Rust安全机制和特权代码安全强化

Demystifying and Enhancing Rust Security: A Privilege-Code Perspective

徐辉
复旦大学 计算机科学技术学院

9/17/2023

Outline

I. Demystifying the Security Mechanism of Rust

II. Unsafe Code Characterization and Usage Mitigation

III. Detecting Bugs Incurred by Privileged Code

I. Demystifying the Security Mechanism of Rust

1) Trends of Software Security via Language Design

1) Trends of Software Security via Language Design
2) Security Mechanisms of Rust: Philosophy and Implementation
3) An Empirical Study of the Rust Ecosystem

Most Dangerous Software Vulnerabilities (by MITRE, 2023)

Rank	ID	Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-416	Use After Free
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	CWE-20	Improper Input Validation
7	CWE-125	Out-of-bounds Read
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-862	Missing Authorization
12	CWE-476	NULL Pointer Dereference
13	CWE-287	Improper Authentication
14	CWE-190	Integer Overflow or Wraparound
15	CWE-502	Deserialization of Untrusted Data
16	CWE-77	Impro
17	CWE-119	Impro
18	CWE-798	Use of Hard-coded Credentials
19	CWE-918	Serve
20	CWE-306	Missing Authentication for Critical Function
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
22	CWE-269	Improper Privilege Management
23	CWE-94	Improper Control of Generation of Code ('Code Injection')
24	CWE-863	Incorrect Authorization
25	CWE-276	Incorrect Default Permissions
36	CWE-401	Missing Release of Memory after Effective Lifetime

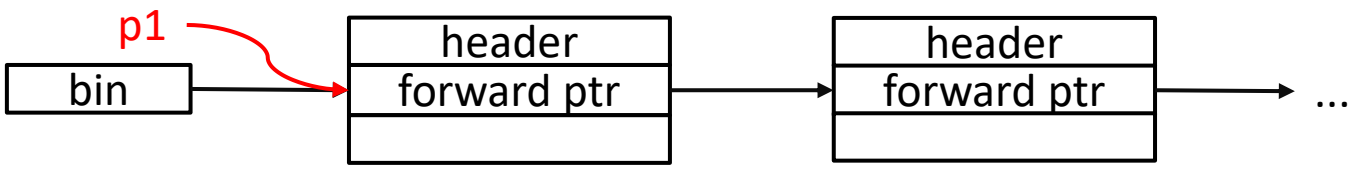
Which vulnerabilities can be mitigated through language design?

Application independent ones, *e.g.*, memory-safety bugs

Source: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

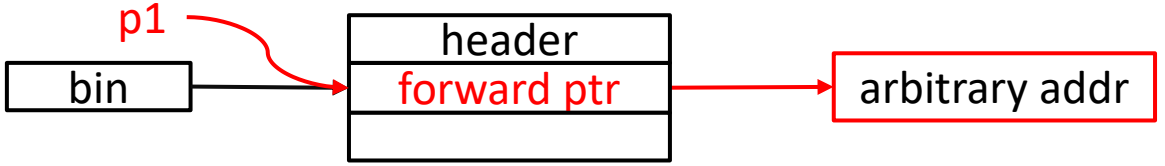
Why Memory Bugs are Dangerous? UAF as an Example

1. free(p1)



Add the block of *p1 to the free list

2. write(p1)

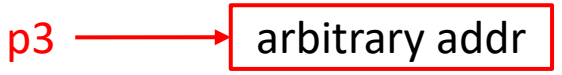


Attackers modify the forward pointer through p1

3. p2 = malloc()

Remove the block from the list; now bin points to the arbitrary address

4. p3 = malloc()

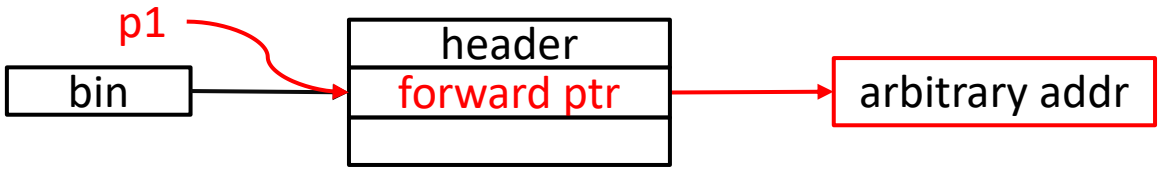


p3 points to the arbitrary address

Detecting UAF is Hard: via Allocator Design

- **Option 1: prevent writing the dangling pointer p1**

2. write(p1)



Problem: incur overhead during each pointer access

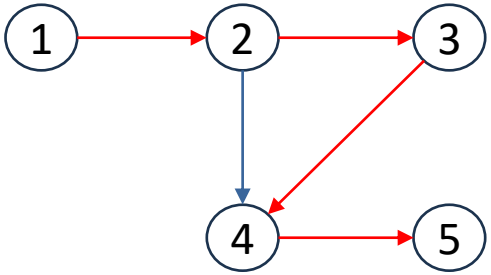
- **Option 2: prevent adding invalid blocks to the free list**

3. p2 = malloc()

How to design an efficient and robust mechanism?

Detecting UAF is Hard: via Static Analysis

- Alias analysis is NP-Hard
  - Hamiltonian path problems => Flow-insensitive may-alias analysis



Hamiltonian Path Problem

```
v4 = &v5
v2 = &v4
v3 = &v4
v2 = &v3
v1 = &v2
```

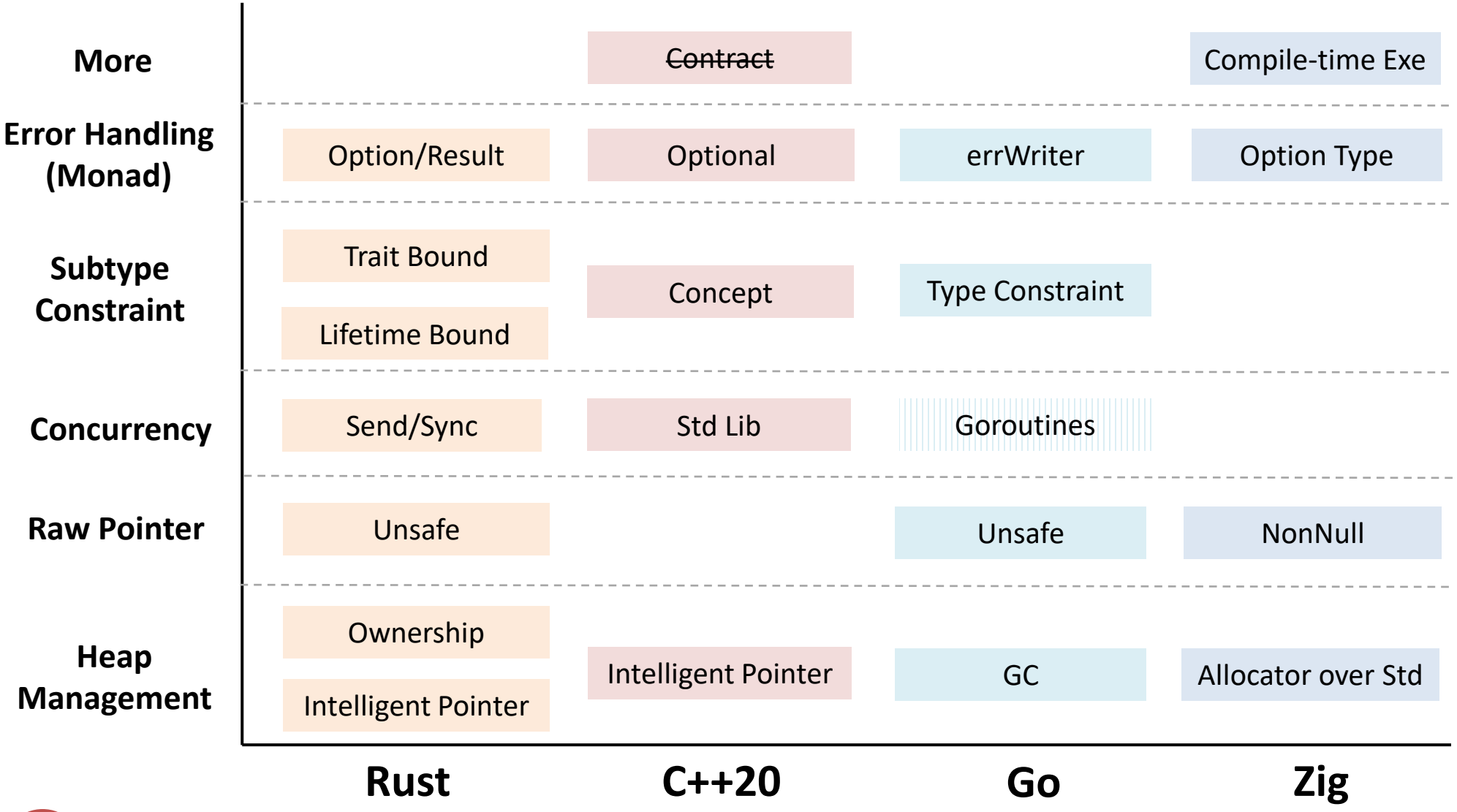
****v1 = v5 ?

