Taming Unsafe Rust with Safety Tags

A New Paradigm for Understanding Unsafe Code

Hui Xu

Fudan University

Oct 27, 2025

Background: Limitation of Rust and Current Efforts

- ❑Rust compiler cannot provide safety guarantee for unsafe code.
- ❑Current practice: Safety comments (informal).
- ❑Current effort: Program verification based on contract.

```rust
/// safety comments
unsafe fn foo() { ... }

#[contract::preconditions]
fn bar() {
    // safety comments
    unsafe { foo() }
}
#[contract::postconditions]
```

Specify the contract

Prove the correctness

Specify the contract

Our research vision for unsafe code handling:

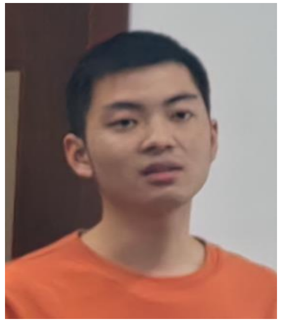
- *a lightweight yet formal approach.*

Artisan-Lab @ Fudan University

Hui Xu
Associate Professor

Zihao Rao
Ph.D. Student (year 2)
Rust Verification

Yehong Zhang
Ph.D. Student (year 3)
Rust API Robustness



Chenhao Cui
Master Student (year 3)
Rust Optimization, OS

Yuzhi Li
Master Student (year 2)
Rust OS Analysis

Changcheng Li
Master Student (year 2)
Range Analysis

Yilin Chen
Master Student(year 1)
Rust OS Analysis

Jiping Zhou
Research Assistant
tag-std

Outline

I. Safety Tags

II. A New Theory for Verification

III. Verification Practice with RAPx

Comment-based Approach for Unsafe Code Handling

- ❑ At unsafe function declarations, specify the requirements for safe use.
  - ➢ In natural language as comments or Rustdoc.

```
/// Safety Requirements: pointer p must be aligned for type T.
pub unsafe fn foo<T> (p: *const T) {
    ...
}
```

- ❑ At unsafe call sites, justify why the use of unsafe code is safe.

```
unsafe {
    // Justification: p is aligned.
    foo(p);
}
```

Issues of Comment-based Approach

- **Consistency:**
  - Missing safety requirements.
  - Inconsistent or incorrect requirements.
- **Ergonomics:**
  - Extensive and repetitive textual descriptions across functions.
- **Precision:**
  - Safety comments lack the precision of formally specified contracts.

Our Proposal: Safety Tags

- Basic version (RFC 3842, under review):
  - A safety tag is an abbreviation of a piece of safety comment.
  - Safety tags are implemented as Rust attributes, allowing them to be analyzed.
  - Safety tags can be compiled to docs.
- Advanced version (more precise):
  - Safet tags can have parameters.
  - A safety tag becomes a safety constraint in a domain-specific language.
  - Safety tags can be used as or translated to contracts.

Project URL: https://github.com/Artisan-Lab/tag-std

RFC 3842

zjp-CN commented on Jul 31 • edited

Summary

This RFC introduces a concise safety-comment convention for unsafe code in standard libraries: tag every public unsafe function with `#[safety::requires]` and call with `#[safety::checked]`.

Safety tags refine today's safety-comment habits: a featherweight syntax that condenses every requirement into a single, check-off reminder.

The following snippet compiles today if we enable enough nightly features, but we expect Clippy and Rust-Analyzer to enforce tag checks and provide first-class IDE support.

```
#[safety::requires( // 💡 define safety tags on an unsafe function
    valid_ptr = "src must be [valid](https://doc.rust-lang.org/std/ptr/index.html#safety) for reads",
    aligned = "src must be properly aligned, even if T has size 0",
    initialized = "src must point to a properly initialized value of type T"
)]
pub unsafe fn read<T>(ptr: *const T) { }

fn main() {
    #[safety::checked( // 💡 discharge safety tags on an unsafe call
        valid_ptr, aligned, initialized = "optional reason"
    )]
    unsafe { read(&()) };
}
```

Rendered

😊 👍 23 🎉 1 😕 1 ❤️ 2 👀 6

Use Case of Safety Tags

- At unsafe function declarations, specify safety tags.

```
#[safety::requires(Align)]
pub unsafe fn foo<T> (p: *const T) {
    ...
}
```

- At unsafe call sites, discharge all tags with reasons.
  - Three ways: checked, delegated, transformed.

```
unsafe {
    #[safety::checked(Align, "reason")]
    foo(p);
}
```

Safety Tag Delegation and Transformation

- Delegation: The unsafe caller directly inherit the safety tags of the callee.

```
#[safety::requires(Align)]
unsafe fn bar<T>(p: *mut T) {
    #[safety::delegated(Align)]
    unsafe { foo(p) }
}
```

- Transformation: The safety tags are transformed to other forms.

```
#[safety::requires(Align||ValidNum)]
unsafe fn bar<T>(p: *mut T, x: i32) {
    if x > 0 {
        #[safety::transformed(Align||ValidNum)]
        unsafe { foo(p); }
    }
}
```

Define Once, Reuse Multiple Times

- ❑Define safety tags in a 'toml' file within as assets.
- ❑The definition is automatically referenced and resolved.

```toml
package.name = "core"

[tag.Align]
args = [ "p", "T" ]
desc = "pointer `{p}` must be aligned for type `{T}`"
expr = "p % alignment(T) = 0"
url = "https://doc.rust-lang.org/nightly/std/ptr/index.html#alignment"

[tag.alias]
...
```

Demo: Enforce Safety Check as Linter

```
aisr@aisr:~/demo/foo$
```

Issues of Comment-based Methods

- Consistency:
  - ~~Missing safety requirements.~~
  - Inconsistent or incorrect requirements.
- Ergonomics:
  - ~~Extensive and repetitive textual descriptions across functions.~~
- Precision:
  - Safety comments lack the precision of formally specified contracts.

