# Initial Steps Toward Verifying the Rust Standard Library Using Verus

Elanor Tang, Travis Hance, and Bryan Parno

Carnegie Mellon University

# Why Verify the Rust Standard Library?

- *Safe* interface
- *Unsafe* implementation

**Want to provide safety and correctness guarantees**

# Safety Depends on Correctness

```rust
pub const fn from_utf8(v: &[u8]) -> Result<&str, Utf8Error> {
    match run_utf8_validation(v) {
        Ok(_) => {
            // SAFETY: validation succeeded.
            Ok(unsafe { from_utf8_unchecked(v) })
        }
        Err(err) => Err(err),
    }
}
```

Checks to see if a sequence of bytes is in UTF-8 format

If so, calls the unsafe function `from_utf8_unchecked()`

```rust
pub const unsafe fn from_utf8_unchecked(v: &[u8]) -> &str {
    // SAFETY: the caller must guarantee that the bytes `v` are valid UTF-8.
    // Also relies on `&str` and `&[u8]` having the same layout.
    unsafe { mem::transmute(v) }
}
```

Performs the unsafe operation `transmute()`

# Safety Depends on Correctness

```rust
pub const fn from_utf8(v: &[u8]) -> Result<&str, Utf8Error> {
    match run_utf8_validation(v) {
        Ok(_) => {
            // SAFETY: validation succeeded.
            Ok(unsafe { from_utf8_unchecked(v) })
        }
        Err(err) => Err(err),
    }
}
```

Checks to see if a sequence of bytes is in UTF-8 format

If so, calls the unsafe function `from_utf8_unchecked()`

```rust
pub const unsafe fn from_utf8_unchecked(v: &[u8]) -> &str {
    // SAFETY: the caller must guarantee that the bytes `v` are valid UTF-8.
    // Also relies on `&str` and `&[u8]` having the same layout.
    unsafe { mem::transmute(v) }
}
```

Performs the unsafe operation `transmute()`

# Safety Depends on Correctness

```rust
pub const fn from_utf8(v: &[u8]) -> Result<&str, Utf8Error> {
    match run_utf8_validation(v) {
        Ok(_) => {
            // SAFETY: validation succeeded.
            Ok(unsafe { from_utf8_unchecked(v) })
        }
        Err(err) => Err(err),
    }
}
```

Checks to see if a sequence of bytes is in UTF-8 format

If so, calls the unsafe function `from_utf8_unchecked()`

**Safety of `mem::transmute(v)` depends on correctness of `run_utf8_validation(v)`**

```rust
pub const unsafe fn from_utf8_unchecked(v: &[u8]) -> &str {
    // SAFETY: the caller must guarantee that the bytes `v` are valid UTF-8.
    // Also relies on `&str` and `&[u8]` having the same layout.
    unsafe { mem::transmute(v) }
}
```

Performs the unsafe operation `transmute()`

# Initial Target: Verifying the String Library

Enables greater security

Presents nontrivial verification challenges

# Challenges: Verifying the String Library

- Reasoning about pointer *provenance*

Provenance: Captures what you are allowed to do with a pointer based on the source it was derived from

Snippet from string standard library

```
unsafe {
    let block = ptr.add(index) as *const usize;
}
```

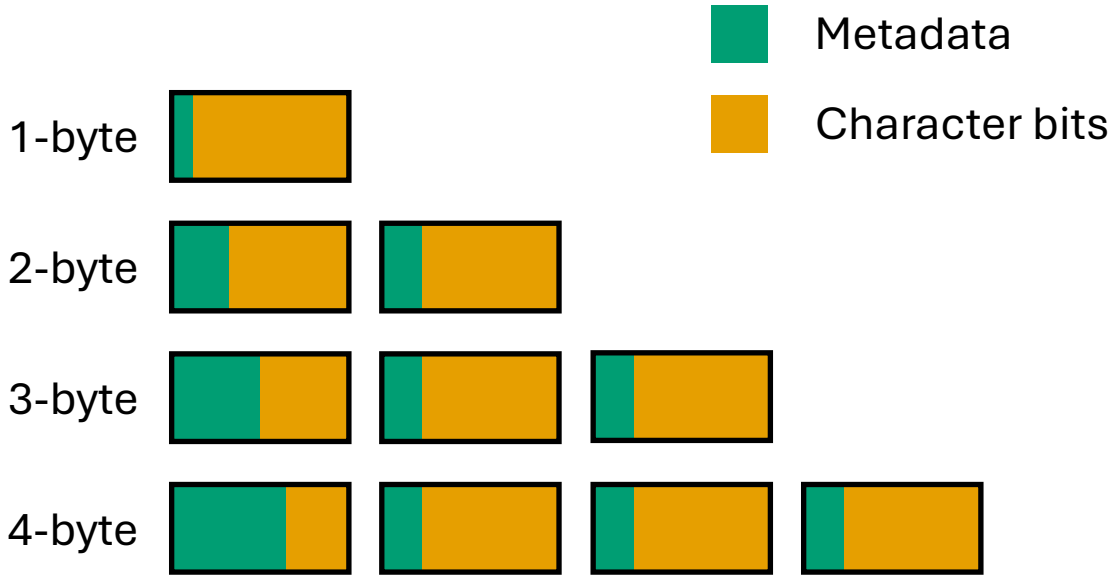
Unsafe, precondition from Rust documentation involves the provenance of `ptr`