May-Alias Analysis

  - More complicated alias analysis problems：
    - Flow-sensitive, path-sensitive, control-sensitive, context-sensitive…
    - Raw pointer, point-to
    - Concurrent code

[Horwitz'97] Susan Horwitz, Precise Flow-Insensitive May-Alias Analysis is NP-Hard, 1997

(Security) Features of System PL: Rust/C++/Go/Zig



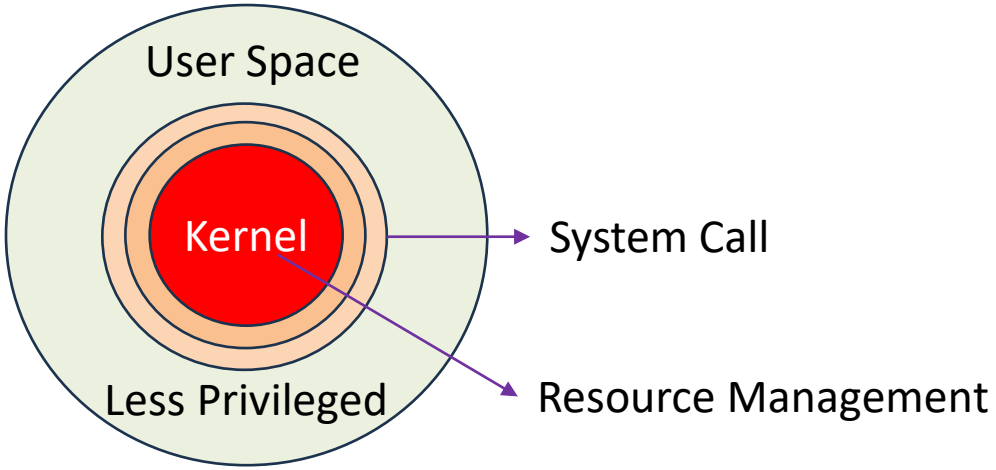
This features listed may not be complete

2) Security Mechanisms of Rust: Philosophy and Impl.

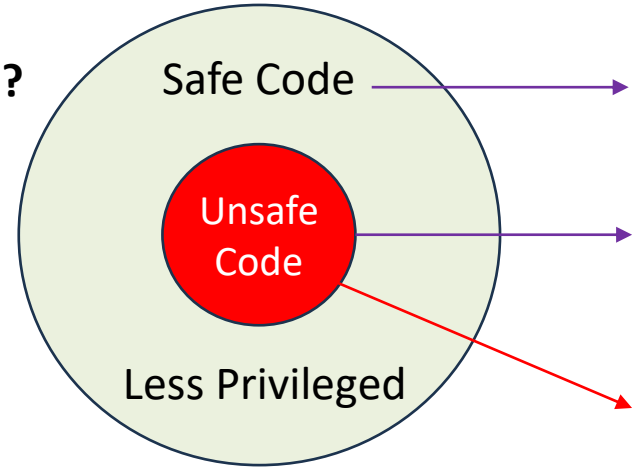
1) Trends of Software Security via Language Design
2) Security Mechanisms of Rust: Philosophy and Implementation
3) An Empirical Study of the Rust Ecosystem

Idea of Rust for Security

Operating System:
Ring Model



Rust:
Code Privilege Model?



Security objective of Rust:

- Type safety (nothing special)
- No heap bugs (auto)
- No other undefined behaviors

Interior unsafe: encapsulate privileged code within safe APIs

May break the security protocol

Auto Heap Management: Rust Tackles the Problem via Ownership

- Each object is owned by one variable
- Ownership can be moved or borrowed
  - Mode of borrowing: immutable/mutable

```
let mut alice = Box::new(1);
let bob = alice;
println!("alice:{}", alice);
```

alice owns the Box object

move the ownership from alice to bob

```
let mut alice = Box::new(1);
let bob = &mut alice;
**bob = **bob + 1;
println!("alice:{}", alice);
```

bob borrows the ownership

bob is dead; return the ownership

- **Exclusive mutability principle**: an object cannot be mutable and shared at one program point. How?

Lifetime Inference with a Constraint-based Method

- Infer the minimum lifetime of each reference
  - not based on lexical scopes or blocks

```
let mut a: i32 = 1;
let mut b: i32 = 2;
let mut p: & T = &a;
// program point 1
if condition {
    // program point 2
    print(*p);
    // program point 3
    p = &b;
    // program point 4
}
// program point 5
print(*p);
// program point 6
```

p is alive.

p is alive.

p is dead, return the ownership to a

p is reassigned, alive

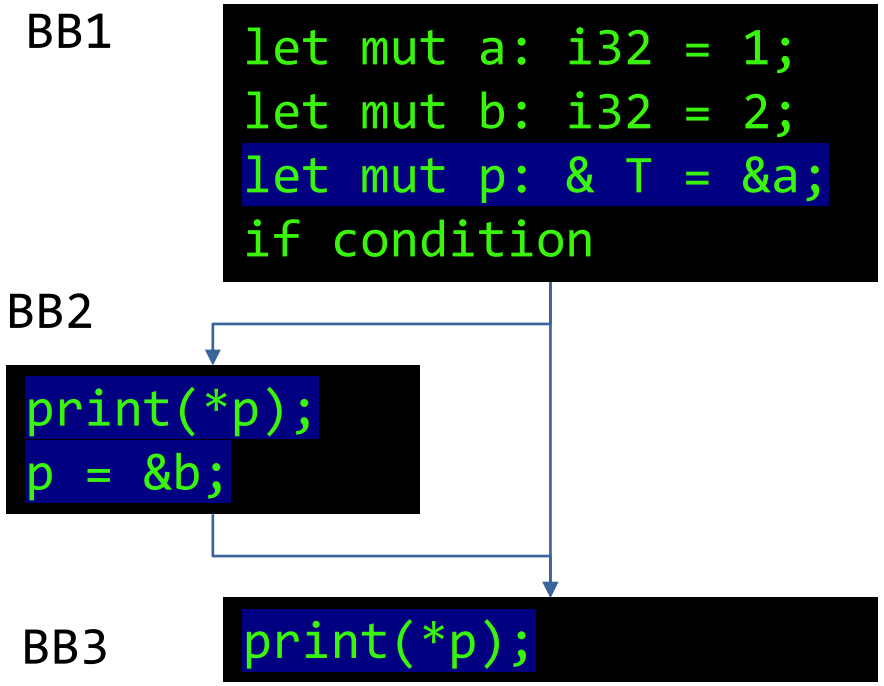
p is alive

p is dead

https://rust-lang.github.io/rfcs/2094-nll.html

Constraint Extraction: Liveness

- Forward def-use analysis
- (L: {P}) @ P denotes lifetime L is alive at the point P
- Traditional liveness analysis: backward data-flow analysis

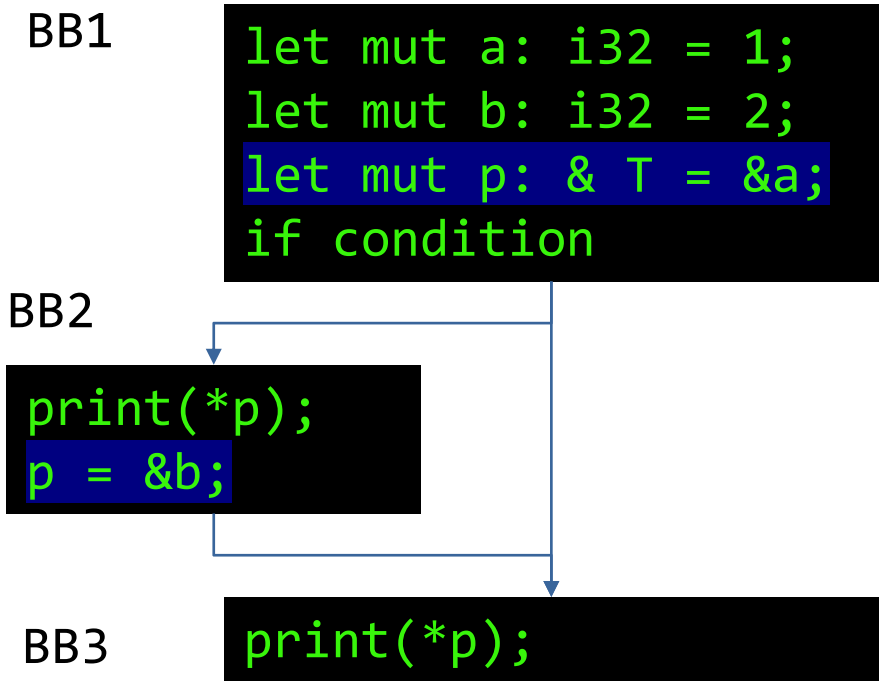



```
Def: ('pa: {BB1/3}) @ BB1/3

Use: ('pa: {BB2/0}) @ BB2/0

Def: ('pb: {BB2/2}) @ BB2/2

Use: (phi('pa,'pb): {BB3/0}) @ BB3/0
```

```
'pa = {BB1/3, BB2/0, BB3/0}
'pb = {BB2/2, BB3/0}
```

Constraint Extraction: Subtyping

- The lifetime of each reference should not exceed its referent
- (L1: L2) @ P means lifetime L1 outlives lifetime L2 at point P