Safety Tags Can be Translated to Docs and Contracts

```
#[safety::requires(Align(p, T))]
```

doc

```
/// pointer p must be aligned for type T.
```

contract

```
#[rapx::requires(Align(p, T)]
```

Other forms of contracts and tools

Two Main Types of Safety Tags

- **Precondition: Constraint to be satisfied when calling the unsafe function.**
  - Most of such safety tags are sufficient and necessary condition;
  - Only a few of them are sufficient but unnecessary.
- **Hazard: Constraint satisfaction cannot be examined at the program point.**
  - Temporarily leave the program in a vulnerable state.

Example

```
pub const unsafe fn read<T>(src: *const T) -> T
```

§ Safety

Behavior is undefined if any of the following conditions are violated:

- `src` must be valid for reads.
- `src` must be properly aligned. Use `read_unaligned` if this is not the case.
- `src` must point to a properly initialized value of type `T`.

```
Precondition: ValidPtr(src, T, 1)
```

```
Precondition: Align(src, T)
```

```
Precondition: Init(src, T, 1)
```

Ownership of the Returned Value

`read` creates a bitwise copy of `T`, regardless of whether `T` is `Copy`. If `T` is not `Copy`, using both the returned value and the value at `*src` can violate memory safety. Note that assigning to `*src` counts as a use because it will attempt to drop the value at `*src`.

```
Hazard: Alias(src, ret) <= Trait(T, Copy)
```


https://doc.rust-lang.org/std/ptr/fn.read.html

Comparison of Safety Tags vs Contracts

- Contracts have postconditions, while safety tags do not.
  - Postconditions are often used by the function with interior unsafe code.
- Safety tags use hazard instead.
  - Automatic hazard elimination analysis.

```rust
fn foo(s: &mut String) {
    let s2 = ManuallyDrop::new(
        unsafe { ptr::read(s) }
    );
    assert_eq!(s2, "foo");
}
#[contract::postconditions]
```

Time of use: introduce hazard

Time of check: hazard eliminated

s2 is dead.

Issues of Comment-based Methods

- **Consistency:**
  - ~~Missing safety requirements.~~ ✓
  - ~~Inconsistent or incorrect requirements.~~ ?

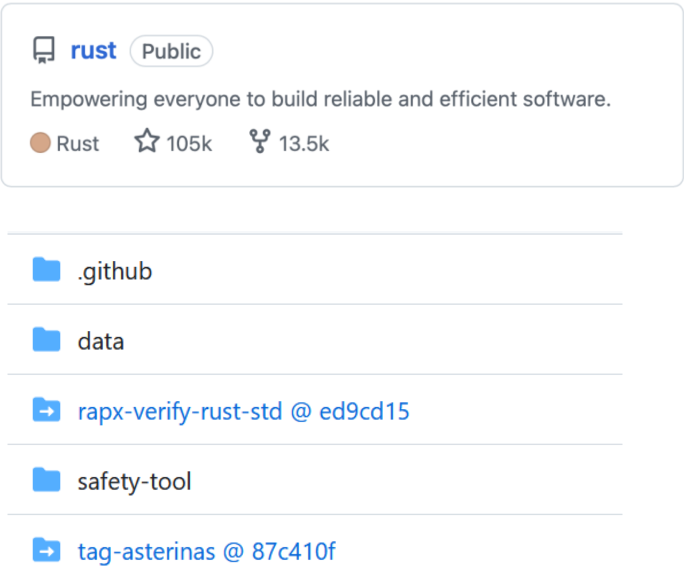
Safety tags also enables consistency check through verification.

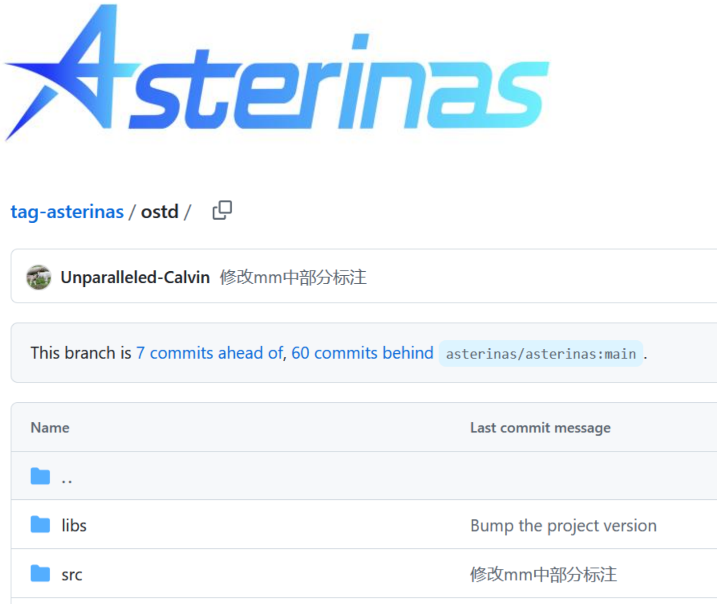
- **Ergonomics:**
  - ~~Extensive and repetitive textual descriptions across functions.~~ ✓
- **Precision:**
  - ~~Safety comments lack the precision of formally specified contracts.~~ ✓ ?

Is It Possible to Represent All Safety Constraints as Tags?

- Theoretically, safety constraints are Turing-complete.
- We are optimistic because:
  - There are numerous safety tags, as they can be defined by each crate.
  - Safety tags are abbreviations of safety comments.
    - If it is possible with safety comments, then it is also possible with tags.
    - Programs are not random or meaningless abstractions.
    - The challenge lies in the self-containment and precision when converting to contracts.

