Verifying the Rust Standard Library Using Verus

Elanor Tang, Travis Hance (MPI), Chris Hawblitzel (Microsoft),
Natalie Neamtu, Jake Ginesin, and Bryan Parno

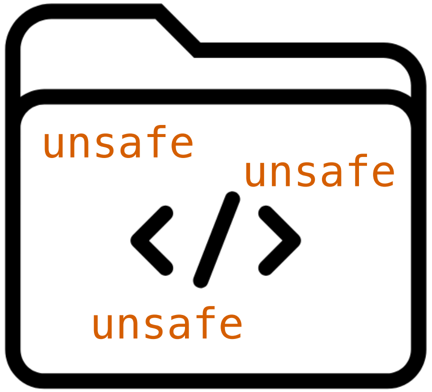
Carnegie Mellon University

2025 New England Systems Verification Day

Why Verify the Rust Standard Library?

Memory safety errors

...except in ***unsafe*** code



Rust standard library

20

CVEs

Goal: Provide safety and correctness guarantees

Icon created by SANB from Noun Project

Challenges and Tool Motivation

The Rust standard library is…

Evolving

Large
(500,000 SLOC)

Implemented with **unsafe** code

Verification tools must…

Provide ***“Rust-like”*** proof environment

Provide good ***automation***

Reason about ***raw pointer operations***, among others

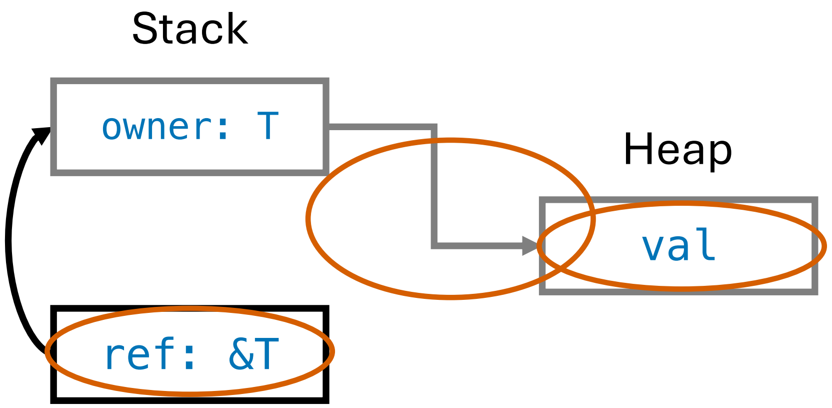
Verus does all of this

Icons created by Larea and Waldis Production from Noun Project

Rust: Ownership, References, and Borrowing

- Each value in Rust has a variable that's its ***owner***.
- Create an alias by making a ***reference***.
  - Called ***borrowing***. Done with **&** operator.
  - Think of a reference ***&T*** as ***(T, ptr)***.

✓ Enforced by the Rust ***borrow-checker***.



Rust shared reference

A reference can be ***shared*** (immutable) or ***mutable***, which determines the read/write access allowed.

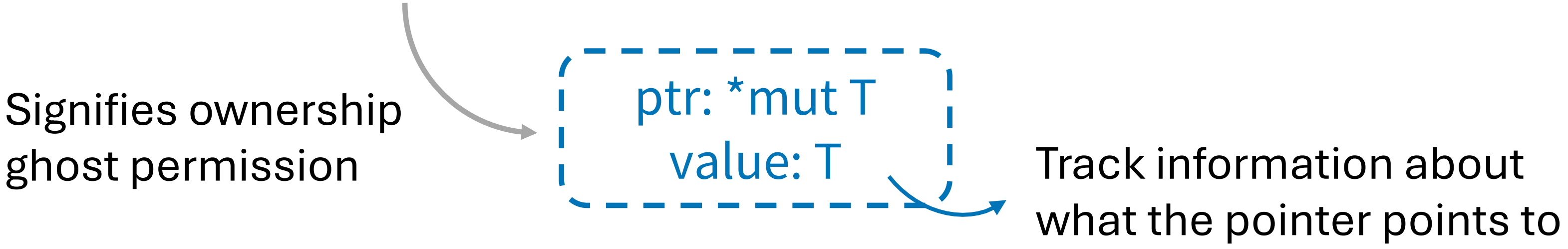
The ***lifetime*** of a reference cannot be longer than the owner's lifetime.

Verus: Ownership Ghost Permissions

- ***Ownership ghost permissions*** track the ***evolving state of a resource*** in unsafe code.
- Type-check permissions with Rust’s ***borrow-checker*** to ensure safety.

```
unsafe {
    let (p, Tracked(mut points_to)) = allocate::<u32>(4);

}
```



Signifies ownership ghost permission

ptr: *mut T
value: T

Track information about what the pointer points to

(example simplified for demonstration purposes)

Ownership Ghost Permissions: Mutability

Mutability of permission reference must ***match*** mutability of the pointer operation.