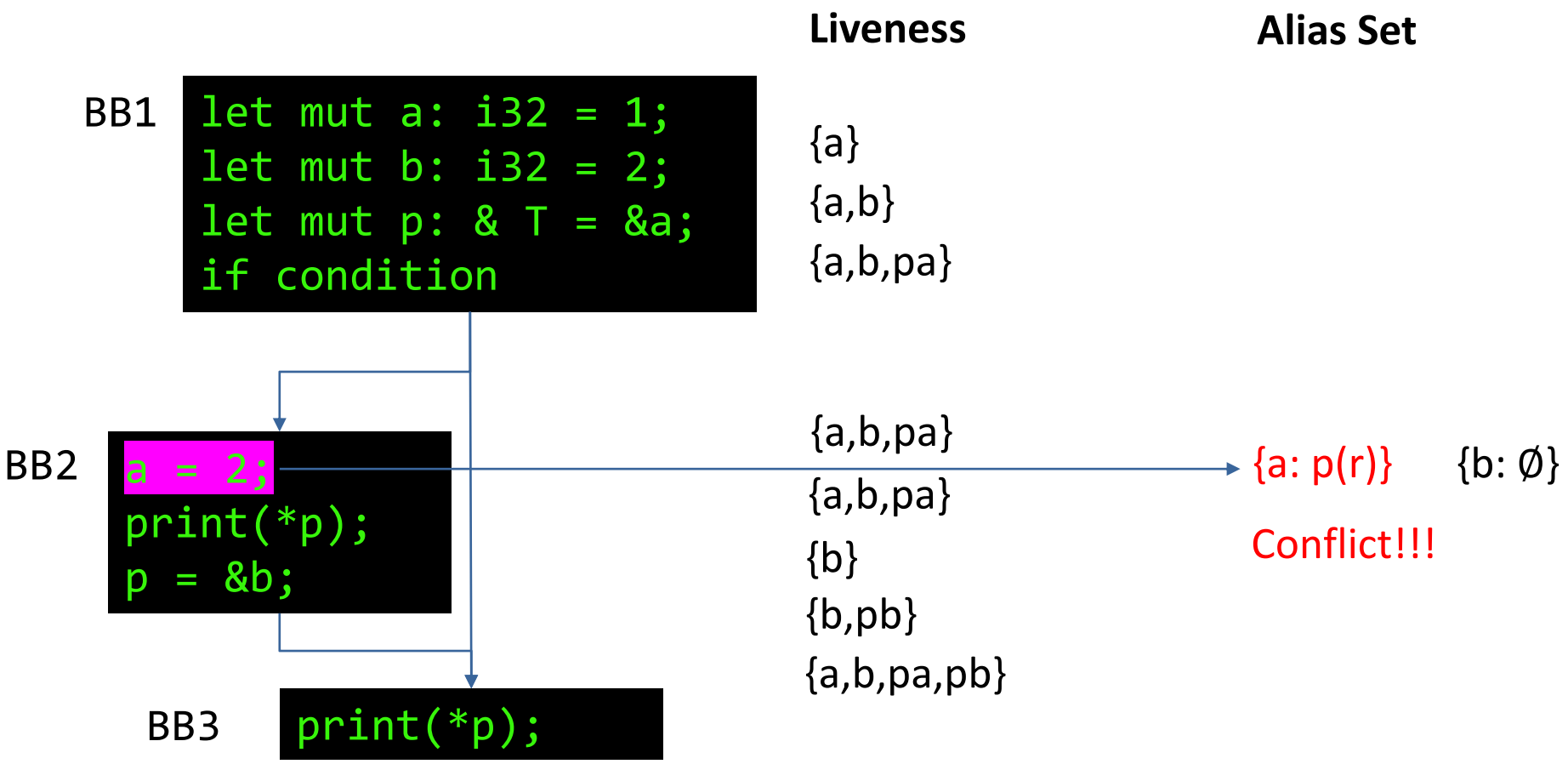
('a: 'pa) @ BB1/3
Def: ('pa: {BB1/3}) @ BB1/3

Use: ('pa: {BB2/0}) @ BB2/0
('b: 'pb) @ BB2/2
Def: ('pb: {BB2/2}) @ BB2/2

Use: (phi('pa,'pb): {BB3/0}) @ BB3/0

'pa = {BB1/3, BB2/0, BB3/0}
'a = {BB1/1, BB1/2, BB1/3, BB2/0, BB3/0}
'pb = {BB2/2, BB3/0}
'b = {BB1/2, BB1/3, BB2/0, BB2/1, BB2/2, BB3/0}

Detecting Shared Mutable Aliases



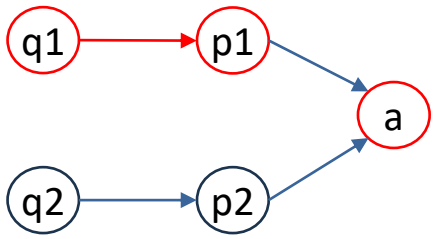
Why the Approach is Efficient?

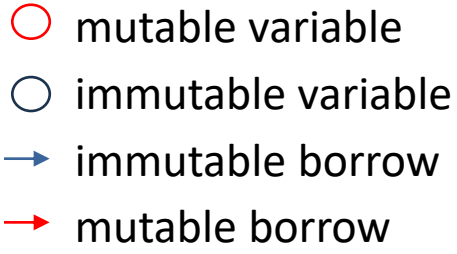
- Avoid hard alias-analysis problems
  - No need to track multi-level aliases
  - The mutability does not propagate

```
let mut a = 1;
let mut p1 = &a;
let p2 = &a;
let mut q1 = &mut p1;
let q2 = &p2;
```

p1 is mutable; immutable borrow; *p1 is read only

q1 is mutable; mutable borrow; **q1 is read only





Limitations of Ownership

- We may need both shared & mutable, *e.g.,* double-linked list


```
struct Node {
    val: u64,
    prev: Option<Rc<RefCell<List>>>,
    next: Option<Rc<RefCell<List>>>,
}
```

Option 1: Shared Pointer
(with runtime cost)

```
struct Node {
    val: u64,
    next: *mut List,
    prev: *mut List,
}
```

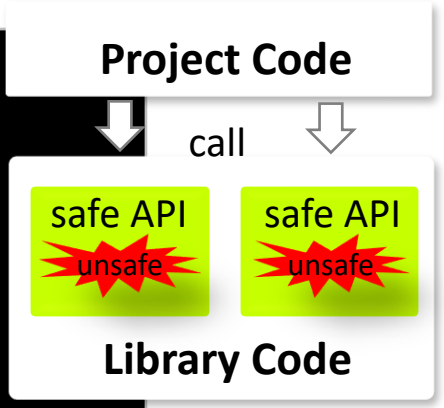
Option 2: Raw Pointer
(unsafe code, bypass borrow check)

- Ownership also requires RAII because dropping unit data is bad
  - Use unsafe code to create uninitialized object

Interior Unsafe

- Encapsulate unsafe code within safe APIs
- Prevent developers from directly using unsafe code

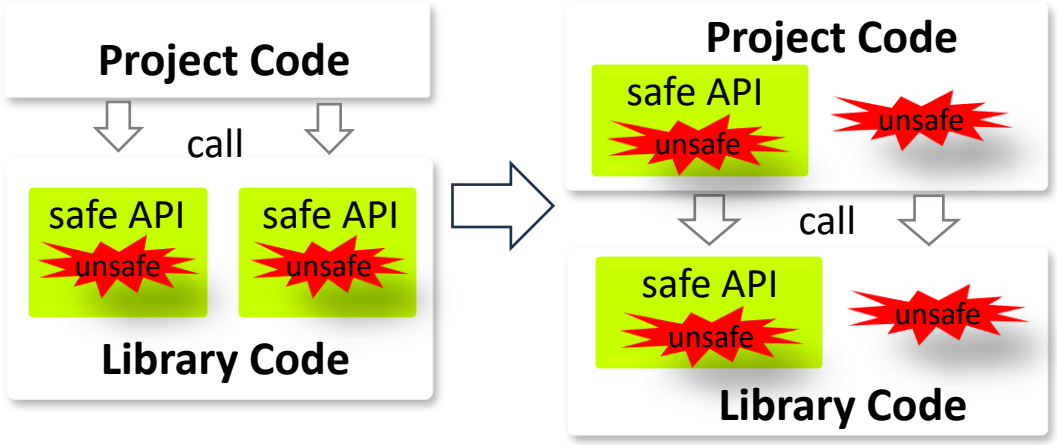
```
impl<T> Vec<T> {
    //safe API encapsulation
    pub fn push(&mut self, value: T) {
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
}
```



Code of Vec from Rust std-lib

Limitation: Security vs Control + Productivity

- Control: need privileged code to implement particular features
  - e.g., low-level code, double-linked list, concurrent code
  - API soundness with interior unsafe code cannot be verified by compiler
- Productivity: advanced language features
  - Generic, trait: increase the difficult of verification



Ideal Case

Real Case

```
impl<T> Vec<T:Bound?> {
    pub fn push(&mut self,
                value: T) {
        ...
    }
}
```

Generic Parameters with Trait Bound

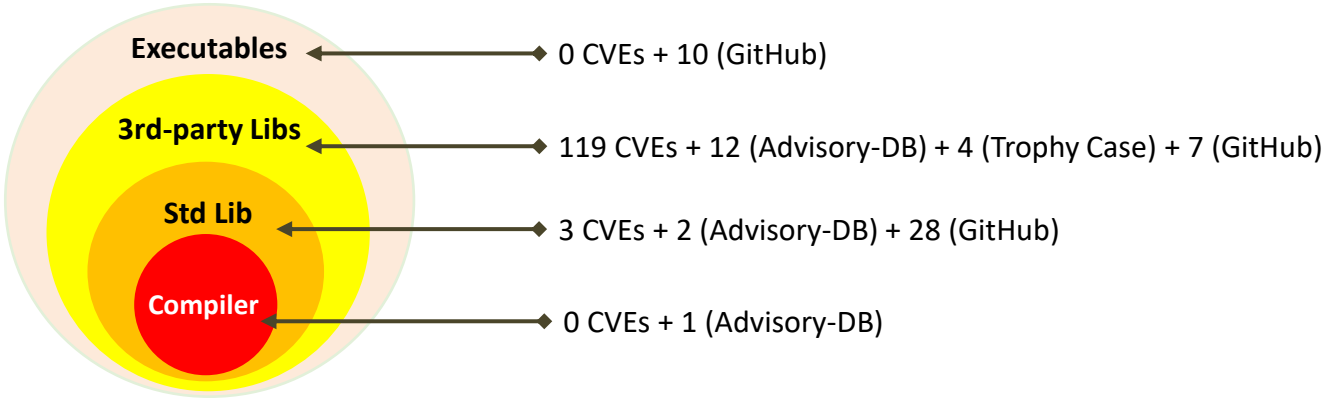
3) An Empirical Study of the Rust Ecosystem