Experiments





Rust Standard Library

Almost complete

Coverage > 90%

Rust for Linux

Ongoing

Asterinas

Halfway done

Coverage ≈ 60%

Safety Tags for the Standard Library

Category	Safety Property	Meaning	Usage
Layout	Align(p, T)	p % alignment(T) = 0 && sizeof(T) % alignment(T) = 0	precondition
	Sized(T)	sizeof(T) = const, const ≥ 0	option
	ZST(T)	sizeof(T) = 0	precondition
	!Padding(T)	Padding(T) = 0	precondition
Pointer	!Null(p)	p != 0	precondition
	!Dangling(p)	allocator(p) != none	precond, hazard
	Allocated(p, T, len, A)	∀ i ∈ 0..sizeof(T) * len, allocator(p+i) = A	precondition
	InBound(p, T, len, arrage)	[p, p+(len+1)*sizeof(T)) ∈ arrage	precondition
	!Overlap(dst, src, len, T)	|dst-src| > sizeof(T) * len	precondition
Content	ValidInt(exp, vrange)	exp ∈ vrange	precondition
	ValidString(arange)	mem(arange) ∈ UTF-8	precond, hazard
	ValidCStr(p, len)	mem(p+len, p+len+1) = null	precondition
	Init(p, T, len)	∀i∈0..len, mem(p+i*sizeof(T), p+(i+1)*sizeof(T)) = validobj(T)	precond, hazard
	Unwrap(x, T, target)	unwrap(x) = target, target ∈ {Ok(T), Err, Some(T), None}	precondition
Aliasing	Owning(p)	ownership(*p) = none	precondition
	Alias(p1, p2)	p1 = p2	hazard
	Alive(p, l)	lifetime(*p) ≥ l	precondition
Misc	Pinned(p)	p = &*p	hazard
	!Volatile(p)	volatile(*p) = t, t ∈ {true, false}	precondition
	Opened(fd)	opened(fd) = true	precondition
	Trait(T, trait)	trait ∈ Trait(T), trait ∈ {Copy, Unpin, ...}	Option

“Annotating and Auditing the Safety Properties of Unsafe Rust”, Zihao Rao, et al, arXiv:2504.21312, 2025.

Outline

I. Safety Tags

II. A New Theory for Verification

III. Verification Practice with RAPx

The Theorem to Establish: Origin of Undefined Behavior (UB)

- UB originates exclusively from unsafe code.
  - Why?
- UB is solely determined by the safety constraints of that unsafe code.
  - What?
  - Why?

Safety Promise of Rust

"For Rust, this (soundness) means well-typed programs cannot cause Undefined Behavior. This promise only extends to safe code however; for unsafe code, it is up to the programmer to uphold this contract."

- **Safe code cannot cause undefined behavior.**
- **Only unsafe code may exhibit undefined behavior.**

https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#soundness-of-code--of-a-library

Soundness Criterion of Safe Rust

- A safe function f_s is sound iff

$$\forall P_{f_s}, P_{f_s} \nrightarrow UB$$

  - where P_{f_s} denotes any program that uses f_s and contains no unsafe code.
  - Proof: Assuming $\exists P_{f_s},\ P_{f_s} \rightarrow UB$, this contradicts the safety promise of Rust.

"We say that a library (or an individual function) is sound if it is impossible for safe code to cause Undefined Behavior using its public API."

https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#soundness-of-code--of-a-library

How to Define the Soundness Criterion of Unsafe Functions?

- The safe function should prevent all UB of the interior unsafe code.
  - Requirement: What are the sufficient conditions?
  - Although Rust compiler is unable to verify; this can be manually checked.

```
/// Safety Requirements (sufficient condition)
pub unsafe fn foo<T> (p: *const T) {...}

fn bar() {
    // Manually Check
    unsafe { foo(p) }
}
```

Observations from Existing Safety Comments

❑Pervasiveness:

- ➢Each unsafe function has a set of safety constraints to avoid undefined behavior.
- ➢These constraints are sufficient conditions.

❑Uniformity:

- ➢The safety constraints of each API are uniform across all call sites.

$$\forall f_{\boldsymbol{u}}, \exists \boldsymbol{SC}_{f_{\boldsymbol{u}}} \text{ s.t. } \forall \boldsymbol{P}_{f_u}, \boldsymbol{P}_{f_u} \vDash \boldsymbol{SC}_{f_{\boldsymbol{u}}} \Rightarrow \boldsymbol{P}_{f_u} \nrightarrow \boldsymbol{UB}$$

- $f_{\boldsymbol{u}}$ is an unsafe function
- $\boldsymbol{SC}_{f_{\boldsymbol{u}}}$ is the safety constraint of $f_{\boldsymbol{u}}$
- $\boldsymbol{P}_{f_u}$ is a program that uses $f_{\boldsymbol{u}}$ and contains no other unsafe code

Soundness Criterion of Unsafe Functions

- An unsafe function f_u with safety constraint SC_{f_u} is sound iff

$$\forall P_{f_u}, P_{f_u} \vDash SC_{f_u} \Rightarrow P_{f_u} \nrightarrow UB$$

  - where P_{f_u} denotes any program that uses f_u and contains no other unsafe code.

Theorem Proved: Origin of Undefined Behavior (UB)

- **UB originates exclusively from unsafe code.**
  - True: Otherwise, it contradicts the safety promise.
- **UB is solely determined by the safety constraints of that unsafe code.**
  - Assume an unsafe function f_u with safety constraint SC_{f_u} is sound.
    - $\forall P_{f_u}, P_{f_u} \vDash SC_{f_u} \Rightarrow P_{f_u} \nrightarrow UB$
  - Assume a program uses f_u and satisfies SC_{f_u}; but the program leads to UB.
    - $\exists P_{f_u}, \text{s.t.}\, P_{f_u} \vDash SC_{f_u} \wedge P_{f_u} \rightarrow UB$
  - This leads to a contradiction.

Sound Function Encapsulation

- A safe function f_s is sound iff
  - It contains no unsafe code, or
  - $\forall f_u \in \text{UnsafeCallee}(f_s),\ f_s \vDash SC_{f_u}$
- An unsafe function f_u is sound iff
  - $\forall f'_u \in \text{UnsafeCallee}(f_u), SC_{f_u} \wedge f_u \vDash SC_{f'_u}$
- We can unify them by treating a safe function as with empty constraint.
  - $SC_{f_s} = \emptyset$
- A function f is sound iff
  - $\forall f_u \in \text{UnsafeCallee}(f), SC_f \wedge f \vDash SC_{f_u}$

Can we extend the result to structs?

