conditions are valid.

If everything is proven, the code is accepted as correct (under those specs).



# Verus: Ghost Code

**What is Ghost Code?**

Special annotations and code segments used only for verification.

Written alongside actual Rust code, but removed after compilation.

No runtime overhead, as ghost code never runs.

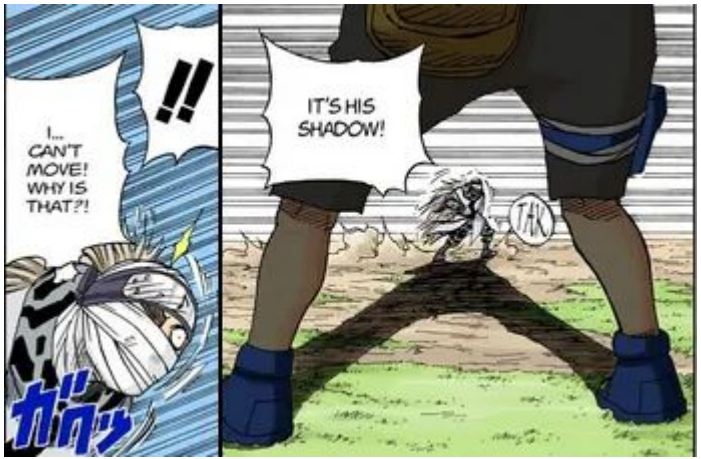
**Main Uses**

Specify preconditions and postconditions of functions.

Express and verify loop invariants.

Model abstract states and ownership (e.g., using linear ghost types).

https://tvtropes.org/pmwiki/pmwiki.php/Main/ShadowPin




Consider executable code in the concrete world to be a projection of the ghost code in the abstract world.
The ghost code can carry more abstract information than the executable code across function, module boundaries.
Verus “lifts” most executable code to the abstract world automatically .
You can also create your own abstract entities(assumptions, ghost types)

https://github.com/rverus-lang

# Linear ghost permission: PointsTo from vstd(Verus Standard Library

Raw point manipulation is dangerous.

How to play with raw pointers safely?

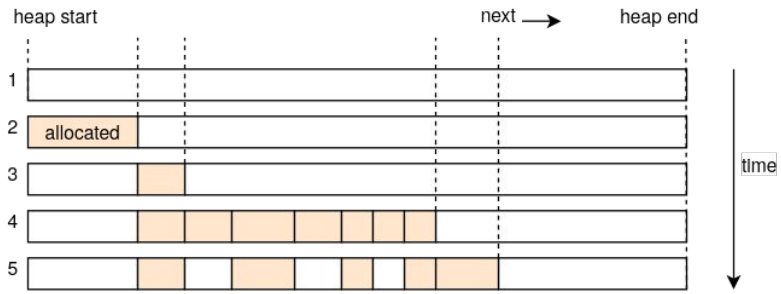
How to read/write a typed object from/to a memory address safely?

Safety condition: a type in Rust has a layout: size and alignment. It is undefined behavior to write a typed object to an address with wrong size alignment.

What does the allocator do?

The allocator returns memory regions that do not overlap.

https://os.phil-opp.com/allocator-designs/




External spec for the allocator in Vstd

```
#[cfg(feature = "std")]
#[verifier::external_body]
pub fn allocate(size: usize, align: usize) -> (pt: (
    *mut u8,
    Tracked<PointsToRaw>,
    Tracked<Dealloc>,
))
    requires
        valid_layout(size, align),
        size != 0,
    ensures
        pt.1@.is_range(pt.0.addr() as int, size as int),

        pt.2@@ == (DeallocData {
            addr: pt.0.addr(),
            size: size as nat,
            align: align as nat,
            provenance: pt.1@.provenance(),
        }),
        pt.0.addr() as int % align as int == 0,
        pt.0@.metadata == Metadata::Thin,
        pt.0@.provenance == pt.1@.provenance(),
```

Proving pointer addition preserves alignment

```
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;
    // vstd::arithmetic::div_mod::lemma_mod_adds(addr as int, size as int, alignment as int);
}
```

Writing to an array

```
// A pedagogical example showcasing the usage of PointsToRaw ghost permission
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
    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);
    ptr_mut_write(p as *mut V, Tracked(&mut pointsToFirst), first);
    let provenance = expose_provenance(p);
    let new_p: *mut V = with_exposed_provenance(p as usize + size, provenance);
    let tracked mut pointsToSecond = pointsToSecond.into_typed::<V>((new_p as usize) as
usize);
    ptr_mut_write(new_p , Tracked(&mut pointsToSecond), second);
}
```

# Summary of Our Work

- We investigate the memory safety guarantee of the Rust programming language and the organization of Rust standard library.
- We explore how ghost code of Verus can express safety conditions explicitly and eliminate informal comments and runtime assertions in std.
- We formally verify the functional correctness of the BinaryHeap push method
- We craft a simple proof of the correctness of pointer indexing, showcasing how linear ghost type works
- We also explore how user-defined ghost types can argument function interface with more information and meta states.(Omitted)

# Conclusions

**Verifying std is feasible but requires manual specs**.

**Incremental approach**:

Start with small modules, proceed to more complex data structures.

**Verus + Rust**:

A synergy that uses Rust's ownership as a foundation, then adds formal specs for higher-level properties.

**Looking Ahead**:

If widely adopted, this can greatly increase trust in critical Rust code—especially in OS or embedded contexts.

# Q&A / References

The Verus Project

Rust Standard Library Docs

[Prusti, Creusot, Kani model checkers] (tools for Rust verification)

Any specific conference papers about Verus, concurrency proofs, etc.