- Signature of `ptr_mut_write` requires a ***mutable*** reference, to ensure exclusive access to memory.

```
unsafe {
    let (p, Tracked(mut points_to)) = allocate::<u32>(4);
    ptr_mut_write(p, Tracked(&mut points_to), 5);


}
```

(example simplified for demonstration purposes)

Ownership Ghost Permissions: Mutability

Mutability of permission reference must ***match*** mutability of the pointer operation.

- Signature of `ptr_mut_write` requires a ***mutable*** reference, to ensure exclusive access to memory.
- Enforced by Rust’s ***borrow-checker***.

```
unsafe {
    let (p, Tracked(mut points_to)) = allocate::<u32>(4);
    ptr_mut_write(p, Tracked(&points_to), 5);      // FAILS



}
```

(example simplified for demonstration purposes)

Ownership Ghost Permissions: Lifetime

Permission is valid for ***exactly as long*** as the allocation’s lifetime.

- Signature of `deallocate` requires ***ownership transfer*** of `points_to`.

```
unsafe {
    let (p, Tracked(mut points_to)) = allocate::<u32>(4);
    ptr_mut_write(p, Tracked(&mut points_to), 5);
    deallocate(p, 4, Tracked(points_to));

}
```

(example simplified for demonstration purposes)

Ownership Ghost Permissions: Lifetime

Permission is valid for ***exactly as long*** as the allocation’s lifetime.

- Signature of `deallocate` requires ***ownership transfer*** of `points_to`.
- Rust borrow-checker ***forbids references*** after that: it is no longer in scope.

```
unsafe {
    let (p, Tracked(mut points_to)) = allocate::<u32>(4);
    ptr_mut_write(p, Tracked(&mut points_to), 5);
    deallocate(p, 4, Tracked(points_to));
    ptr_mut_write(p, Tracked(&mut points_to), 5);  // FAILS
}
```

(example simplified for demonstration purposes)

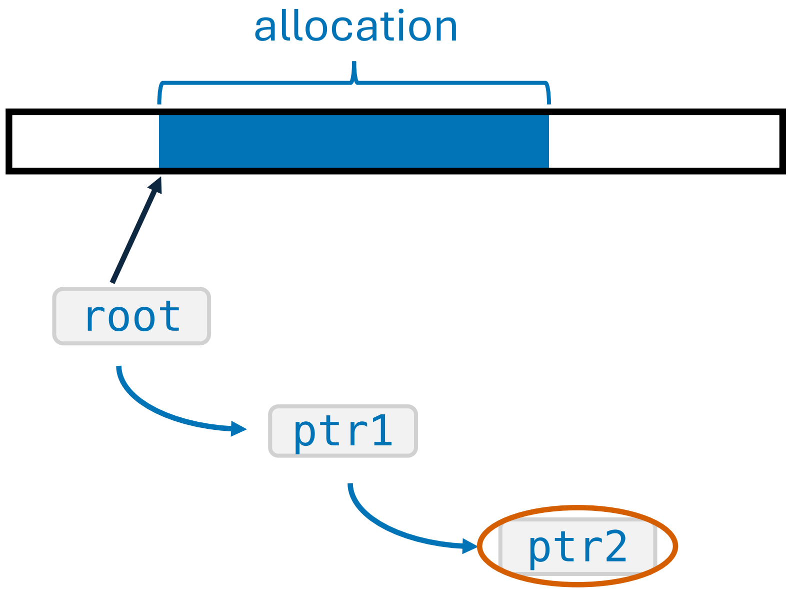
Challenge #1: Handling Pointer Provenance

Provenance captures what you are allowed to *do* with a pointer, based on the *source it was derived from*.