Will talk about why it is hard later

Will talk more about `add()` later

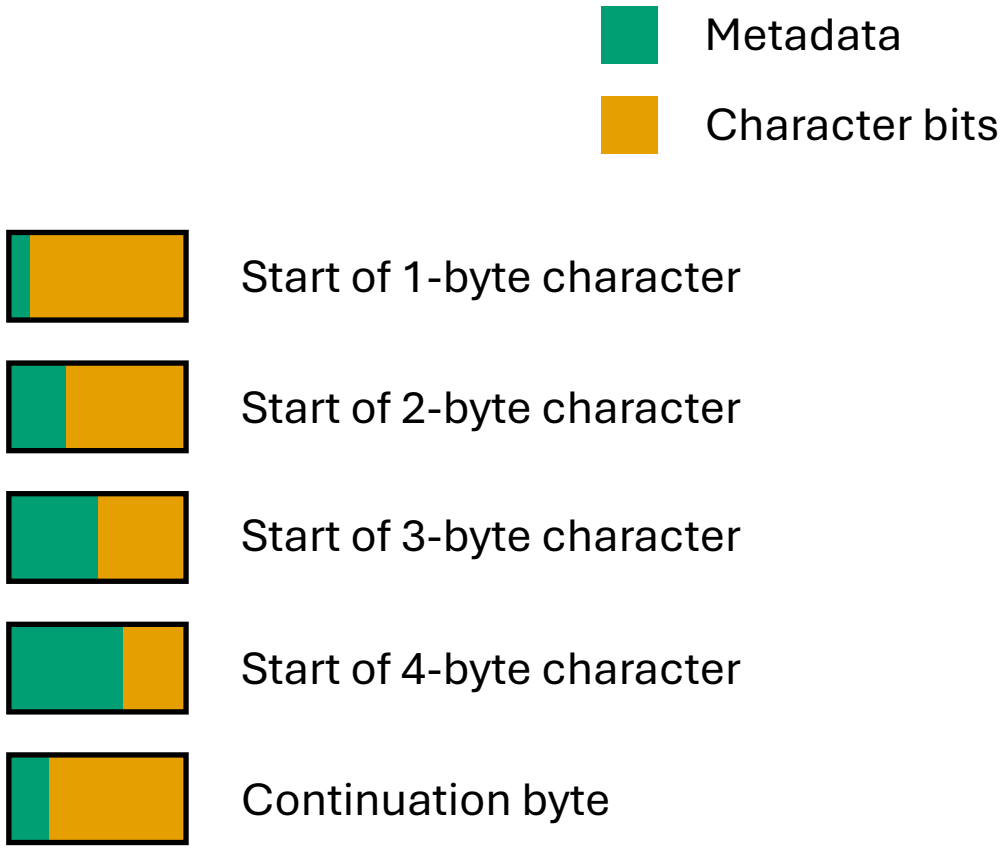
# Challenges: Verifying the String Library

- Reasoning about pointer *provenance*
- Complex UTF-8 reasoning
  - Variable-width encoding of characters
  - Complicated interpretation of byte values



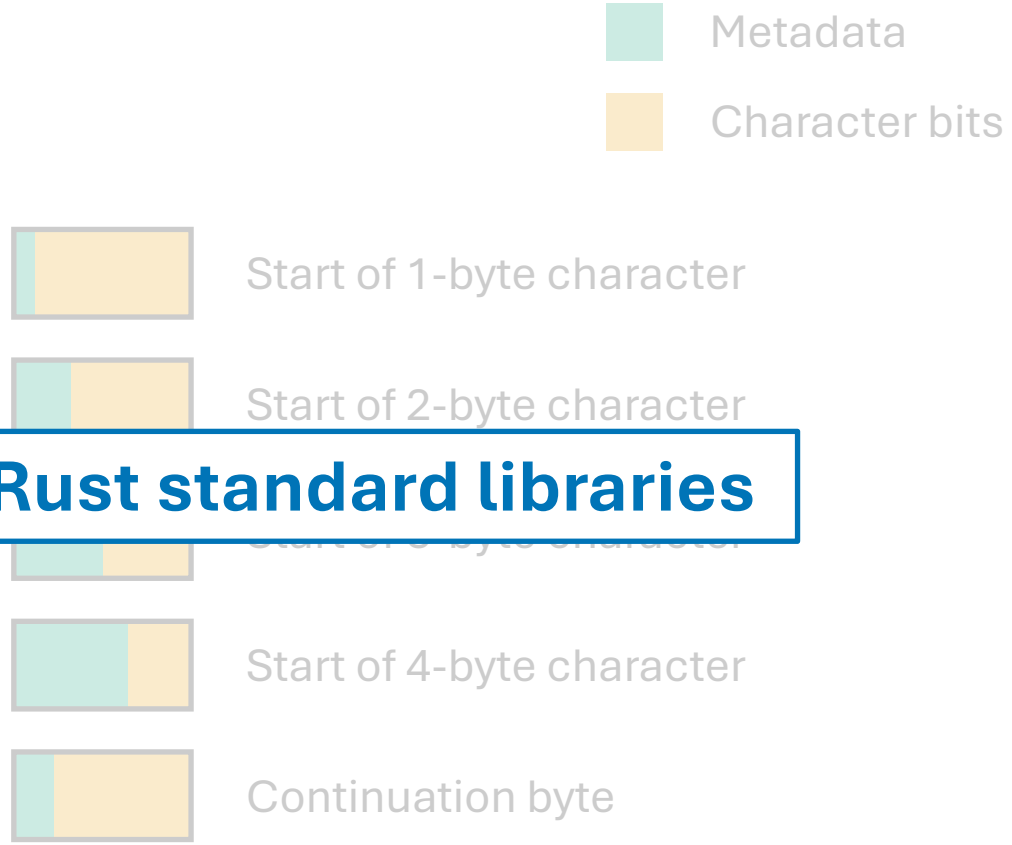
# Challenges: Verifying the String Library

- Reasoning about pointer *provenance*
- Complex UTF-8 reasoning
  - Variable-width encoding of characters
  - Complicated interpretation of byte values
- Low-level bit manipulation



# Challenges: Verifying the String Library

- Reasoning about pointer *provenance*
- Complex UTF-8 reasoning
  - Variable-width encoding of characters
  - byte values
- Low-level bit manipulation