Structs are More Complicated

- Static methods (without &self parameter) are the same as functions.
  - They can be called directly.
- Dynamic methods (with self/&self parameter) can only be executed after a constructor.
  - The constructor may help the method to satisfy some safety constraint.
- Method safety declarations are more flexible.
- Methods with a mutable self parameter could affect other methods.
  - When evaluating the soundness of a method, we should consider all possible method invocations before invoking the method.
  - Such vulnerable fields can be marked as unsafe now.

Example Struct

```
struct Foo<'a> {
    ptr: *mut u8,
    len: usize
}
impl Foo {
    pub fn from(p: *mut u8, l: usize) -> Foo {
        Foo { ptr: p, len: l }
    }
    pub fn unsafe get(&self) -> &[u8] {
        slice::from_raw_parts(self.ptr, self.len)
    }
    pub unsafe fn set_len(&mut self, l: usize) {
        self.len = l;
    }
}
```

Safe Constructor

Unsafe Method

Unsafe Method without interior unsafe code

Alternative Way of Defining the Struct

```
struct Foo {
    ptr: *mut u8,
    len: usize
}
impl Foo {
    pub fn unsafe from(p: *mut u8, l: usize) -> Foo {
        Foo { ptr: p, len: l }
    }
    pub fn get(&self) -> &[u8] {
        unsafe { slice::from_raw_parts(self.ptr, self.len) }
    }
    pub unsafe fn set_len(&mut self, l: usize) {
        self.len = l;
    }
}
```

Unsafe Constructor

Safe Method

Soundness Criteria of a Struct

- A struct $\boldsymbol{S} = \{\boldsymbol{C}, \boldsymbol{F}, \boldsymbol{M}, \boldsymbol{d}\}$ is sound only iff
  - $\forall \boldsymbol{f} \in \{\boldsymbol{C}, \boldsymbol{F}\}, \forall \boldsymbol{P_f},\ \boldsymbol{P_f} \vDash \boldsymbol{SC_f} \Rightarrow \boldsymbol{P_f} \nrightarrow \boldsymbol{UB}$
  - $\forall \boldsymbol{c} \in \boldsymbol{C}, \boldsymbol{m} \in \{\boldsymbol{M}, \mathbf{d}\}, \forall \boldsymbol{P_{c,m}},\ \boldsymbol{P_{c,m}} \vDash \boldsymbol{SC_c} \wedge \boldsymbol{SC_m} \Rightarrow \boldsymbol{P_{c,m}} \nrightarrow \boldsymbol{UB}$

- $\boldsymbol{C}$ is the set of constructors of the struct (including the literal constructor)
- $\boldsymbol{F}$ is the set of static methods of the struct
- $\boldsymbol{M}$ is the set of dynamic methods of the struct (including literal field assignments)
- $\boldsymbol{d}$ is the destructor of the struct
- $\boldsymbol{P_f}$ is a program that uses f and contains no other unsafe code
- $\boldsymbol{P_{c,m}}$ is a program that uses c and m, and contains no other unsafe code
- $\boldsymbol{SC_f}$ is the safety constraint of f; If f is safe, $SC_f = \emptyset$

Sound Struct Encapsulation

- A struct $\boldsymbol{S} = \{\boldsymbol{C}, \boldsymbol{F}, \boldsymbol{M}, \boldsymbol{d}\}$ is sound iff:
  - All static method encapsulations are sound:

$$\forall \boldsymbol{f} \in \{\boldsymbol{C}, \boldsymbol{F}\}, \boldsymbol{f_u} \in \mathbf{UnsafeCallee}(\boldsymbol{f}), \boldsymbol{SC_f} \wedge \boldsymbol{f} \vDash \boldsymbol{SC_{f_u}}$$

  - All dynamic methods encapsulations are sound:

$$\forall \boldsymbol{m} \in \{\boldsymbol{M}, \mathbf{d}\}, \boldsymbol{f_u} \in \mathbf{UnsafeCallee}(\boldsymbol{m}), \underset{?}{\boldsymbol{I}} \wedge \underset{?}{\boldsymbol{BI}} \wedge \boldsymbol{SC_m} \wedge \boldsymbol{m} \vDash \boldsymbol{SC_{f_u}}$$

Invariants of Struct

- The properties that all constructors can ensure.
  - $\forall c \in C, c \wedge SC_c \vDash I$
- Minimal invariants for a struct instance in Rust:
  - $I \supset \{Allocated, Align, Init\}$
  - Otherwise, a well-typed Rust program with the objects may cause UB.
  - Not include the objects pointed by raw pointers.

Broken Invariant

- A struct may contain some methods that break the safety invariants.
  - Typically, via the method parameter &mut self or mut self.
  - The broken invariants are denoted as BI.
- Each struct may have one or several such disruptive methods.
  - $BI = BI_{m_1} \cup \cdots \cup BI_{m_n}$