Spatial

Temporal

Mutability



Challenge #1: Handling Pointer Provenance

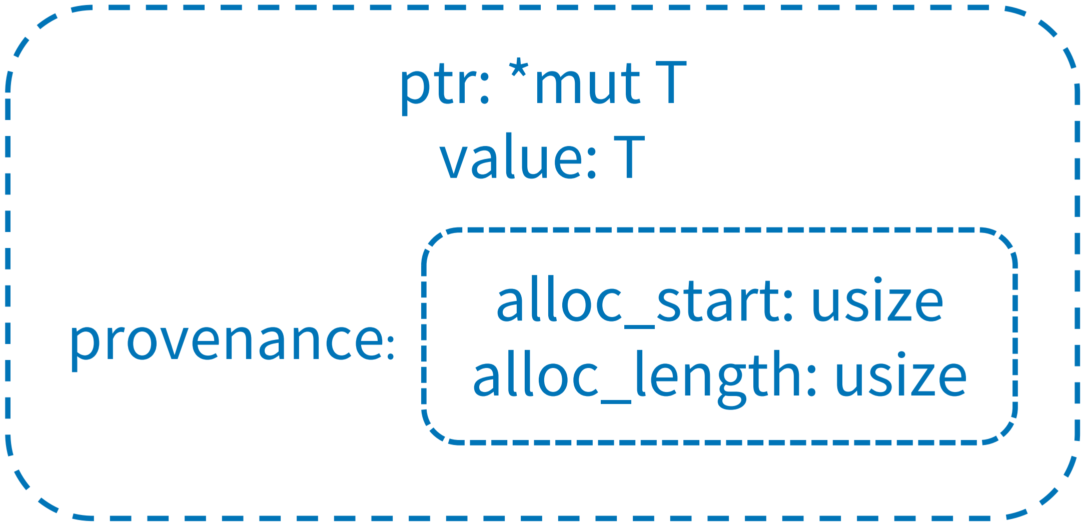
Provenance captures what you are allowed to *do* with a pointer, based on the *source it was derived from*.

✓Spatial → Extend ownership ghost permissions with provenance information

✓Temporal

✓Mutability

Addressed by Rust's borrow-checker on the lifetime and mutability of ghost permisions



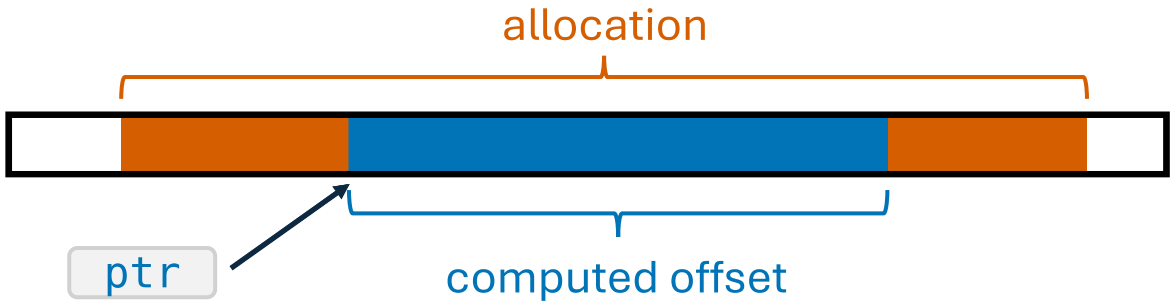
Extended ownership ghost permission

Example: `ptr::add`

```
pub unsafe fn add(ptr: *const T, count: usize) -> *const T
```

Advance `ptr: *const T` by `count` elements of type `T`

Example: `ptr::add`



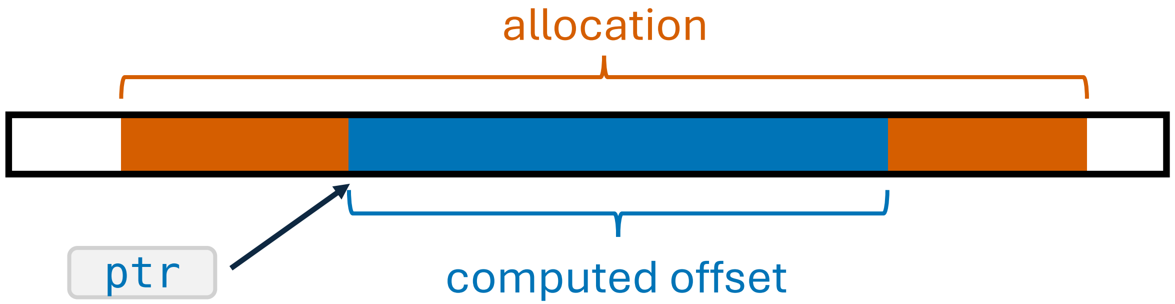
```
pub unsafe fn add(ptr: *const T, count: usize) -> *const T
```

If the computed offset is non-zero, then

1. `ptr` must have a valid allocation (not freed).

2. Memory range between `ptr` and the result must be within bounds.

Example: `ptr::add`



```
pub unsafe fn add_verus<T>(ptr: *const T, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> *const T
```