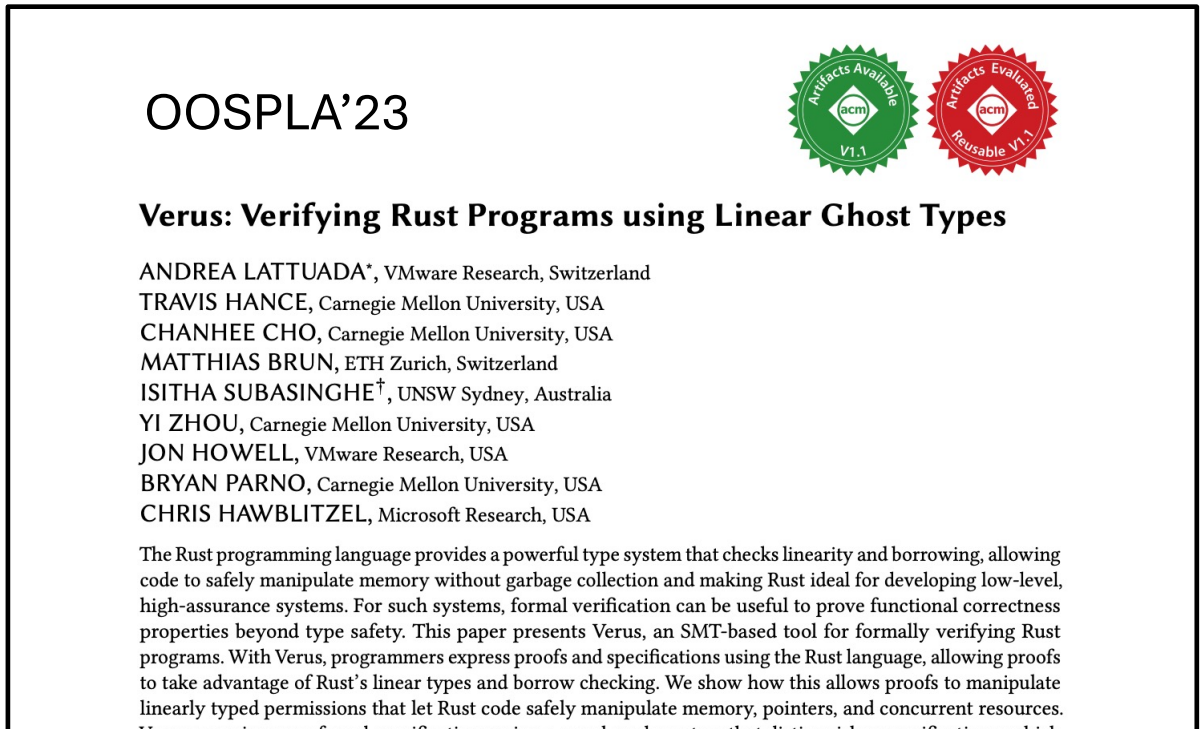

**Solutions apply to verifying other Rust standard libraries**

# Overview

- **Verus**
  - *Ownership ghost permissions* to reason about pointers
- **Challenges**
  - Key challenge: **Handling pointer provenance**
  - Verus challenge: Ergonomically incorporating spec/proof code into existing Rust code
- **What Went Well**
  - Successful verification of complex functions

# Our Approach: Verus

- Automated SMT-based verification tool
- Uses ***ownership ghost permissions*** which are ***borrow-checked*** to safely manipulate pointers



OOSPLA'23

Artifacts Available acm V1.1

Artifacts Evaluated acm Reusable V1.1

**Verus: Verifying Rust Programs using Linear Ghost Types**

ANDREA LATTUADA*, VMware Research, Switzerland
TRAVIS HANCE, Carnegie Mellon University, USA
CHANHEE CHO, Carnegie Mellon University, USA
MATTHIAS BRUN, ETH Zurich, Switzerland
ISITHA SUBASINGHE†, UNSW Sydney, Australia
YI ZHOU, Carnegie Mellon University, USA
JON HOWELL, VMware Research, USA
BRYAN PARNO, Carnegie Mellon University, USA
CHRIS HAWBLITZEL, Microsoft Research, USA

The Rust programming language provides a powerful type system that checks linearity and borrowing, allowing code to safely manipulate memory without garbage collection and making Rust ideal for developing low-level, high-assurance systems. For such systems, formal verification can be useful to prove functional correctness properties beyond type safety. This paper presents Verus, an SMT-based tool for formally verifying Rust programs. With Verus, programmers express proofs and specifications using the Rust language, allowing proofs to take advantage of Rust's linear types and borrow checking. We show how this allows proofs to manipulate linearly typed permissions that let Rust code safely manipulate memory, pointers, and concurrent resources.

```
ptr : *mut T
value: T
```

Track information about what the pointer points to

# Ownership Ghost Permissions

Signifies ownership ghost permission

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

}
```

**Stack**

```
p = 0x1004
```

**Heap**

| Address | Value |
|---|---|
| 0x1000 | – |
| 0x1004 | UnInit |
| 0x1008 | – |
| 0x100c | – |

**Verus' perspective**

```
p = 0x1004
points_to = {
    ptr: 0x1004,
    value: UnInit,
}
```

(example simplified for demonstration purposes)

# Ownership Ghost Permissions

Signifies ownership ghost permission

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

**Stack**

```
p = 0x1004
```

**Heap**

| Address | Value |
|---|---|
| 0x1000 | – |
| 0x1004 | 5 |
| 0x1008 | – |
| 0x100c | – |

**Verus' perspective**