Outline

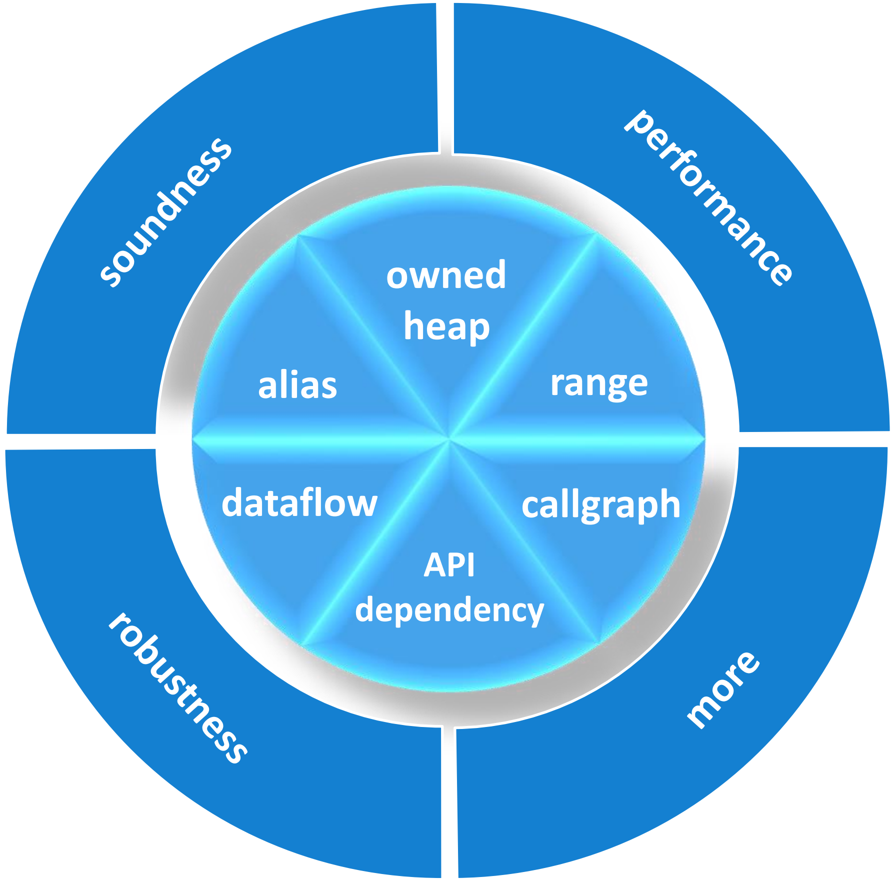
I. Safety Tags

II. A New Theory for Verification

III. Verification Practice with RAPx

RAPx: A static analysis platform for Rust

- ❑Separate fundamental analysis tasks from upper-level applications.
- ❑Our verification is based on these core analysis modules.



Project URL: https://github.com/Artisan-Lab/RAPx

Verification with RAPx

Step 1: Annotate Unsafe Functions with Safety Tags

Already presented

Step 2: Extract Audit Units from the Target Crate

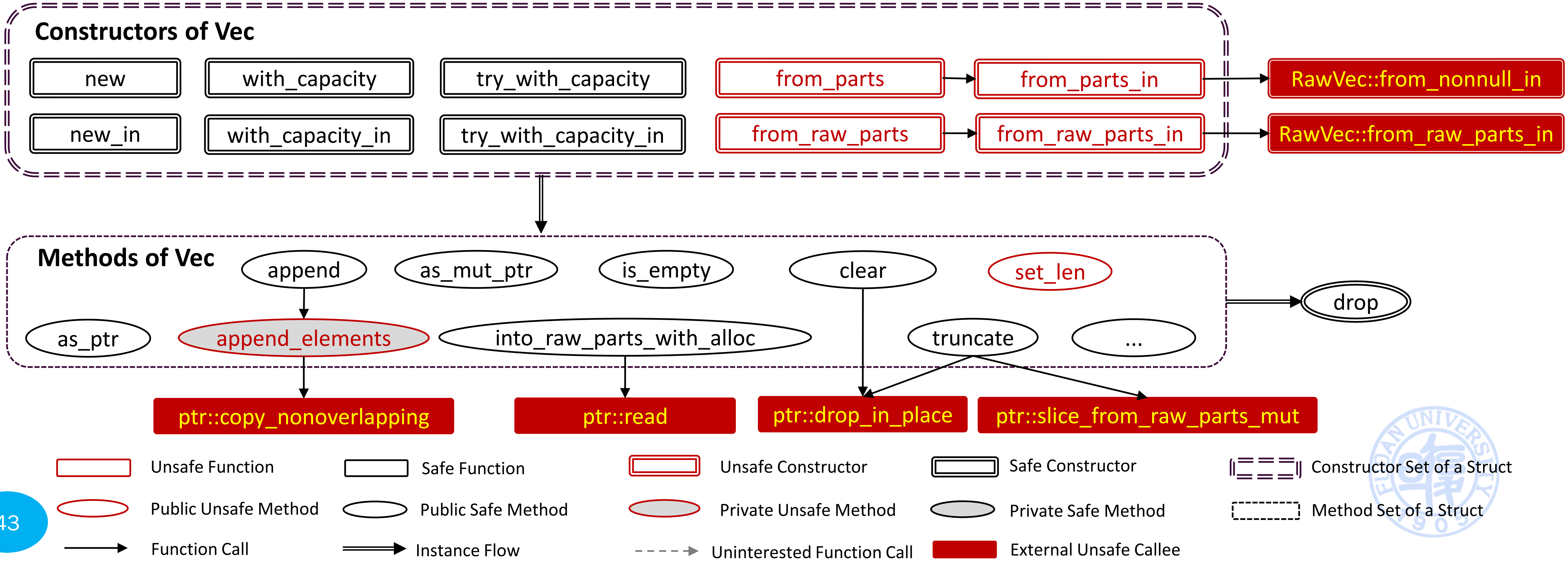
Step 3: Verify the Soundness of Each Audit Unit

Each audit unit has a set of soundness requirements to be verified.
If all these requirements are satisfied, the soundness of the crate is verified.

Model Unsafety Propagations with Graph (UPG)

- UPG considers both function calls and instance flows.
- UPG does not consider function calls with a safe callee.