If the computed offset is non-zero, then

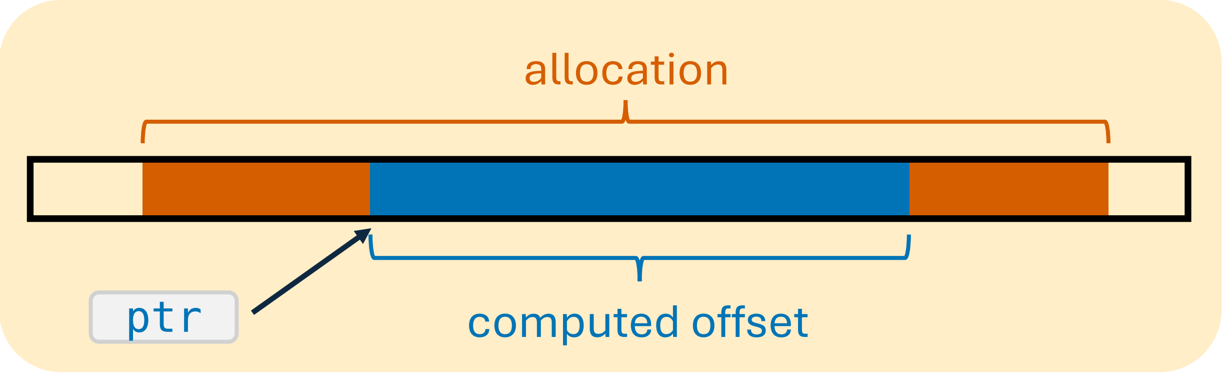
1. `ptr` must have a valid allocation (not freed).

provenance: alloc_start: usize
alloc_length: usize

Tells us that this memory has not been deallocated

2. Memory range between `ptr` and the result must be within bounds.

Example: `ptr::add`



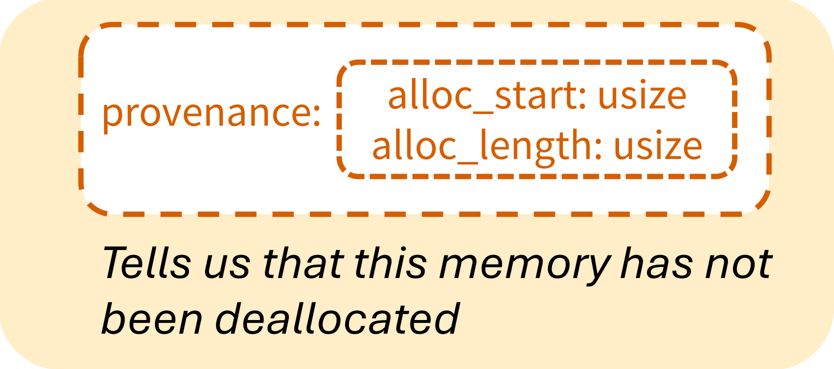
```
pub unsafe fn add_verus<T>(ptr: *const T, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> *const T
```

If the computed offset is non-zero, then

✓ ❶ `ptr` must have a valid allocation (not freed).

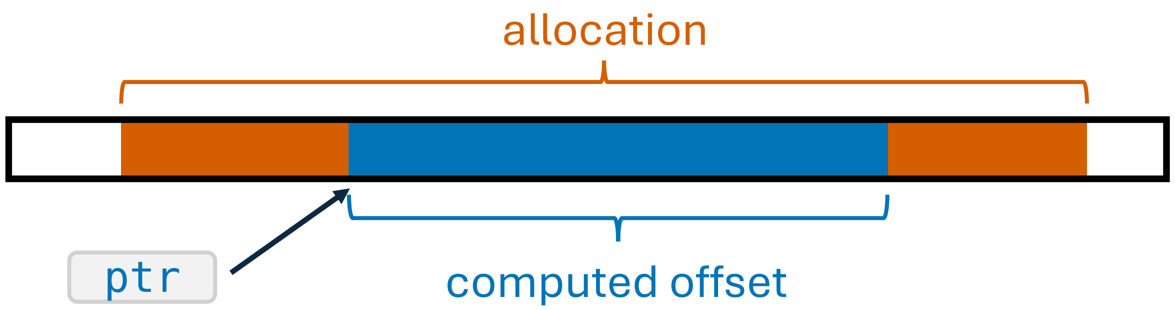
```
perm.provenance() == ptr.provenance
```

❷ Memory range between `ptr` and the result must be within bounds.



Tells us that this memory has not been deallocated

Example: `ptr::add`



```
pub unsafe fn add_verus<T>(ptr: *const T, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> *const T
```

If the computed offset is non-zero, then

✓ ❶ `ptr` must have a valid allocation (not freed).

```
perm.provenance() == ptr.provenance
```

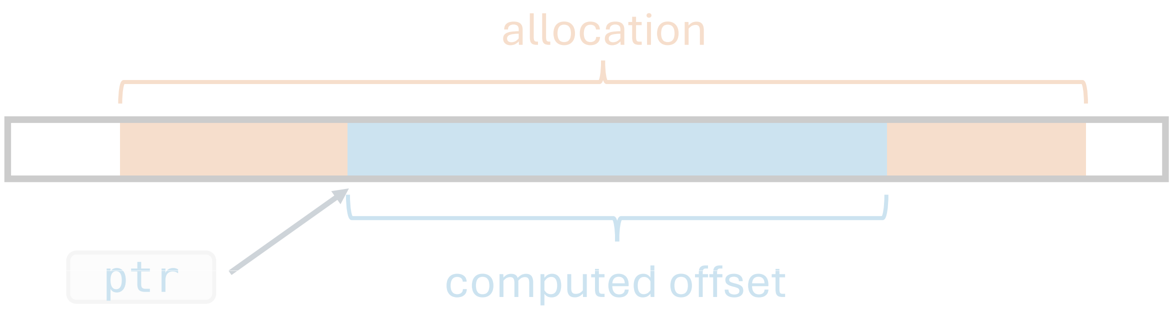
provenance: alloc_start: usize
alloc_length: usize