```
p = 0x1004
points_to = {
    ptr: 0x1004,
    value: 5,
}
```

(example simplified for demonstration purposes)

# Features of Ownership Ghost Permissions

- Mutability of permission matches mutability of the pointer operation.
- Lifetime of permission is tied to lifetime of allocation.

Enforced by ***borrow-checking*** the permissions

- ***Erase*** permissions during compilation.

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

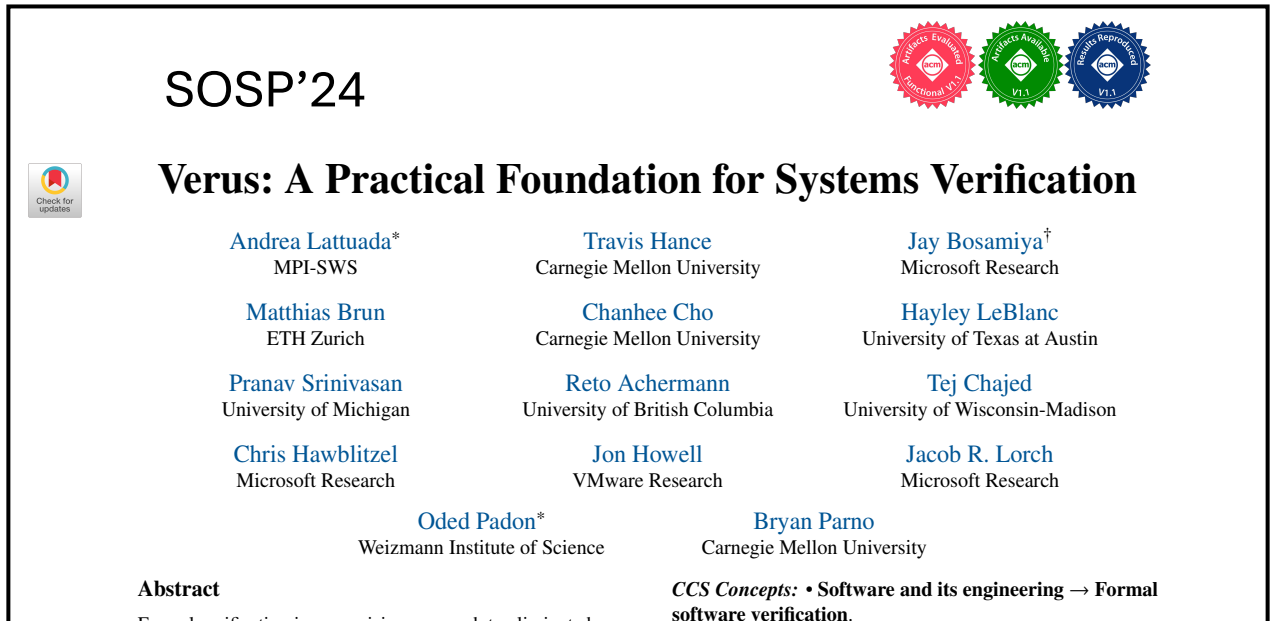
(example simplified for demonstration purposes)

# Verus: Benefits and Limitations

✓ Can reason about safe and unsafe code

✓ Has the automation to scale (~250K total Verus LOC, codebases up to 60K LOC)

X Needs more features to handle pointer provenance reasoning



SOSP'24

Verus: A Practical Foundation for Systems Verification


Andrea Lattuada* MPI-SWS

Travis Hance Carnegie Mellon University

Jay Bosamiya† Microsoft Research

Matthias Brun ETH Zurich

Chanhee Cho Carnegie Mellon University

Hayley LeBlanc University of Texas at Austin

Pranav Srinivasan University of Michigan

Reto Achermann University of British Columbia

Tej Chajed University of Wisconsin-Madison

Chris Hawblitzel Microsoft Research

Jon Howell VMware Research

Jacob R. Lorch Microsoft Research

Oded Padon* Weizmann Institute of Science

Bryan Parno Carnegie Mellon University



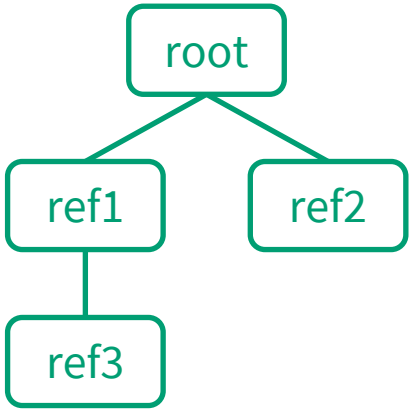
**Abstract**



*CCS Concepts:* • **Software and its engineering → Formal software verification.**

# Key Challenge: Handling Provenance

- Rust has no formal provenance model
- Many provenance models are complicated
  - E.g., tree and stacked borrows




> "The exact structure of provenance is not yet specified, but the permission defined by a pointer's provenance has a ***spatial*** component, a ***temporal*** component, and a ***mutability*** component."
> —Rust pointer documentation

Visualization source: Tree Borrows preprint

# Our Solution: Handling Provenance

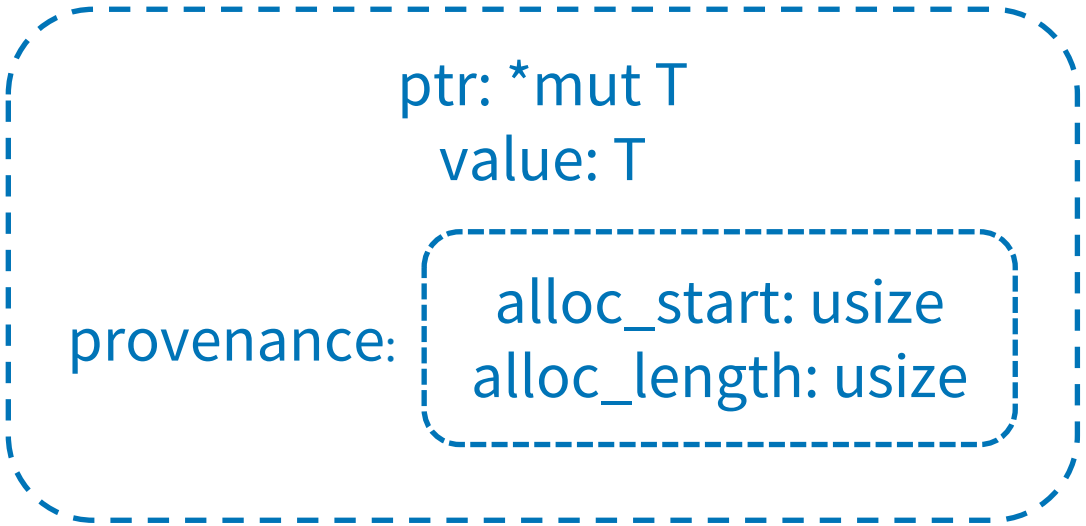
**Key idea: Abstract over the provenance model**

✓Spatial → Extend ownership ghost permissions with provenance information

✓Temporal

✓Mutability

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

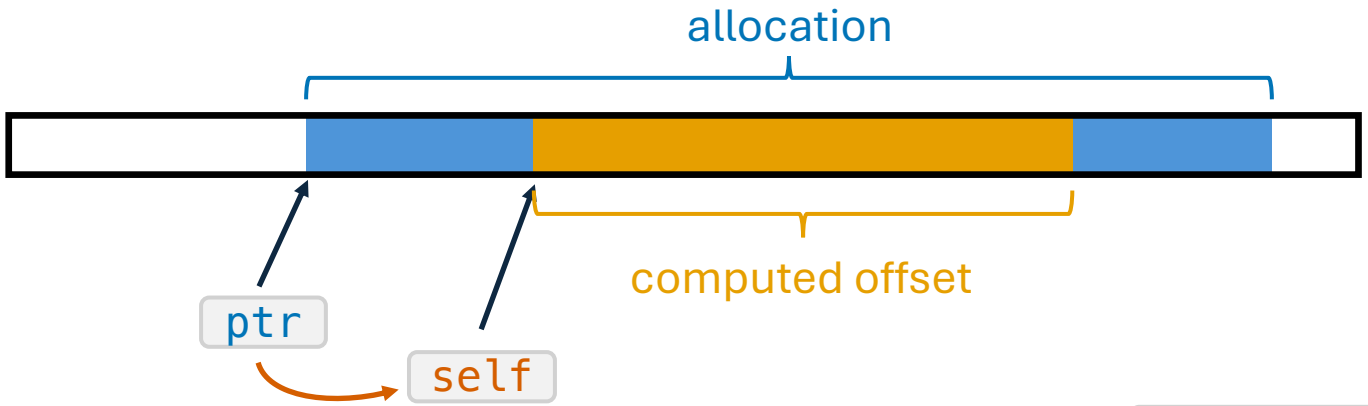



Extended ownership ghost permission

# Example: `ptr.add()`

If the computed offset is non-zero, then

1. `self` must be derived from some pointer to an allocated object
2. Memory range between `self` and the result must be within bounds



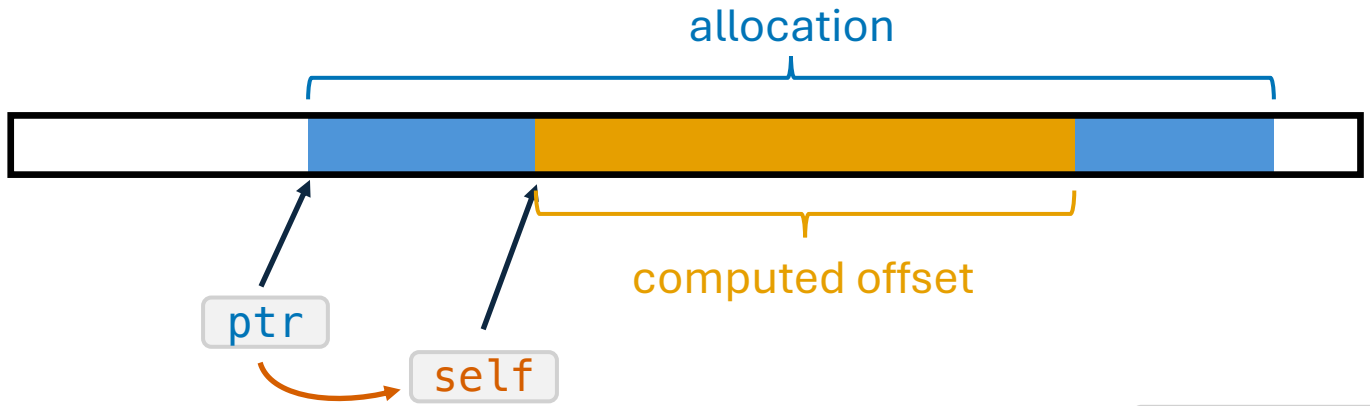

Advance `self: *const T` by `count` elements of type `T`