Example: UPG of Vec



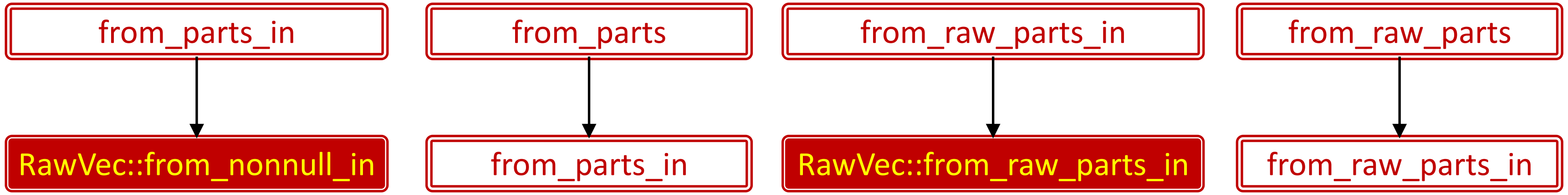
Extract Audit Units: Case 1

- All (external) dependent unsafe functions or methods f_u.
  - They should be annotated with safety constraints.
  - Least requirement: $SC_{f_u} \neq \emptyset$
  - We should assume the safety constraints are sufficient, *i.e.*, f_u is sound.

ptr::read	ptr::copy_nonoverlapping	RawVec::from_raw_parts_in
ptr::drop_in_place	ptr::slice_from_raw_parts_mut	RawVec::from_nonnull_in

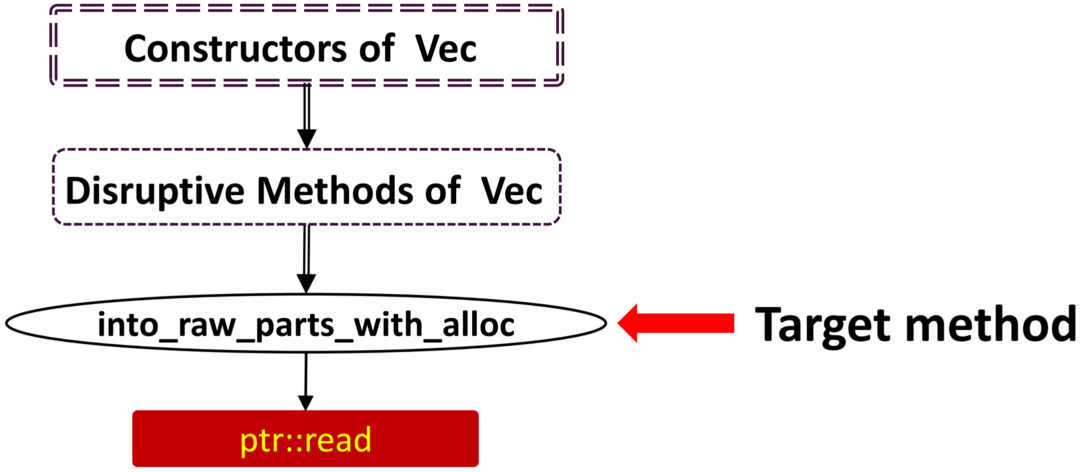
Extract Audit Units: Case 2

- ❑The caller f is a function, and contains unsafe callees.
  - ➢**Soundness Requirement: Function encapsulation.**
  - ➢$\forall f_u \in \mathbf{UnsafeCallee}(f), SC_f \wedge f \vDash SC_{f_u}$
  - ➢It can be used for both verification and safety tag consistency check.



Extract Audit Units: Case 3

- A method m with unsafe callees.
  - **Soundness Requirement: Method encapsulation.**
  - $\forall c \in C, f_u \in \text{UnsafeCallee}(m),\ I \wedge BI \wedge SC_m \wedge m \vDash SC_{f_u}$



Extract Audit Units: Case 3.1

- ❑ **If the struct has extra invariant specified by developers.**
  - ➢ **Soundness Requirement: Constructor encapsulation.**
  - ➢ $\forall c \in C, c \wedge SC_c \vDash I$

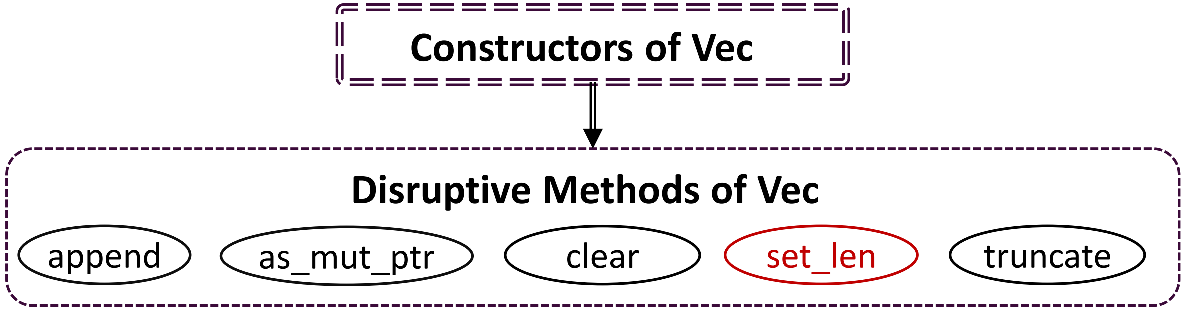
Constructors of Vec

new	with_capacity	try_with_capacity	from_parts	from_raw_parts
new_in	with_capacity_in	try_with_capacity_in	from_parts_in	from_raw_parts_in

```
#[rapx::invariant(len <= buf.cap, Init(buf, T, len))]
pub struct Vec<T, A: Allocator = Global> {
    buf: RawVec<T, A>,
    len: usize,
}
```

Extract Audit Units: Case 3.2

- Analyze breaking invariant.
  - $BI = BI_{m_1} \cup \cdots \cup BI_{m_n}$
  - If the safety constraint of a method ensures the invariant, then the invariant remains preserved.



Prove the Soundness of Audit Units

- A struct is sound iff:
  - All static method encapsulations are sound;
    - Satisfied by case 2.
  - All dynamic method encapsulations are sound.
    - Satisfied by case 3.