Tells us that this memory has not been deallocated

✓ ❷ Memory range between `ptr` and the result must be within bounds.

```
ptr.in_bounds(perm.provenance.alloc_start(),
              perm.provenance.alloc_start() + perm.provenance.alloc_length(),
              count)
```

Example: `ptr::add`



```
pub unsafe fn add_verus<T>(ptr: *const T, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> *const T
```

If the computed offset is non-zero, then

provenance: alloc_start: usize alloc_length: usize

✓ 1 `ptr` must have a valid allocation (not freed)

`perm.pr`

memory has not
d

The only provenance information we needed to add was `alloc_start` and `alloc_length`

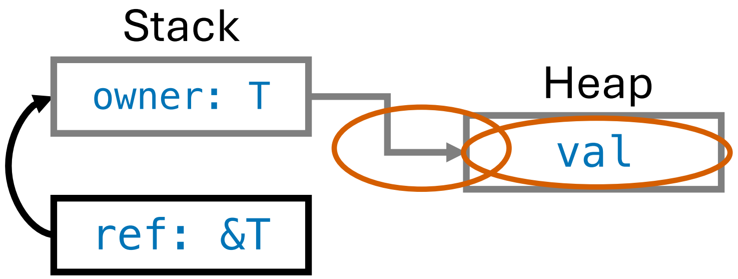
✓ 2 Memory range between `ptr` and the result must be within bounds.

```
ptr.in_bounds(perm.provenance.alloc_start(),
              perm.provenance.alloc_start() + perm.provenance.alloc_length(),
              count)
```

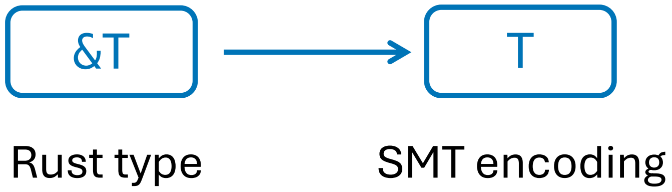
Challenge #2: Shared Reference SMT Encoding

Think of `&T` as `(T, ptr)`

- In most cases, we only care about `T`



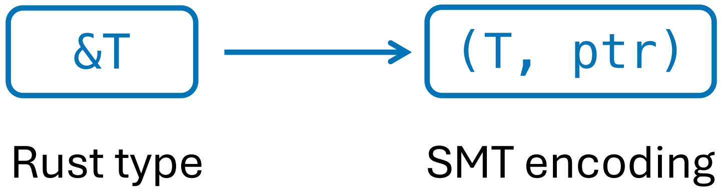
Encoding A



Add `ptr_info(v: &T)` function to get pointer information as needed

- ➢ ***Simpler for users***
- ➢ ***Harder to track and update pointer information internally***

Encoding B

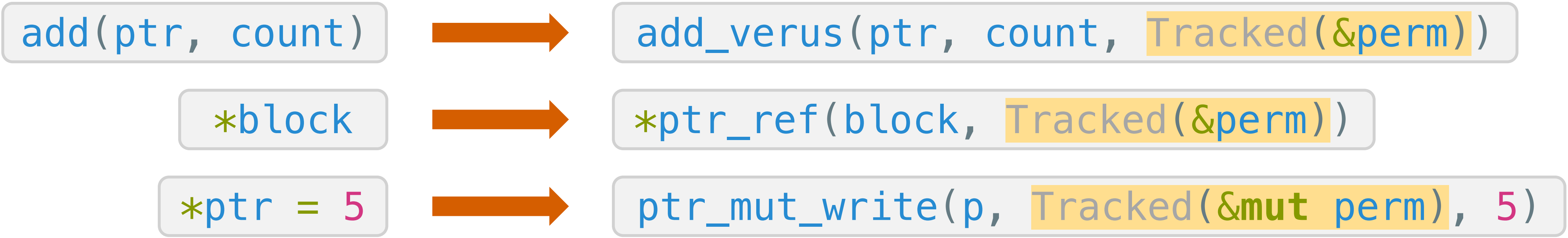


Use `&T` only when you actually need the pointer

- ➢ ***Straightforward to implement***
- ➢ ***Often unavoidable to have &T when we do not need the pointer***

Challenge #3: Ergonomically Incorporating Spec/Proof Code into Existing Rust Code