```
impl<T: Sized> *const T {
  pub const unsafe fn add(self, count: usize) -> output

  {...}
}
```

# Example: `ptr.add()`

If the computed offset is non-zero, then

1. `self` must be derived from some pointer to an allocated object
2. Memory range between `self` and the result must be within bounds




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

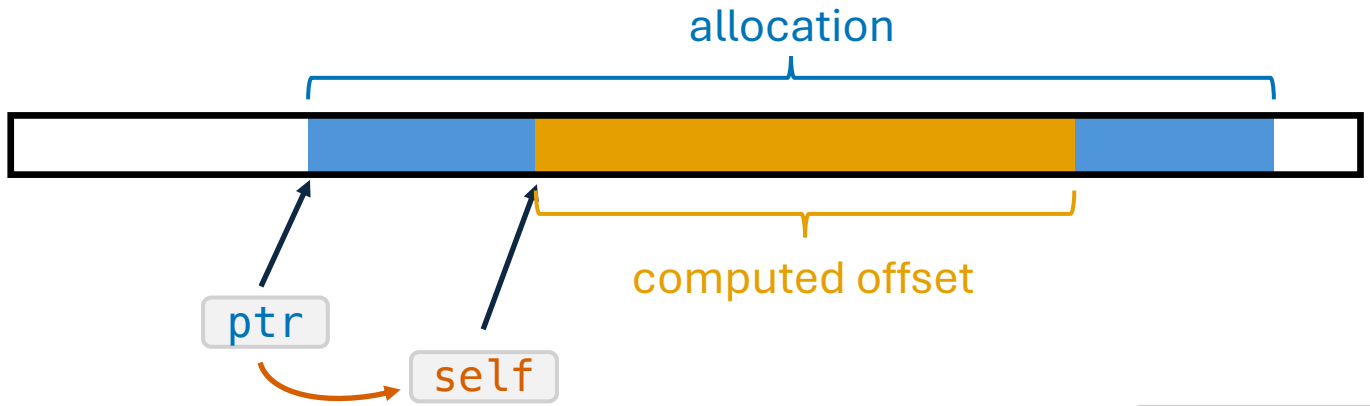
```
impl<T: Sized> *const T {
  pub const unsafe fn add_verus(self, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> (output: Self)
  requires