Verification with RAPx

Step 1: Annotate Unsafe Functions with Safety Tags — Already presented

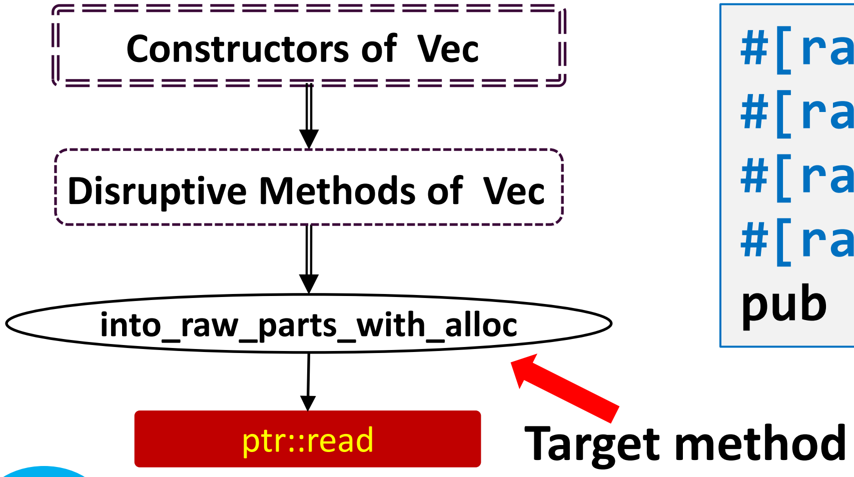
Step 2: Extract Audit Units from the Target Crate

Step 3: Verify the Soundness of Each Audit Unit — Under development

Project URL: https://github.com/Artisan-Lab/RAPx

Verification Target

```rust
pub fn into_raw_parts_with_alloc(self) -> (*mut T, usize, usize, A) {
    let mut me = ManuallyDrop::new(self);
    let len = me.len();
    let capacity = me.capacity();
    let ptr = me.as_mut_ptr();
    let alloc = unsafe { ptr::read(me.allocator()) };
    (ptr, len, capacity, alloc)
}
```