Need to use Verus versions of functions



```
add(ptr, count)    → add_verus(ptr, count, Tracked(&perm))
*block             → *ptr_ref(block, Tracked(&perm))
*ptr = 5           → ptr_mut_write(p, Tracked(&mut perm), 5)
```

Solution: Support in progress for attribute-based syntax

```
#[with_ghost_arg(Tracked(perm): Tracked<&PointsToRaw>)]
pub unsafe fn add<T>(ptr: *const T, count: usize) -> *const T
```

Challenge #3: Ergonomically Incorporating Spec/Proof Code into Existing Rust Code

Need to use Verus versions of functions

```
add(ptr, count)
```

→

```
add_verus(ptr, count, Tracked(&perm))
```

Need to do this in a way that still enables type-checking, so we can keep borrow-checking our permissions

Solution: Support in progress for attribute-based syntax

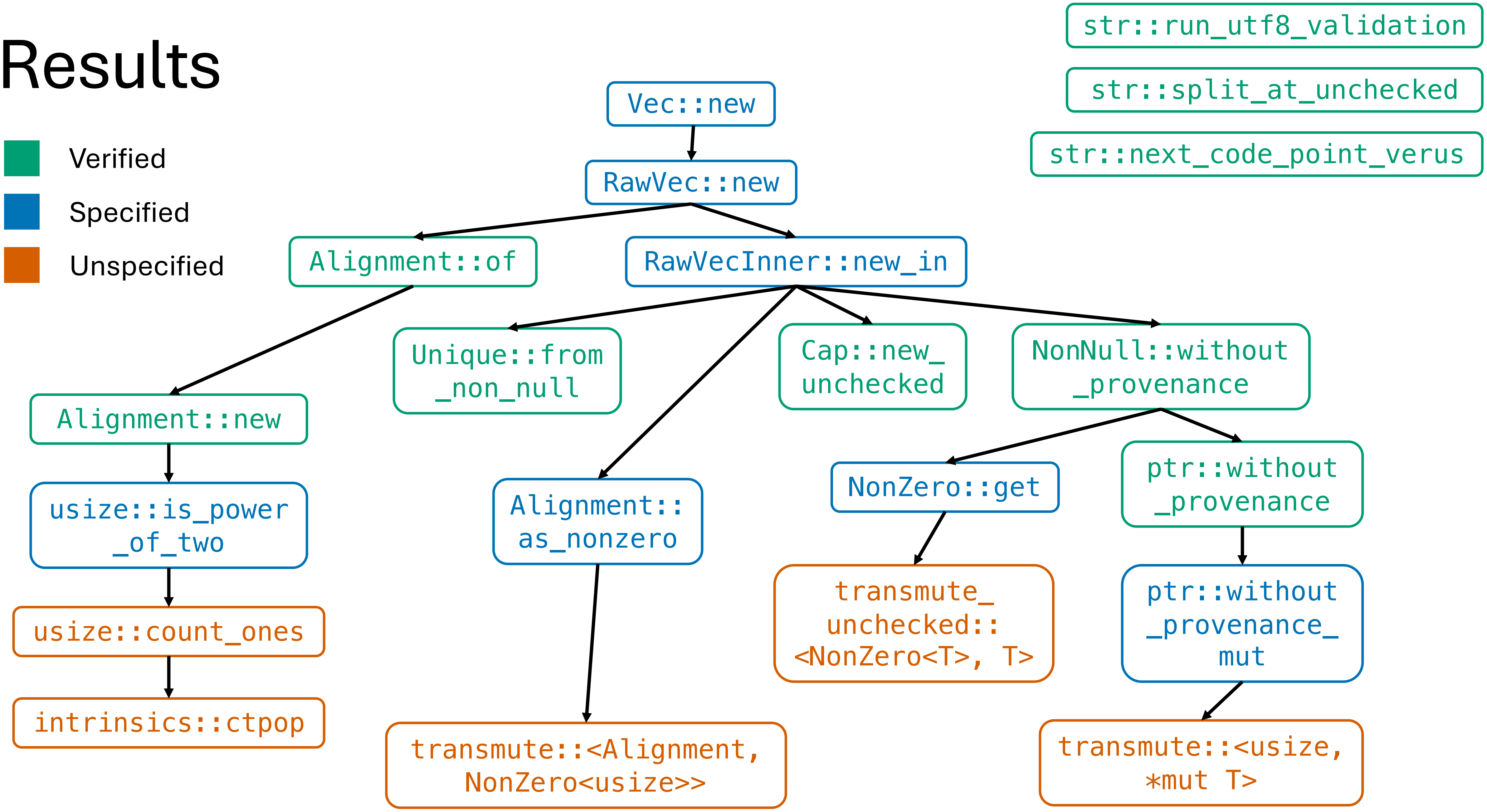
```
#[with_ghost_arg(Tracked(perm): Tracked<&PointsToRaw>)]
pub unsafe fn add<T>(ptr: *const T, count: usize) -> *const T
```