  {...}
}
```

provenance: alloc_start: usize, alloc_length: usize

# Example: `ptr.add()`

If the computed offset is non-zero, then

1. `self` must be derived from some pointer to an allocated object
2. Memory range between `self` and the result must be within bounds




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

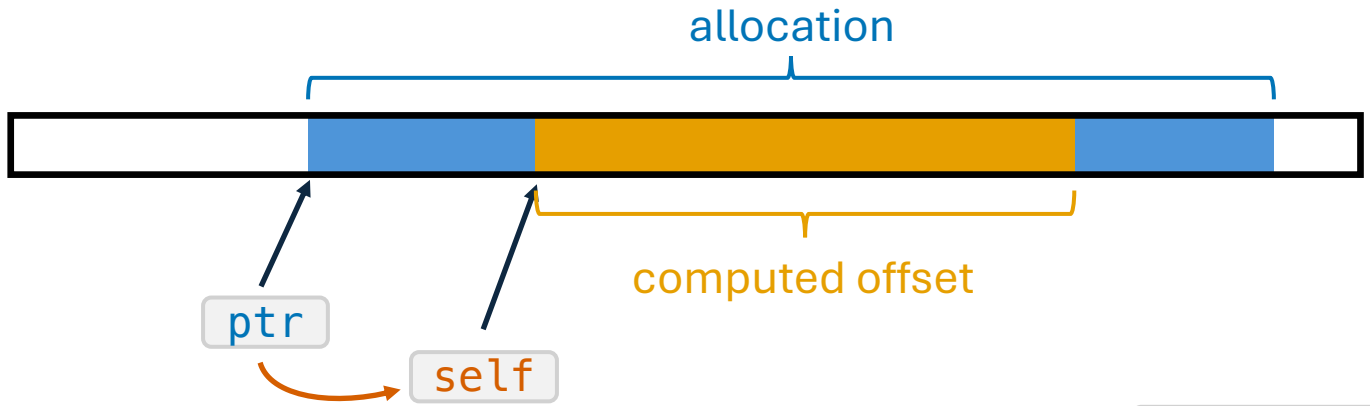
```
impl<T: Sized> *const T {
  pub const unsafe fn add_verus(self, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> (output: Self)
  requires
    count * size_of::<T>() > 0
     ==>



  {...}
}
```

provenance: alloc_start: usize, alloc_length: usize

# Example: `ptr.add()`

If the computed offset is non-zero, then

1. `self` must be derived from some pointer to an allocated object
2. Memory range between `self` and the result must be within bounds




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

```
impl<T: Sized> *const T {
  pub const unsafe fn add_verus(self, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> (output: Self)
  requires
    count * size_of::<T>() > 0
    ==> perm.provenance() == self@.provenance



  {...}
}
```

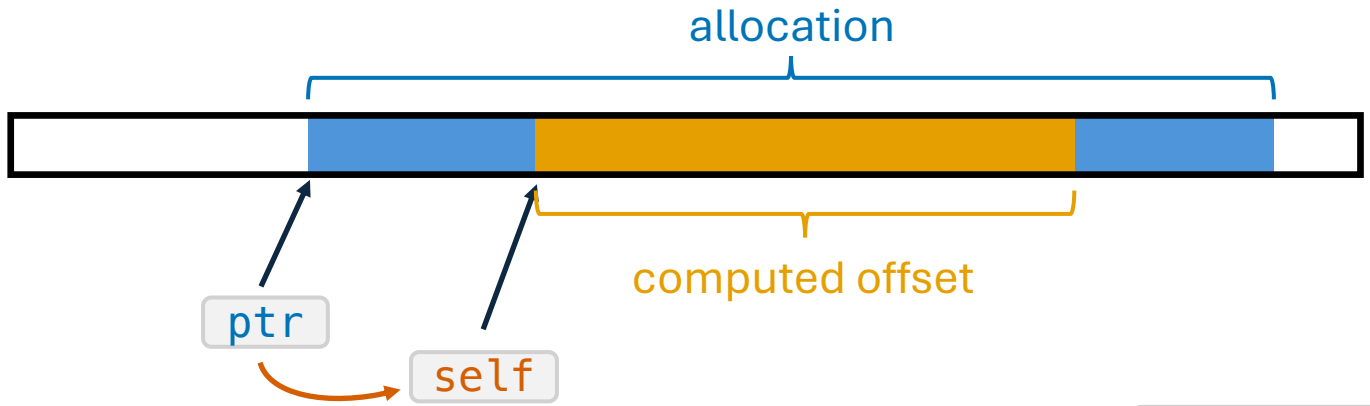
provenance: alloc_start: usize, alloc_length: usize

# Example: `ptr.add()`

If the computed offset is non-zero, then

✓ ❶ `self` must be derived from some pointer to an allocated object

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




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

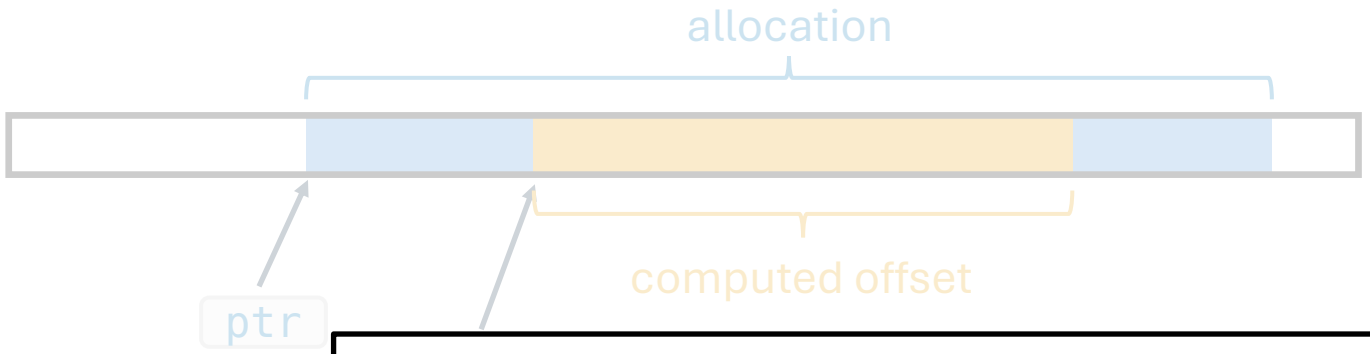
```
impl<T: Sized> *const T {
  pub const unsafe fn add_verus(self, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> (output: Self)
  requires
    count * size_of::<T>() > 0
    ==> perm.provenance() == self@.provenance
        && (self@.addr + count * size_of::<T>())
            < self@.provenance.alloc_start() + self@.provenance.alloc_len()
        && self@.addr >= self@.provenance.alloc_start(),
  {...}
}
```

provenance: alloc_start: usize, alloc_length: usize

# Example: `ptr.add()`

If the computed offset is non-zero, then

1. `self` must be derived from some pointer to an allocated object
2. Memory range between `self` and the result must be within bounds




**The only provenance information we needed was `alloc_start` and `alloc_length`**

s of type `T`