1) Trends of Software Security via Language Design
2) Security Mechanisms of Rust: Philosophy and Implementation
3) An Empirical Study of the Rust Ecosystem

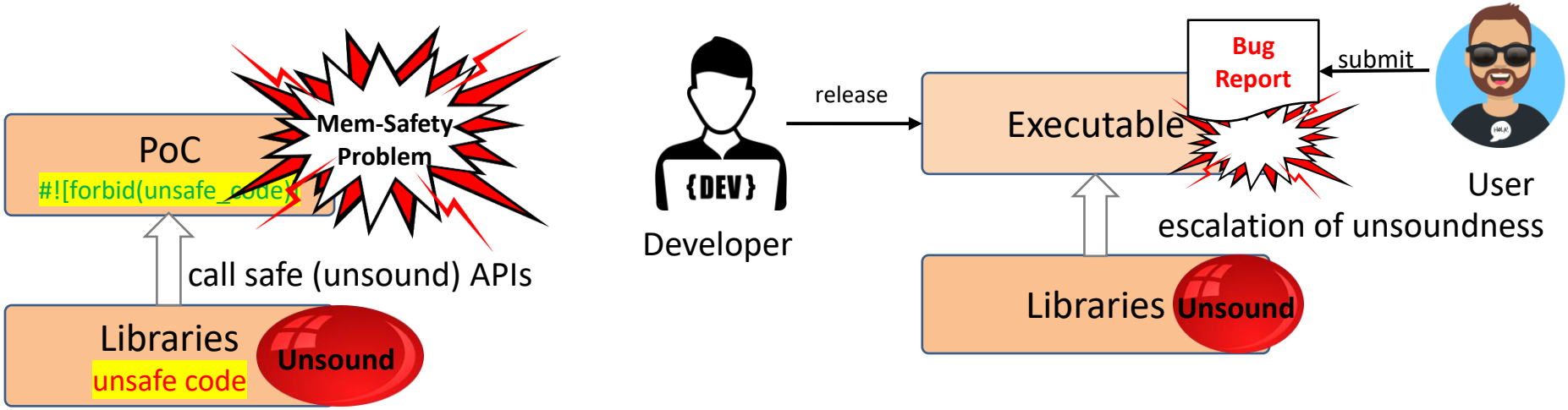
[Xu'21] Hui Xu, *et al.* "Memory-safety challenge considered solved? An in-depth study with all Rust CVEs", *TOSEM*, 2021.

Overview

- A survey of 185 memory-safety bugs before 2020-12-31



- All these bugs need unsafe code, except one compiler bug
- Most CVEs are API soundness issues



Some Bugs are Unique for Rust

- 35 bugs are automatic memory reclaim issues
- 81 bugs are unsoundness issues of APIs
  - 29/81 simple issues
  - 52/81 related to generics and traits

Culprit		Consequence					Total
		Buf. Over-R/W	Use-After-Free	Double Free	Uninit Mem	Other UB	
Auto Memory Reclaim	Bad Drop at Normal Block	0 + 0 + 0	1 + 9 + 6	0 + 2 + 1	0 + 2 + 0	0 + 1 + 0	22
	Bad Drop at Cleanup Block	0 + 0 + 0	0 + 0 + 0	1 + 7 + 0	0 + 5 + 0	0 + 0 + 0	13
Unsound Function	Bad Func. Signature	0 + 2 + 0	1 + 5 + 2	0 + 0 + 0	0 + 0 + 0	1 + 2 + 4	17
	Unsoundness by FFI	0 + 2 + 0	5 + 1 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	12
Unsound Generic or Trait	Insuff. Bound of Generic	0 + 0 + 1	0 + 33 + 2	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	36
	Generic Vul. to Spec. Type	3 + 0 + 1	1 + 0 + 0	0 + 0 + 0	1 + 0 + 1	1 + 2 + 0	10
	Unsound Trait	1 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 2 + 0	6
Other Errors	Arithmetic Overflow	3 + 1 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	5
	Boundary Check	1 + 9 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 0 + 0	12
	No Spec. Case Handling	2 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	2 + 1 + 1	9
	Exception Handling Issue	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	4
	Wrong API/Args Usage	0 + 3 + 0	1 + 4 + 0	0 + 0 + 0	0 + 1 + 1	0 + 5 + 2	17
	Other Logical Errors	0 + 4 + 1	2 + 3 + 4	0 + 0 + 1	0 + 1 + 0	1 + 4 + 1	22
Total		40	82	12	12	39	185

Case 1: Auto Memory Reclaim

- Drop an object automatically if no variable owns it.

```
fn genvec()->Vec<u8>{
    let mut s = String::from("a tmp string");
    //let mut s = ManuallyDrop::new(String::from("a tmp string"));
    let ptr = s.as_mut_ptr();
    unsafe{
        let v = Vec::from_raw_parts(ptr,s.len(),s
        //panic!();
        //mem::forget(s);
        v
    }
}
fn main(){
    let v = genvec(); //v is dangling
    assert_eq!('a' as u8, v[0]);
}
```

create a temporary string s

create a Vec v pointing to s via unsafe

return v

drop s; v becomes dangling

accessing v causes use-after-free

PoC of CVE-2019-16140, CVE-2019-16144

Test the code with Miri: https://play.rust-lang.org

Case 2: Unsound API

- A safe API may lead to undefined behaviors

```
use std::slice;
fn foo (a: &mut [u8]){
    // require 4-byte alignment
    let p = a.as_mut_ptr() as *mut u32;
    unsafe {
        let s = slice::from_raw_parts_mut(p, 1);
        let _x = p[0];
    }
}

fn main(){
    let mut x = [0u8;10];
    foo(&mut x[1..9]);
}
```

accept a parameter of u8 slice

obtain the raw pointer as *u32

obtain a new slice of u32

out-of-bound read!!!

PoC of CVE-2021-45709

Case 3: Vulnerable Generic Parameters

- Generic parameters vulnerable to particular types
  - Existing types
  - Newly customized types, *e.g.*, CVE-2020-25796, CVE-2020-35903

```
use std::slice;
fn foo<T>(a: &mut [T]){
    // require 4-byte alignment
    let p = a.as_mut_ptr() as *mut u32;
    unsafe {
        let s = slice::from_raw_parts_mut(p, 1);
        let _x = p[0];
    }
}

fn main(){
    let mut x = [0u8;10];
    foo(&mut x[1..9]);
}
```

accept a parameter of u8 slice

monomorphize T with a u8 slice

PoC of advanced CVE-2021-45709

Case 4: Vulnerable Generic Parameters: Insufficient Trait Bound

```
use std::rc::Rc;
#[derive(Clone)]
struct MyStruct<T> {t:T}
unsafe impl<T> Send for MyStruct<T>{}
//unsafe impl<T:Send> Send for MyStruct<T>{}

fn main(){
    let mut s = MyStruct{t:Rc::new(String::from("da
    for i in 0..999{
        let mut c = s.clone();
        std::thread::spawn(move ||{
            if !Rc::get_mut(&mut c.t).is_none(){
                (*Rc::get_mut(&mut c.t).unwrap()).clear();
            }
            println!("c.t = {:?}", c.t);
        });
    }
}
```

define a generic struct MyStruct

impl Send Trait for MyStruct

monomorphize T of MyStruct with Rc (thread-unsafe)

concurrency bug!!!

PoC of CVE-2020-35870, CVE-2020-35871, and CVE-2020-35886

Fixed (improved) in latest Rust (after 1.63)

Case 5: Unsound Trait

- Reimplement a safe function may lead to undefined behaviors.

```
trait MyTrait {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<Self>()
    }
}
impl dyn MyTrait {
    pub fn is<T: MyTrait + 'static>(&self) -> bool {/*...*/}
    pub fn downcast<T: MyTrait + 'static>(self: Box<Self>)
        -> Result<Box<T>, Box<dyn MyTrait>> {/*...*/}
}
impl MyTrait for u128{}
impl MyTrait for u8{
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<u128>()
    }
}
fn main(){
    let s = Box::new(10u8);
    let r = MyTrait::downcast::<u128>(s);
}
```

return the type of the trait object

reimpl type_id() for u8 with errors

bug: out-of-bound access!!!