Results

🟩 Verified

🟦 Specified

🟧 Unspecified



str::run_utf8_validation

str::split_at_unchecked

str::next_code_point_verus

Vec::new

RawVec::new

Alignment::of

RawVecInner::new_in

Alignment::new

Unique::from_non_null

Cap::new_unchecked

NonNull::without_provenance

usize::is_power_of_two

Alignment::as_nonzero

NonZero::get

ptr::without_provenance

usize::count_ones

transmute_unchecked::<NonZero<T>, T>

ptr::without_provenance_mut

intrinsics::ctpop

transmute::<Alignment, NonZero<usize>>

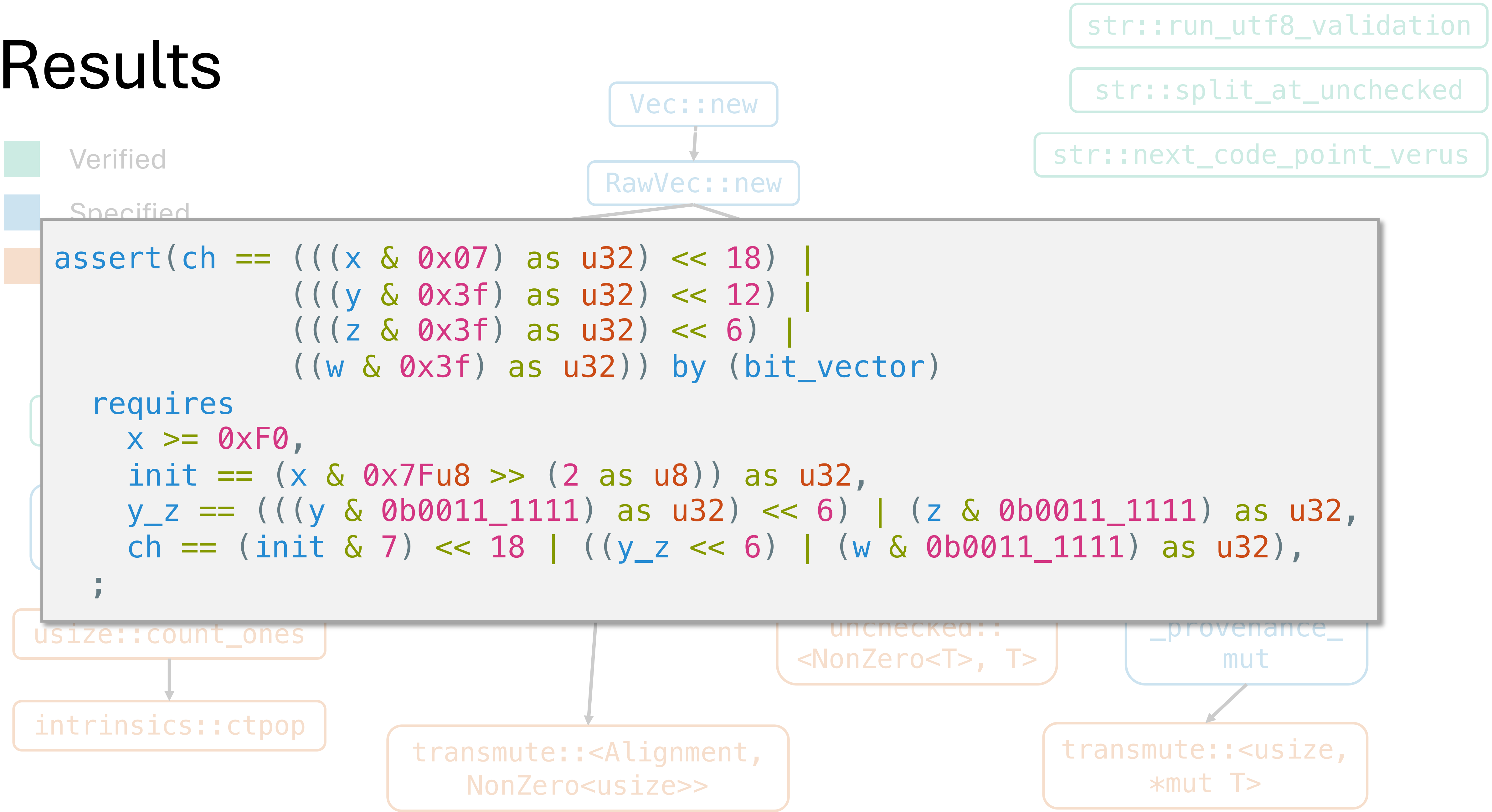
transmute::<usize, *mut T>

Results

🟩 Verified

🟦 Specified

🟧



str::run_utf8_validation
str::split_at_unchecked
str::next_code_point_verus
Vec::new
RawVec::new
usize::count_ones
intrinsics::ctpop
transmute::<Alignment, NonZero<usize>>
unchecked::<NonZero<T>, T>
provenance mut
transmute::<usize, *mut T>