```
impl<T: Sized> *const T {
  pub const unsafe fn add_verus(self, count: usize, Tracked(perm): Tracked<&PointsToRaw>) -> (output: Self)
  requires
    count * size_of::<T>() > 0
    ==> perm.provenance() == self@.provenance
        && (self@.addr + count * size_of::<T>())
            < self@.provenance.alloc_start() + self@.provenance.alloc_len()
        && self@.addr >= self@.provenance.alloc_start(),
  {...}
}
```

provenance:
- alloc_start: usize
- alloc_length: usize

# Additional Wrinkle

Carrying around pointer/provenance information at the SMT level may get expensive.

Original Encoding


Rust type SMT encoding

- **Efficient**
- **Loses information about the pointer and its provenance**

New Encoding


Rust type SMT encoding

- **Includes pointer and provenance information**
- **Currently evaluating how expensive**

**Potential challenge: Incorporating both encodings**

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

- Need to use Verus versions of functions

```
ptr.add(count)
```

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

```
*block
```

```
*ptr_ref(block, Tracked(&perm))
```

```
*ptr = 5
```

```
ptr_mut_write(p, Tracked(&mut perm), 5)
```

- Rust currently lacks support for ghost code

# Proposed Solution: Ergonomically Incorporating Spec/Proof Code into Existing Rust Code

```
pub fn ptr_mut_write<T>(ptr: *mut T, Tracked(points_to): Tracked<&mut PointsTo<T>>, v: T)
```

- Proposal to move ownership ghost function parameters out of the main function signature

```
pub fn ptr_mut_write<T>(ptr: *mut T, v: T)
    with_ghost_arg (points_to: &mut PointsTo<T>)
```

- Support in progress for attribute-based syntax

```
#[with_ghost_arg(points_to: &mut PointsTo<T>)]
pub fn ptr_mut_write<T>(ptr: *mut T, v: T)
```

# Proposed Solution: Ergonomically Incorporating Spec/Proof Code into Existing Rust Code

```
pub fn ptr_mut_write<T>(ptr: *mut T, Tracked(points_to): Tracked<&mut PointsTo<T>>, v: T)
```

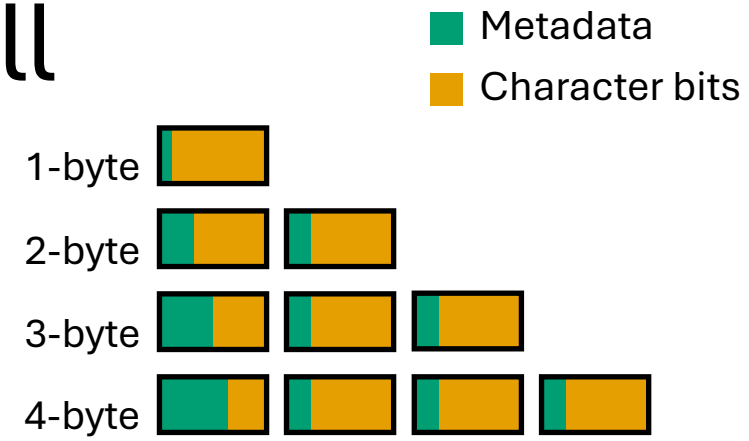
- Proposal to move ownership ghost function parameters out of the main function signature

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

- Support in progress for attribute-based syntax

```
#[with_ghost_arg(points_to: &mut PointsTo<T>)]
pub fn ptr_mut_write<T>(ptr: *mut T, v: T)
```

# Putting It Together: What Went Well




Successful verification of complex functions

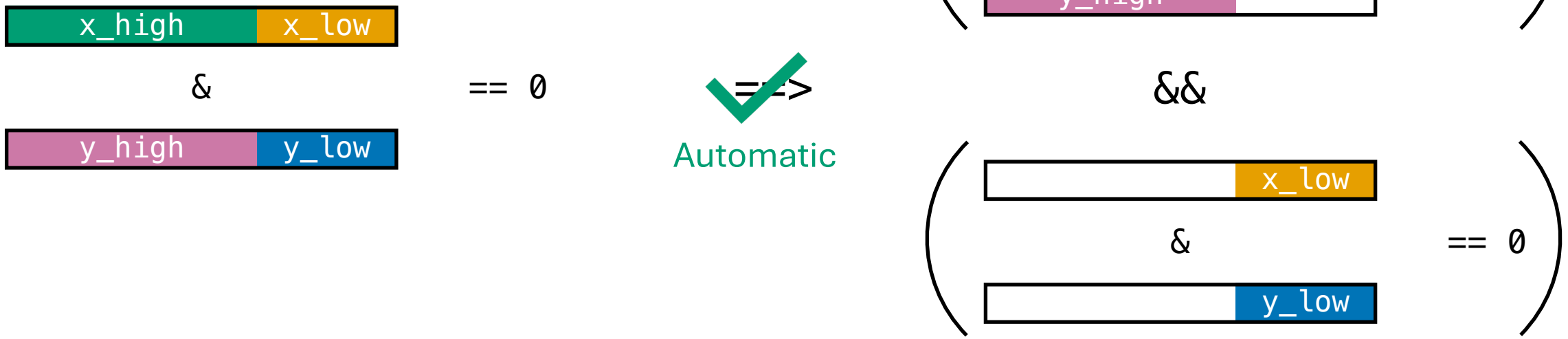
Clean mathematical spec for UTF-8

```
pub const fn run_utf8_validation(v_ref: &[u8]) -> (result: Result<(), Utf8Error>)
    ensures
        is_ok(&result) <==> valid_utf8(v_ref@),
{