Simplified PoC of CVE-2019-12083

Our Interest: Towards More Secure Rust Crates

1) How to mitigate the usage of unsafe code?
2) How to verify the security of unsafe code?
   - ✓ Detect bugs caused by unsafe code
   - ✓ Soundness verification of interior unsafe API

Rice's Theorem

⇩

NP-hard Problem

⇩

Soundness vs Completeness vs Efficiency

4) Characteristics of Unsafe Code

4) Characteristics of Unsafe Code
5) Contracts of Using Unsafe Code
6) Replacement of Unsafe Code

Privileged Unsafe Code

Application Scenarios	Five Types of Unsafe Code in Rustdoc				
	Raw Ptr	Unsafe Fn	Unsafe Trait	Static Mut	Union
Low-level control	✓	✓			
Interoperability		✓			✓
Non-exclusive Mutability	✓	✓			
Delayed Initialization	✓	✓			
Transmute		✓			
Unchecked Operations	✓	✓			
Tailored Allocator			✓		
Concurrent Objects			✓		
Global Objects				✓	

Low-level Control: OS Dev as An Example

Case 1: write to device via raw pointer

```
let vga_buffer = 0xb8000 as *mut u8;
unsafe{
    *vga_buffer.offset(0) = byte;
    *vga_buffer.offset(0 + 1) = colour;
}
```

Case 2: I/O read/write via inline assembly

```
unsafe {
    asm!("in al, dx", out("al") value, in("dx") port
         , ...);
}
```

Case 3: memory allocation is unsafe itself

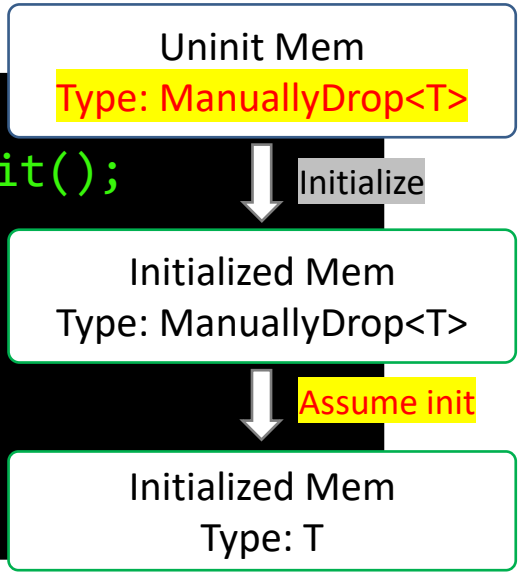
```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
}
unsafe impl GlobalAlloc for XXX {
    unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
        ...
    }
}
```


https://os.phil-opp.com/minimal-rust-kernel/

Delayed Initialization: Uninitialized Memory

- Create an uninitialized object is unsafe
  - The method mem::uninitialized() is deprecated
- Use MaybeUninit: create an object of MaybeUninit is safe
  - Assume initialization done is unsafe

```
//Step1: create an object of MaybeUninit
let mut v:MaybeUninit<Vec<i32>> = MaybeUninit::uninit();

//Step2: code to initialize the vector
unsafe { v.as_mut_ptr().write(vec![1,2,3]); }

//Step3: initialization done; unwrap MaybeUninit
let v = unsafe { v.assume_init() };
```



https://doc.rust-lang.org/std/mem/union.MaybeUninit.html

Unchecked Operations

- Skip the validity (e.g., boundary) check
- More efficient but insecure

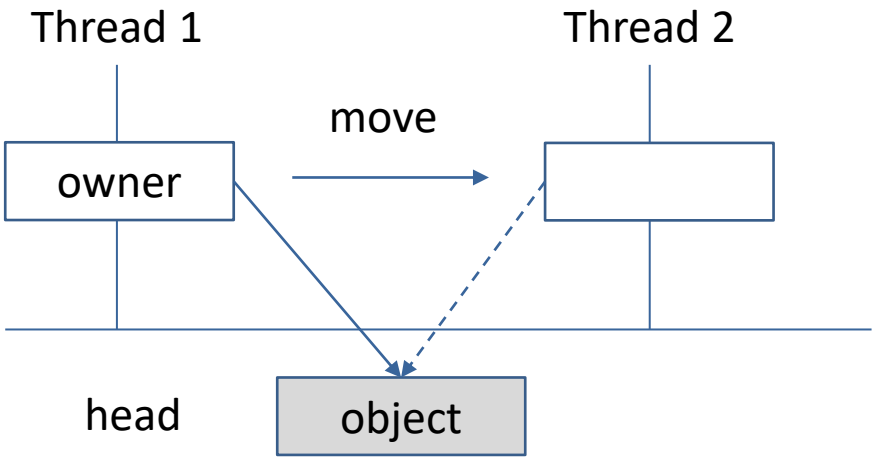
```
let mut x = vec![1, 2, 3];
unsafe {
    let elem = x.get_unchecked_mut(100);
    *elem = 10;
}
unsafe {
    let (left, right) = v.split_at_unchecked(50);
}
```


https://doc.rust-lang.org/std/vec/struct.Vec.html

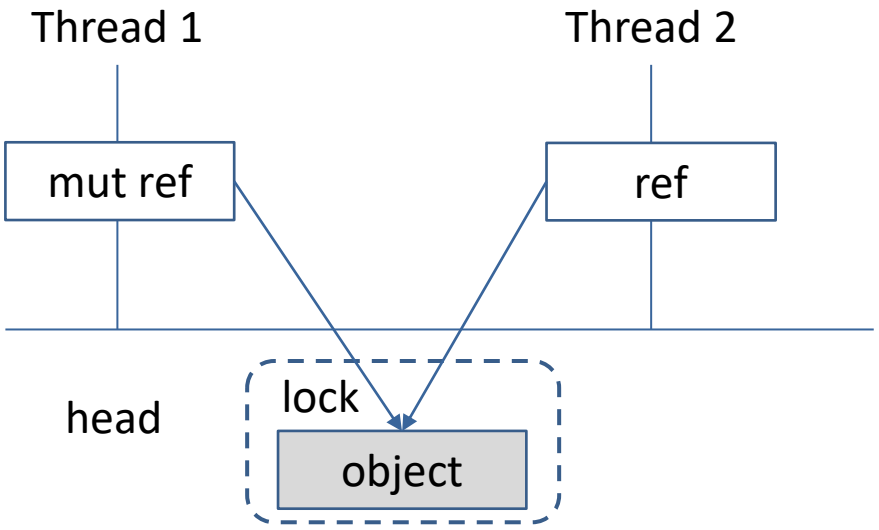
Concurrency Safety

- Whether allow an object to be used in multiple threads

```
struct MyType{ ... }
unsafe impl Send for MyType{}
unsafe impl Sync for MyType{}
```



Send: object can be passed among threads



Sync: object ref can be passed among threads

5) Contracts of Using Unsafe Code

4) Characteristics of Unsafe Code
5) Contracts of Using Unsafe Code
6) Replacement of Unsafe Code

[Cui'23a] Mohan Cui, *et al*. "Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming." *arXiv* 2023.

Contract: Some Unsafe APIs Have Preconditions

- Recall the bug of unsound APIs

```
fn foo<T>(a: &mut [T], l: usize){
    // require 4-byte alignment
    let p = a.as_mut_ptr() as *mut u32;
    unsafe {
        let s = slice::from_raw_parts_mut(p, l);
    }
    let _x = p[0];
}

fn main(){
    let mut x = [0u8;10];
    foo(&mut x[1..9], 1);
}
```

Rustdoc Provides Safety Requirements (Verbose)

Function std::slice::from_raw_parts_mut

1.0.0 (const: unstable) · source · [-]

```
pub unsafe fn from_raw_parts_mut<'a, T>(data: *mut T, len: usize) -> &'a mut [T]
```

[–] Performs the same functionality as `from_raw_parts`, except that a mutable slice is returned.

Safety

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

- `data` must be valid for both reads and writes for `len * mem::size_of::<T>()` many bytes, and it must be properly aligned. This means in particular:
  - The entire memory range of this slice must be contained within a single allocated object! Slices can never span across multiple allocated objects.
  - `data` must be non-null and aligned even for zero-length slices. One reason for this is that enum layout optimizations may rely on references (including slices of any length) being aligned and non-null to distinguish them from other data. You can obtain a pointer that is usable as `data` for zero-length slices using `NonNull::dangling()`.
- `data` must point to `len` consecutive properly initialized values of type `T`.
- The memory referenced by the returned slice must not be accessed through any other pointer (not derived from the return value) for the duration of lifetime `'a`. Both read and write accesses are forbidden.
- The total size `len * mem::size_of::<T>()` of the slice must be no larger than `isize::MAX`. See the safety documentation of `pointer::offset`.

https://doc.rust-lang.org/beta/core/slice/fn.from_raw_parts_mut.html

Summary of Preconditions

Safety Property (SP)	SUM	Definition and the safety requirement of each Safety Property.	Unsafe API Example
Precondition Safety Property			
Const-Numeric Bound	**72**	Relational operations allow for **compile-time** determination of the **constant** numerical boundaries on one side of an expression, including overflow check, index check, etc.	**impl**<T: ?Sized> *mut T::offset_from
Relative-Numeric Bound	**114**	Relational operations involve expressions where **neither side** is a constant numeric, including address boundary check, overlap check, size check, variable comparison, etc.	**trait Allocator**::grow
Encoding	**16**	Encoding format of the string, includes valid **UTF-8** string, valid **ASCII** string (in bytes), and valid **C-compatible** string (nul-terminated trailing with no `nul` bytes in the middle).	**impl String**::from_utf8_unchecked
Allocated	**134**	Value stored in the **allocated memory**, including data in the valid stack frame and allocated heap chunk, which cannot be `NULL` or dangling.	**impl**<T: Sized> **NonNull**<T>::new_unchecked
Initialized	**59**	Value that has been initialized can be divided into two scenarios: **fully initialized** and **partially initialized**. The initialized value must be **valid** at the given type (a.k.a. **typed**).	**impl**<T> **MaybeUninit**<T>::assume_init
Dereferencable	**96**	The memory range of the given size starting at the pointer must all be within the bounds of a **single allocated object**.	**impl**<T: ?Sized> *const T::as_ref
Aligned	**67**	Value is **properly aligned** via a specific **allocator** or the attribute `#[repr]`, including the alignment and the padding of one Rust type.	**impl**<T: ?Sized> *mut T::swap
Consistent Layout	**110**	Restriction on Type Layout, including 1) The **pointer's** type must be compatible with the **pointee's** type; 2) The **contained value** must be compatible with the generic parameter for the smart pointer; and 3) Two types are **safely transmutable**: The bits of one type can be **reinterpreted** as another type (bitwise move safely of one type into another).	**impl**<T: ?Sized> *mut T::read
Unreachable	**9**	Specific value will trigger **unreachable data flow**, such as **enumeration** index (**variance**), boolean value, closure and etc.	**impl**<T> **Option**<T>::unwrap_unchecked
Exotically Sized Type	**24**	Restrictions on Exotically Sized Types (EST), including **Dynamically Sized Types** (DST) that lack a statically known size, such as trait objects and slices; **Zero Sized Types** (ZST) that occupy no space.	**trait GlobalAlloc**::alloc
System IO	**25**	Variables related to the **system IO** depends on the **target platform**, including TCP sockets, handles, and file descriptors.	**trait FromRawFd**::from_raw_fd
Thread	**3**	Types that can be **transferred** across threads (**Send**) or types that can be safe to **share references** between threads (**Sync**), respectively.	**std**::**marker**::**Sync**