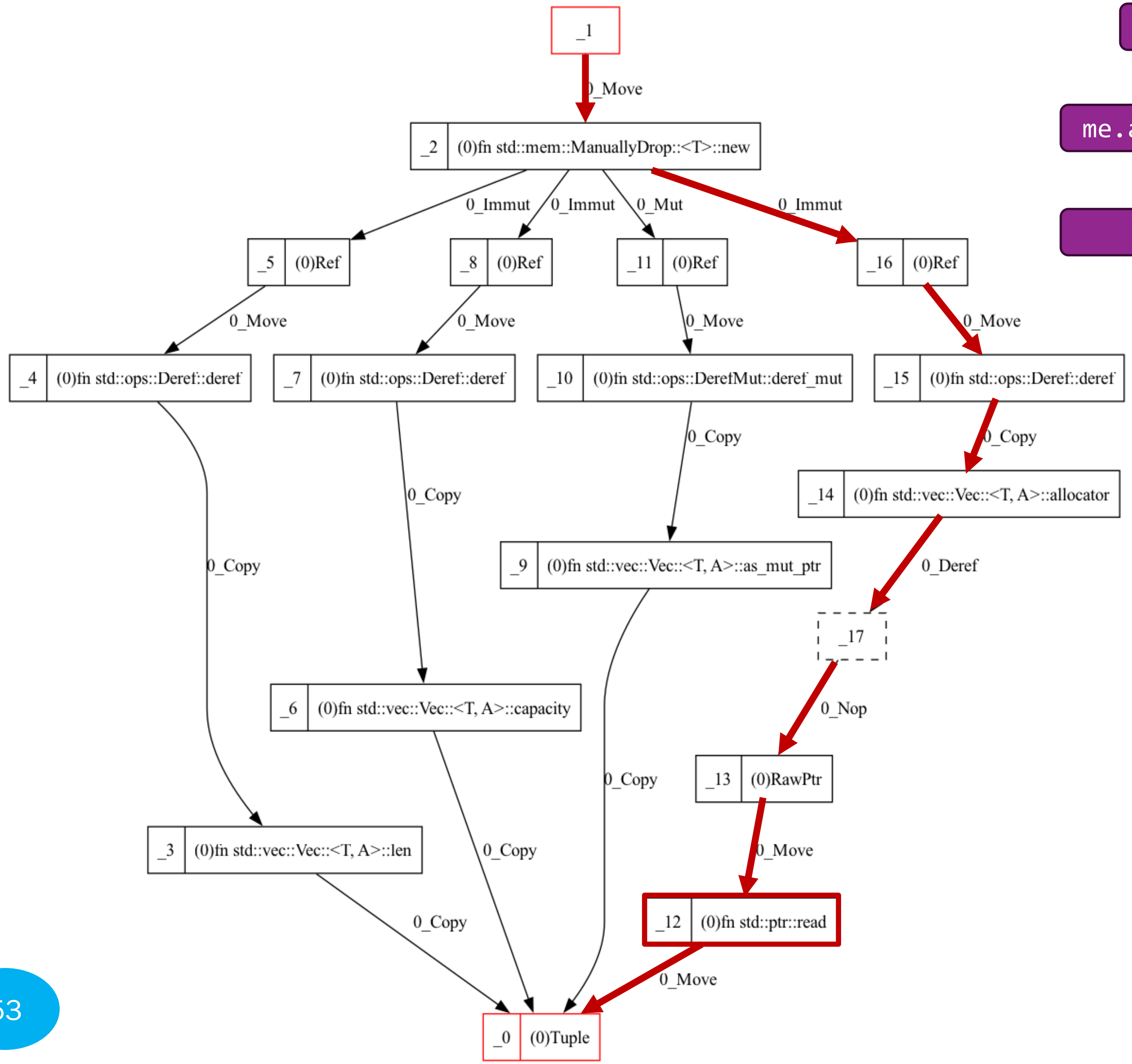


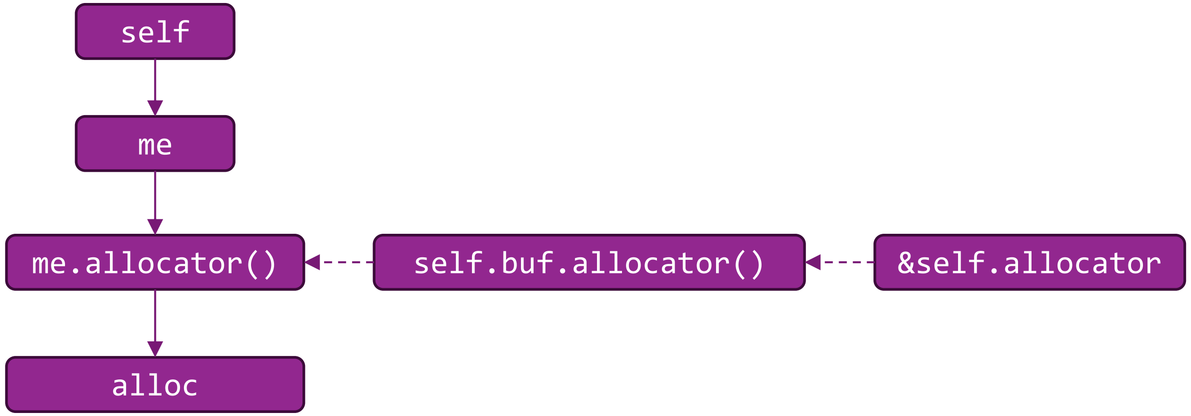

```rust
#[rapx::ValidPtr(src, T, 1)]
#[rapx::Align(src, T)]
#[rapx::Init(src, T, 1)]
#[rapx::Alias(src, ret, Trait(T, Copy))]
pub const unsafe fn read<T>(src: *const T) -> T
```

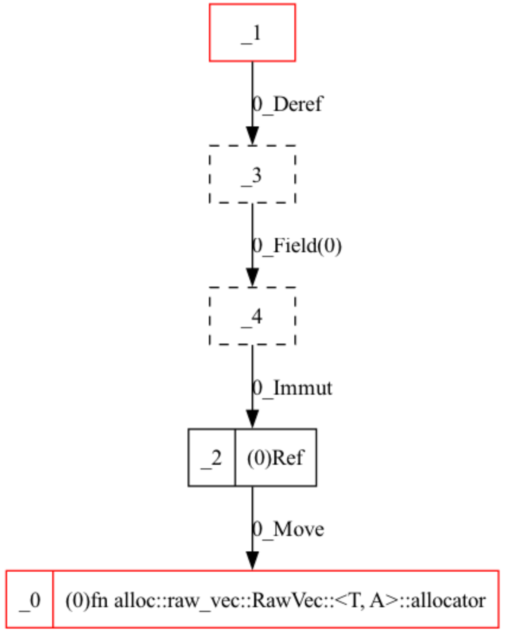
Verification based on MIR

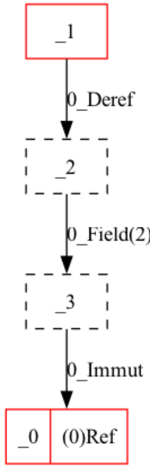
```
_2 = std::mem::ManuallyDrop::<std::vec::Vec<T, A>>::new(move _1)
_5 = &_2
_4 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::Deref>::deref(move _5)
_3 = std::vec::Vec::<T, A>::len(copy _4)
_8 = &_2
_7 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::Deref>::deref(move _8)
_6 = std::vec::Vec::<T, A>::capacity(copy _7)
_11 = &mut _2
_10 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::DerefMut>::deref_mut(move _11)
_9 = std::vec::Vec::<T, A>::as_mut_ptr(copy _10)
_16 = &_2
_15 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::Deref>::deref(move _16)
_14 = std::vec::Vec::<T, A>::allocator(copy _15)
_13 = &raw const (*_14)
_12 = std::ptr::read::<A>(move _13)
_0 = (copy _9, copy _3, copy _6, move _12)
```

Dataflow Graph









Verification: ValidPtr, Align, Init

- Tracing dataflow backward until the constraints can be satisfied.
- Perform forward analysis to check whether any property is invalidated.

```
…
_14:  @ &'{erased} A/#1
_15:  @ &'{erased} std::vec::Vec<T/#0, A/#1>
_16:  @ &'{erased} std::mem::ManuallyDrop<std::vec::Vec<T/#0, A/#1>>

...
_15 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::Deref>::deref(move _16)
_14 = std::vec::Vec::<T, A>::allocator(copy _15)
_13 = &raw const (*_14)
_12 = std::ptr::read::<A>(move _13)
_0 = (copy _9, copy _3, copy _6, move _12)
```

We assume external types are sound:

$$\frac{\Gamma \vdash \ x: \mathrm{Ref}}{x \vDash \{\mathrm{ValidPtr}, \mathrm{Align}, \mathrm{Init}\}} \qquad \frac{\Gamma \vdash x \vDash \{\mathrm{ValidPtr}, \mathrm{Align}, \mathrm{Init}\},\ \ \Gamma \vdash y = \mathrm{addrof}(\mathrm{deref}(x))}{y \vDash \{\mathrm{ValidPtr}, \mathrm{Align}, \mathrm{Init}\}}$$

Verification: Alias

- **Forward analysis to check if hazard is eliminated.**
  - **No mutation is performed through shared references.**
  - **No shared mutable references exist after the function returns.**
- **Based on alias analysis results (over approximation based on MoP).**
  - **Aliases of _12 = {_0,_1,_2,_4,_5,_7,_8,_10,_11,_13,_14,_15,_16}.**

```
_15 = <std::mem::ManuallyDrop<std::vec::Vec<T, A>> as std::ops::Deref>::deref(move _16)
_14 = std::vec::Vec::<T, A>::allocator(copy _15)
_13 = &raw const (*_14)
_12 = std::ptr::read::<A>(move _13)
_0 = (copy _9, copy _3, copy _6, move _12)
```

Rules for shared-reference check as typeof(12) is not Copy:

- The return value should not aggregate multiple references related to 12.
- As the return value contains the object being read, the corresponding function parameter should not be a raw pointer or reference to the object.

Discussion on the Soundness of Verification

- The approach of audit unit extraction should be sound.
- When verifying each audit unit, the soundness depends on two aspects:
  - The abstraction interpretation, which is currently under development.
    - Rigorous design and formal proofs are needed.
  - The underlying analysis modules, such as alias analysis and dataflow analysis.
    - The implementations are biased toward over-approximation.
    - We are designing test suites to better evaluate the soundness of each module.
    - These modules are being continuously refined.


Thanks!
Q&A