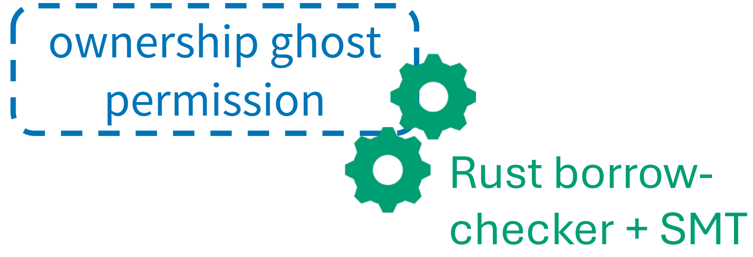
```
assert(ch == (((x & 0x07) as u32) << 18) |
             (((y & 0x3f) as u32) << 12) |
             (((z & 0x3f) as u32) << 6) |
             ((w & 0x3f) as u32)) by (bit_vector)
  requires
    x >= 0xF0,
    init == (x & 0x7Fu8 >> (2 as u8)) as u32,
    y_z == (((y & 0b0011_1111) as u32) << 6) | (z & 0b0011_1111) as u32,
    ch == (init & 7) << 18 | ((y_z << 6) | (w & 0b0011_1111) as u32),
  ;
```

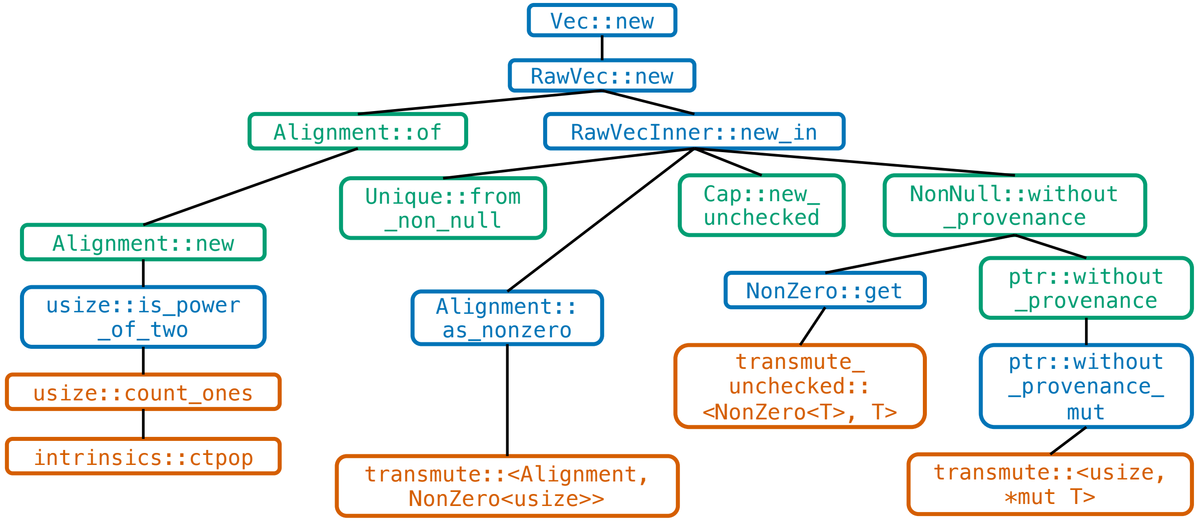
Elanor Tang
elanor@cmu.edu

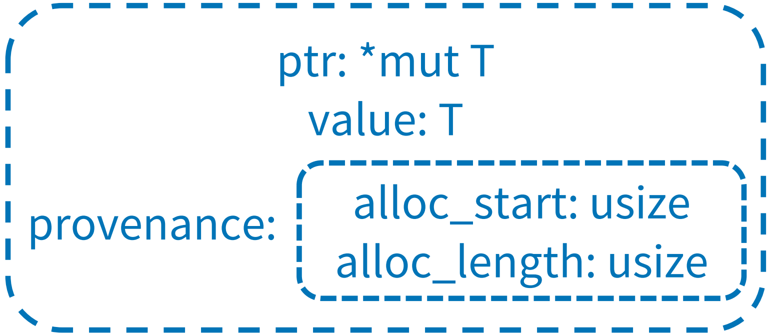
Thank you!

Recap

- Rely on Rust's ***borrow-checker*** and ***ownership ghost types***.
- Straightforward ***pointer provenance*** model.
- Capable of ***verifying complex***, ***real-world code***.







This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE2140739. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.