    Low-level automatic bitwise proofs



}
```

# Low-level bitwise proofs

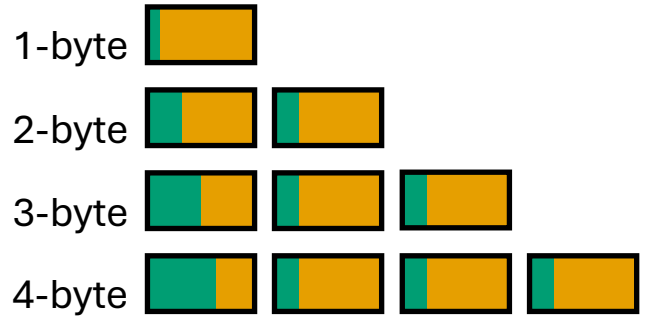
All assertions about bitwise operations were **automatically** proved

- Context of example
  - Showing block of bytes is ASCII
  - Endianness reasoning



# Putting It Together: What Went Well

Successful verification of complex functions



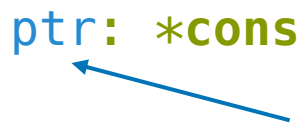

Clean mathematical spec for UTF-8

```
pub const fn run_utf8_validation(v_ref: &[u8]) -> (result: Result<(), Utf8Error>)
    ensures
        is_ok(&result) <==> valid_utf8(v_ref@),
{

}
```

Low-level ***automatic*** bitwise proofs

Pointer ***provenance*** reasoning

# Pointer Provenance Reasoning via Ownership Ghost Permissions



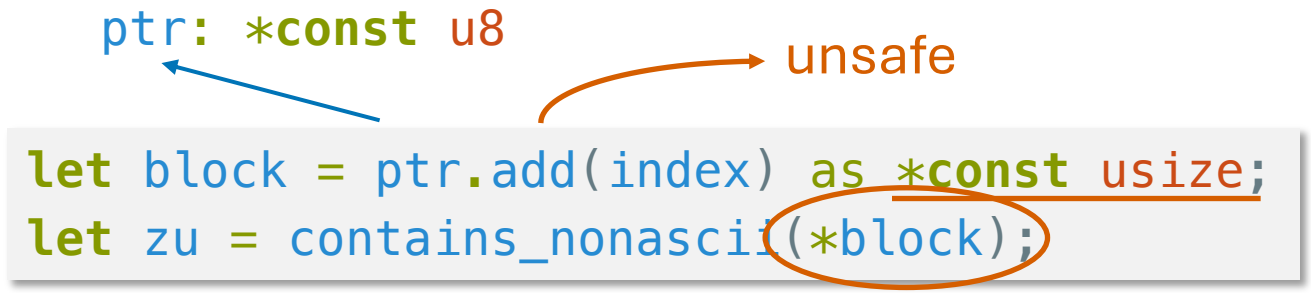

```
let block = ptr.add(index) as *const usize;
let zu = contains_nonascii(*block);
```

Unverified code in `run_utf8_validation()`

```
let block = ptr.add_verus(index, Tracked(perm.into_raw_shared())) as *const usize;
let tracked zu_sub_perm = perm.subrange(index, size_of::<usize>());
proof {
    lemma_mod_eqv_0_trans(zu_sub_perm.ptr()@.addr as int,
        size_of::<usize>() as int, align_of::<usize>() as int);
}
let tracked zu_usize_perm = zu_sub_perm.cast_points_to::<usize>();
let x = *ptr_ref(block, Tracked(zu_usize_perm));
let zu = contains_nonascii(x);
```

Verified code in `run_utf8_validation()`

# Pointer Provenance Reasoning via Ownership Ghost Permissions




Unverified code in `run_utf8_validation()`

```
let block = ptr.add_verus(index, Tracked(perm.into_raw_shared())) as *const usize;
let tracked zu_sub_perm = perm.subrange(index, size_of::<usize>());
proof {
    lemma_mod_eqv_0_trans(zu_sub_perm.ptr()@.addr as int,
        size_of::<usize>() as int, align_of::<usize>() as int);
}
let tracked zu_usize_perm = zu_sub_perm.cast_points_to::<usize>();
let x = *ptr_ref(block, Tracked(zu_usize_perm));
let zu = contains_nonascii(x);
```

Verified code in `run_utf8_validation()`

# Pointer Provenance Reasoning via Ownership Ghost Permissions

```
ptr: *const u8
```

```
let block = ptr.add(index) as *const usize;
let zu = contains_nonascii(*block);
```

Unverified code in `run_utf8_validation()`

```
let block = ptr.add_verus(index, Tracked(perm.into_raw_shared())) as *const usize;
let tracked zu_sub_perm = perm.subrange(index, size_of::<usize>());
proof {
    lemma_mod_eqv_0_trans(zu_sub_perm.ptr()@.addr as int,
        size_of::<usize>() as int, align_of::<usize>() as int);
}
let tracked zu_usize_perm = zu_sub_perm.cast_points_to::<usize>();
let x = *ptr_ref(block, Tracked(zu_usize_perm));
let zu = contains_nonascii(x);
```

Verified code in `run_utf8_validation()`

# Recap: Essential Verification Tool Features

- ✓Mathematical specification language
- ✓Automation
- ✓Bit-level reasoning
- ○Ergonomically integrate spec/proof code into existing Rust code
- ○Provenance-specific pointer reasoning

- ○ Proposal to decouple executable and ownership ghost function parameters
- ○ Support in progress for attribute-based syntax

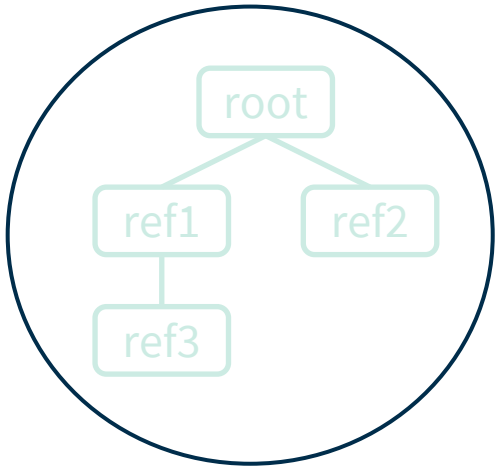
- ✓Verus has pointer reasoning via ownership ghost permissions
- ✓Added provenance information to permissions
- ○ Next step: experiment with more pointer manipulations

Elanor Tang
elanor@cmu.edu

# Recap: Key Discussion Question

**How to add abstraction over the provenance model?**

Abstraction




**We would love to hear your thoughts on the best approach!**

*Thank you!*

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE2140739. Any opinions, findings, and 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.