Contract: Some Unsafe APIs have Postconditions

```
#[derive(Debug)]
pub struct Error<T> {
    pub ptr: *mut T
}

impl<T> Drop for Error<T> {
    fn drop(&mut self) {
        unsafe {
            ptr::drop_in_place(self.ptr);
        }
    }
}

fn main(){
    let x = Box::new(1u32);
    let p = Box::into_raw(x);
    let _ = Error { ptr:p };
    ...
}
```

The address should not be used after drop_in_place()

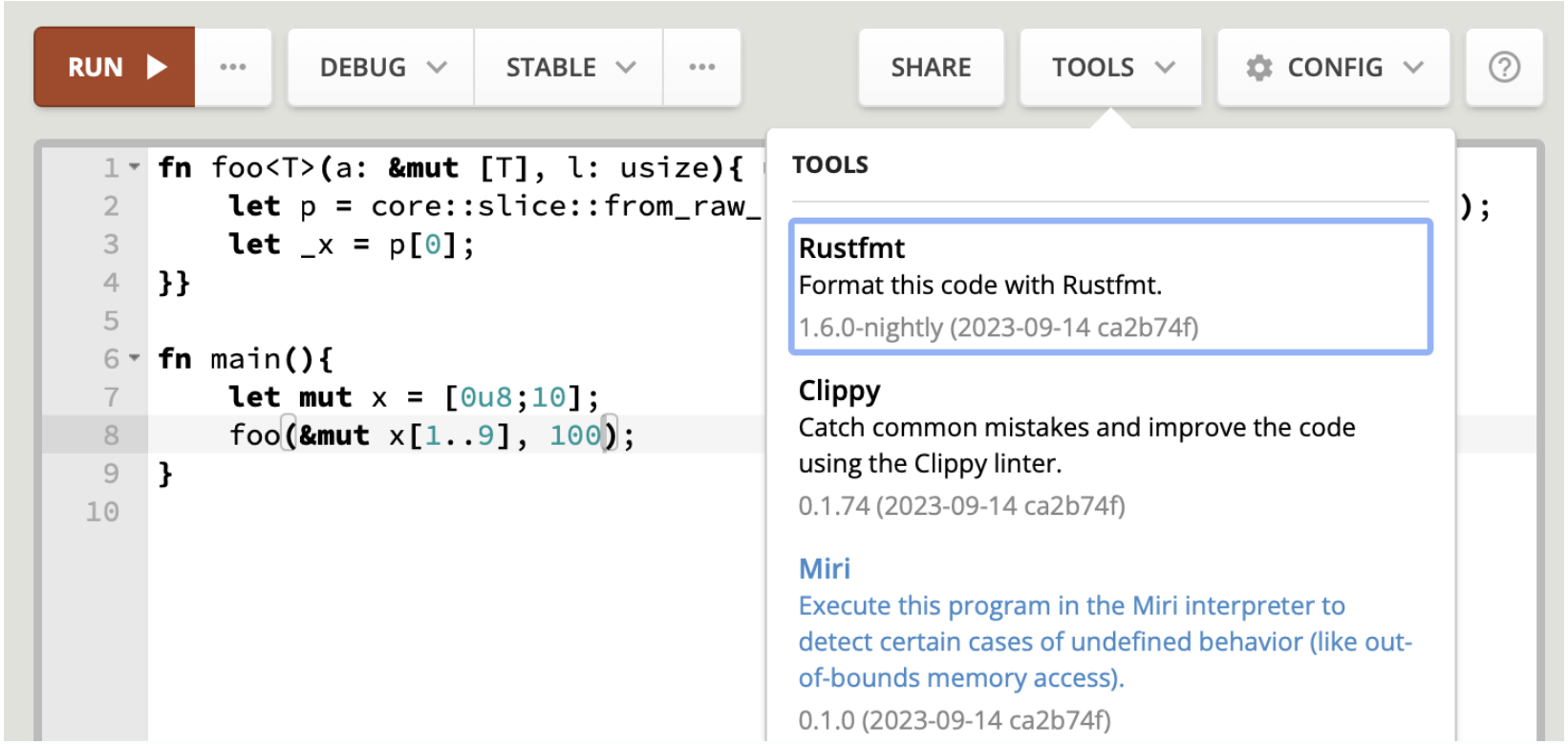
PoC of CVE-2020-36205

Summary of Postconditions

Postcondition Safety Property			
Dual Owner	**31**	Multiple owners (**overlapped objects**) that share the same memory in the ownership system by **retaking the owner** or **creating a bitwise copy**.	**impl**<**T: ?Sized**> **Box**<**T**>::from_raw
Aliasing & Mutating	**30**	Aliasing and mutating rules may be violated, including 1) The presence of **multiple** mutable references; 2) The **simultaneous** presence of mutable and shared references, and the memory the pointer points to cannot get mutated (**frozen**); 3) Mutating **immutable data** owned by an immutable binding.	**impl CStr**::from_ptr
Outliving	**28**	**Arbitrary lifetime** (unbounded) that becomes as big as context demands or **spawned thread**, may outlive the pointed memory.	**impl**<**T: ?Sized**> *const **T**::as_uninit_ref
Untyped	**20**	Value may not be in the **initialized state**, or the **byte pattern** represents an **invalid value** of its type.	**core**::**mem**::zeroed
Freed	**17**	Value may be manually **freed** or **released** by automated **`drop()`** instruction.	**impl**<**T: ?Sized**> **ManuallyDrop**<**T**>::drop
Leaked	**13**	Value may be **leaked** or **escaped** from the ownership system.	**impl**<**T: ?Sized**> *mut **T**::write
Pinned	**5**	Value may be **moved**, although it ought to be pinned.	**impl**<**P: Deref**> **Pin**<**P**>::new_unchecked

Verify Whether The Contracts Can Be Met?

- Miri can detect undefined behaviors of by executing IR.
- Limitation: based on dynamic analysis, need concrete test cases.

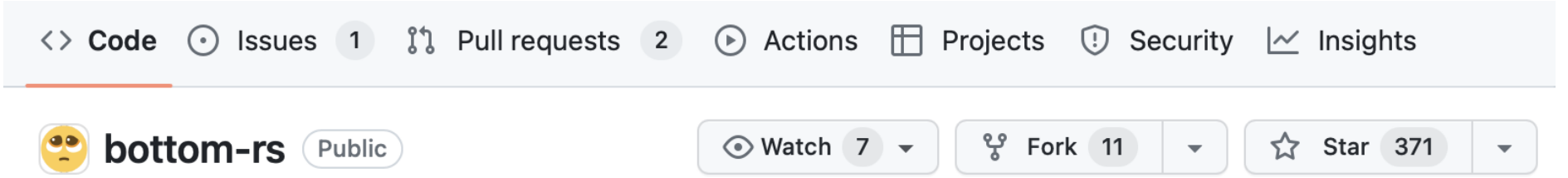


https://play.rust-lang.org

6) Replacement of Unsafe Code

4) Characteristics of Unsafe Code
5) Contracts of Using Unsafe Code
6) Replacement of Unsafe Code

Motivating Example: Misused Unsafe Code



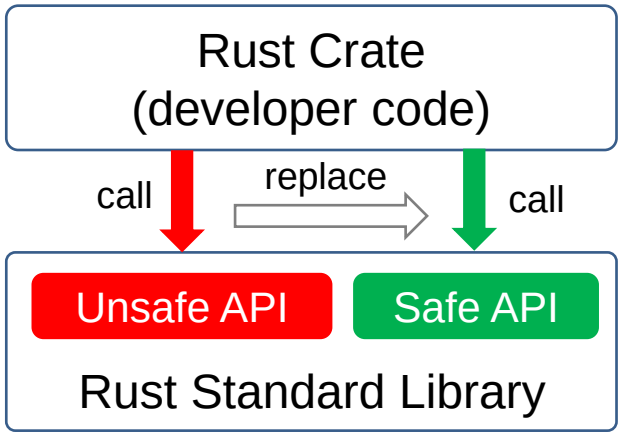

```
    static ref BYTE_TO_EMOJI: [String; 256] = {
-       // SAFETY: safe
-       let mut m: [MaybeUninit<String>; 256] = unsafe { MaybeUninit::uninit().assume_init() };
+       const EMPTY_STRING: String = String::new();
+
+       let mut m = [EMPTY_STRING; 256];
        for i in 0..=255u8 {
-           m[i as usize] = MaybeUninit::new(byte_to_emoji(i));
+           m[i as usize] = byte_to_emoji(i);
        }
-       unsafe { mem::transmute::<_, [String; 256]>(m) }
+       m
    };
```


https://github.com/bottom-software-foundation/bottom-rs/pull/6/files

Detailed Issues

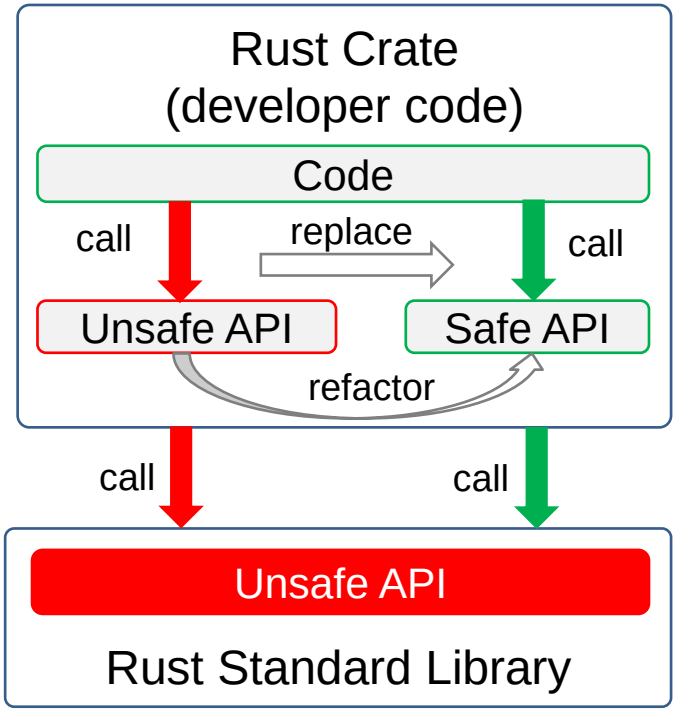
1) Substitutability: is it possible to replace the unsafe code?
2) How to replace: what is the corresponding safe version?
3) Cost of replacement: whether the replacement is worthy?

Case 1: function level
(code replacement)



our current focus

Case 2: crate level
(code refactoring)



Substitutability of Unsafe Code

Application Scenarios	Five Types of Unsafe Code in Rustdoc					Substitutability	
	Raw Ptr	Unsafe Fn	Unsafe Trait	Static Mut	Union	Replaceable?	How?
Low-level control	✓	✓				no	
Interoperability		✓			✓	no	
Non-exclusive Mutability	✓	✓				may	intelli ptr
Delayed Initialization	✓	✓				may	init twice
Transmute		✓				may	safe cast
Unchecked Operations	✓	✓				yes	checked
Tailored Allocator			✓			no	
Concurrent Objects			✓			no	
Global Objects				✓		yes	lock

Example: Replacement of Unchecked

```
let mut x = vec![1, 2, 3];
unsafe {
    let elem = x.get_unchecked_mut(100);
    *elem = 10;
}
unsafe {
    let (left, right) = v.split_at_unchecked(50);
}
```

Always possible, with runtime cost

```
let mut x = vec![1, 2, 3];
unsafe {
    let elem = x.get_mut(100);
    *elem = 10;
}
unsafe {
    let (left, right) = v.split_at(50);
}
```

Example: Replacement of Uninitialized Memory

```
//Step1: create an object of MaybeUninit
let mut v:MaybeUninit<Vec<i32>> = MaybeUninit::uninit();

//Step2: code to initialize the vector
unsafe { v.as_mut_ptr().write(vec![1,2,3]); }

//Step3: initialization done; unwrap MaybeUninit
let v = unsafe { v.assume_init() };
```

Not always possible, with runtime cost

```
//Step1: Create an initliazed object
let mut v:Vec<i32> = vec![0;3]; //not always possible
//Step2: Reinitialize the object
let mut i:i32 = 0;
for it in v.iter_mut() {
    *it = {i+=1;i};
}
```

Example: Replacement of Transmute

```
fn rs_template_state_new() -> *mut std::os::raw::c_void {
    let state = TemplateState::new();
    let boxed = Box::new(state);
-   return unsafe { transmute(boxed) };
+   return Box::into_raw(boxed) as *mut c_void;
}
```

Case of misuse: replacement has no cost

```
let store = [0, 1, 2, 3];
let v = store.iter().collect::<Vec<&i32>>();
- let v = unsafe { mem::transmute::<Vec<&i32>, Vec<Option<&i32>>>(v) };
+ let v = v.into_iter().map(Some).collect::<Vec<Option<&i32>>>();
```

Replacement has cost

- Let developers decide whether to replace

7) Detecting Dangling Pointer Bugs

[Cui&Chen'22] Mohan Cui, Chengchun Chen, *et al*. "SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis", *TOSEM*, 2022.

Recall the Auto Memory Reclaim Bug

```
fn genvec()->Vec<u8>{
    let mut s = String::from("a tmp string");
    //let mut s = ManuallyDrop::new(String::from("a tmp string"));
    let ptr = s.as_mut_ptr();
    unsafe{
        let v = Vec::from_raw_parts(ptr,s.len(),s
        //panic!();
        //mem::forget(s);
        v
    }
}
fn main(){
    let v = genvec(); //v is dangling
    assert_eq!('a' as u8, v[0]);
}
```

create a temporary string s

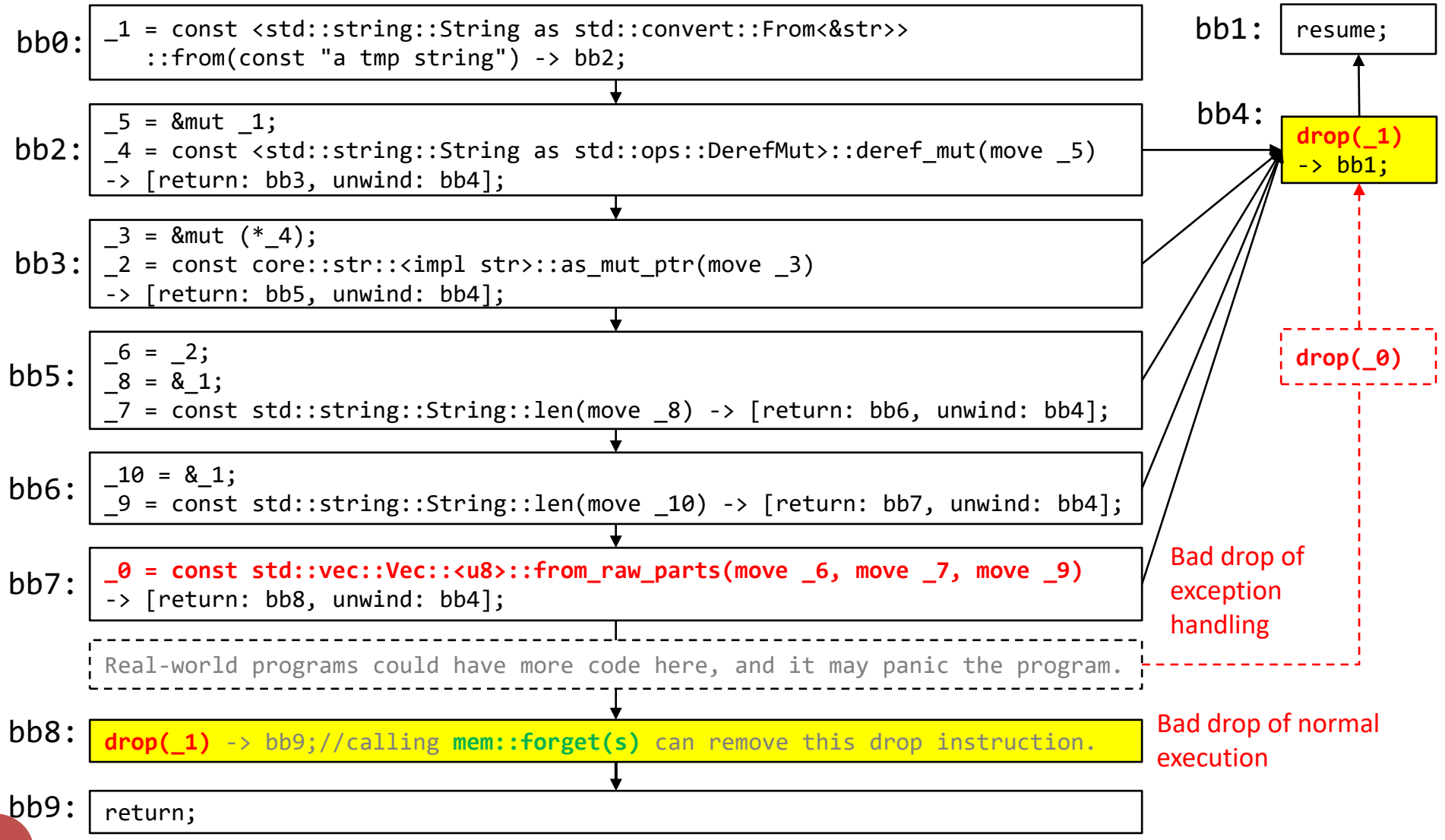
create a Vec v pointing to s via unsafe

return v

drop s; v becomes dangling

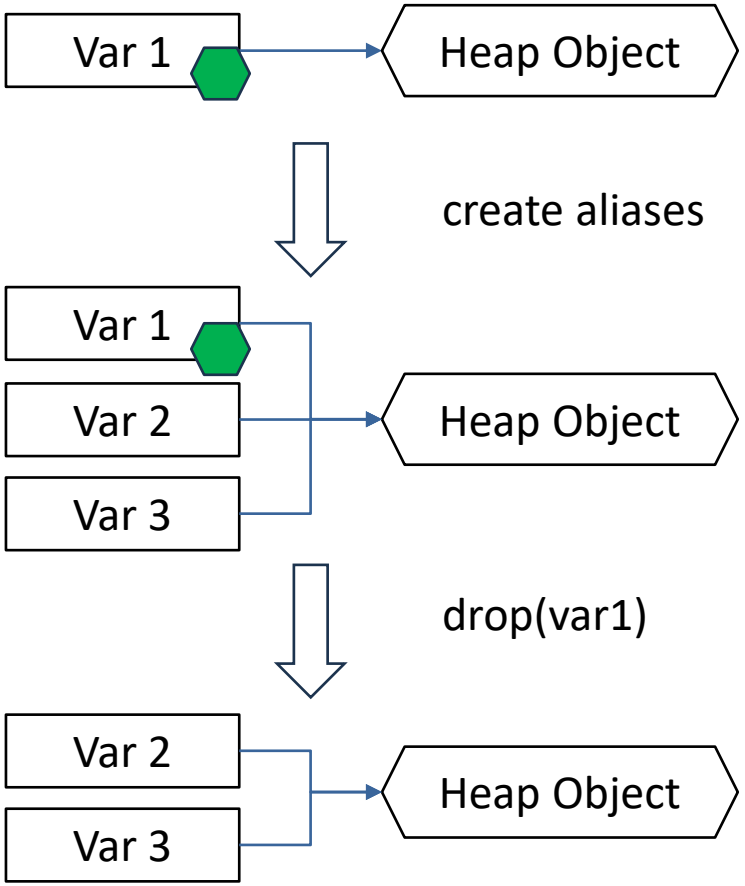
accessing v causes use-after-free

Bug Analysis with Rust MIR

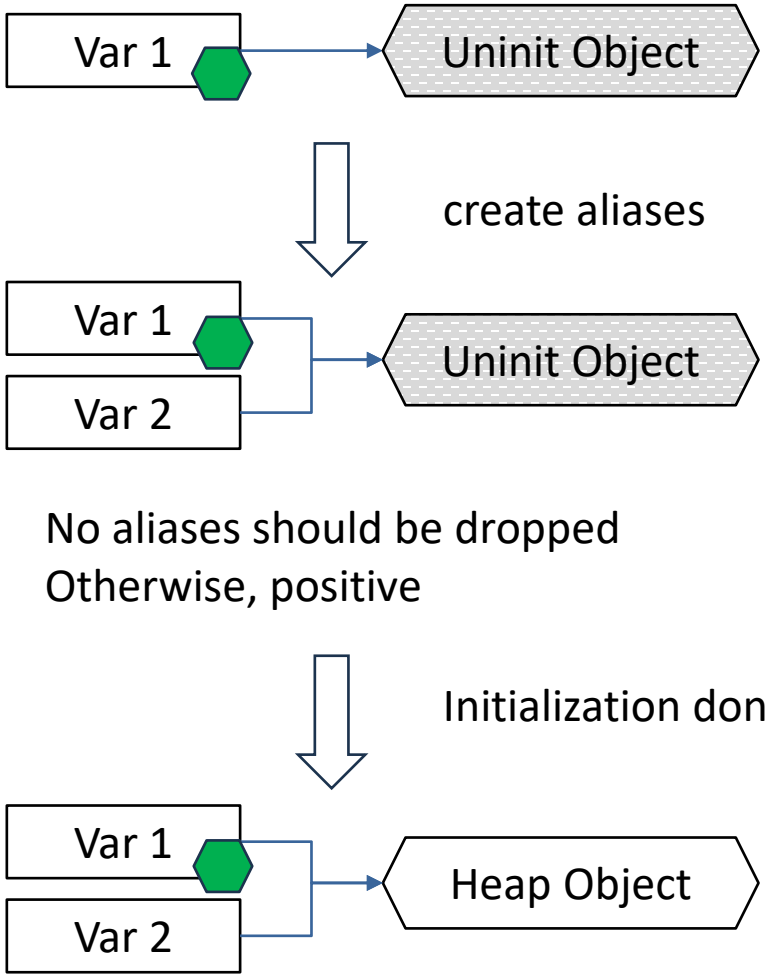


Abstraction of Bug Patterns

ownership



- No aliases should be used or dropped
- Otherwise, positive (bug detected)



- No aliases should be dropped
- Otherwise, positive

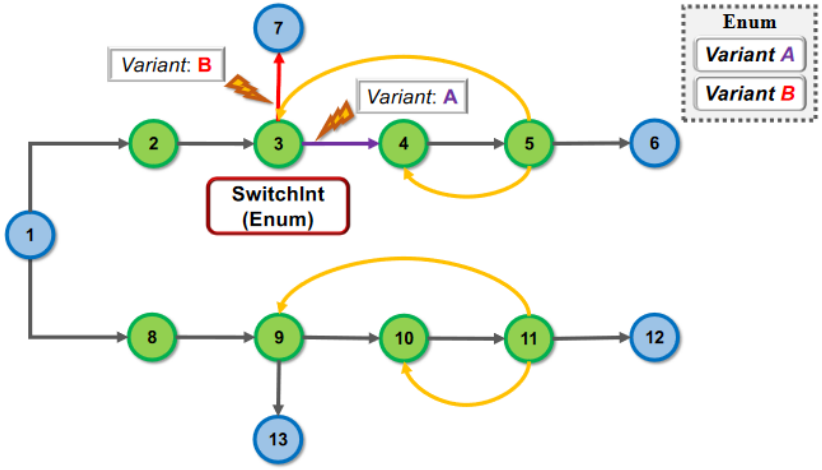
Approach for Detecting Dangling Pointer Bugs

- Requirements:
  - Effective: use-after-free, double free, drop unint memory
  - Precise: should not incur much false positives
  - Efficient: fast
- Approach: path-sensitive analysis

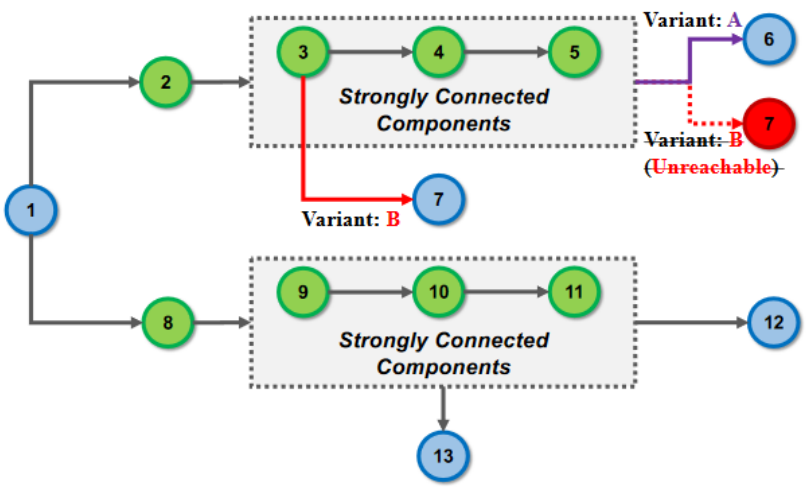
Approach:

1. Path Extraction ⇨ 2. Alias Analysis ⇨ 3. Pattern Detection

- Extract the spanning tree of each function:
  - Compute strongly-connected components with Tarjan Algorithm
  - Compute the may alias sets of each SCC
- Refine the tree based on rules to handle conner cases



Control-flow Graph



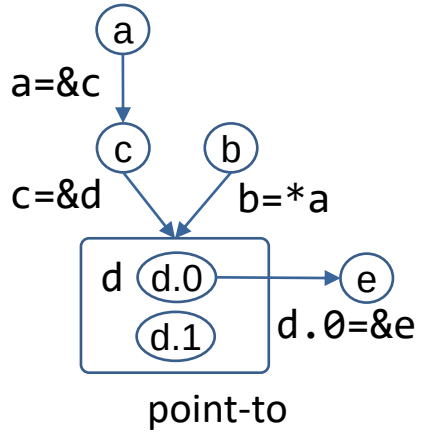
Spanning Tree

Approach:

1. Path Extraction ⇨ **2. Alias Analysis** ⇨ 3. Pattern Detection

- Classic accurate rules for point-to analysis are inefficient
  - Anderson-style

Pattern	MIR	Abstraction
Move	`a = move b`	$S_a = S_a - a,$ $S_b = S_b \cup a$
Copy	`a = b`	$S_a = S_a - a,$ $S_b = S_b \cup a$
Ref	`a = &b`	$S_a = S_a - a,$ $S_{loc(b)} = S_{loc(b)} \cup a$
Deref	`a = *b`	$S_a = S_a - a,$ $\forall v \in pts(b), S_v = S_v \cup a$
Fn(Mov)	`a = func(mov b)`	$Update(S_a, S_b)$
Fn(Copy)	`a = func(b)`	$Update(S_a, S_b)$



alias set: {b, c}

Approach:

1. Path Extraction ⇨ **2. Alias Analysis** ⇨ 3. Pattern Detection

- Our approximation rules:
  - Similar to Steensgaard, but ignore multi-level pointers

```
LValue := Use::Move(RValue)      : e.g., a = move b       =>   S_a = S_a − a,  S_b = S_b ∪ a
    |  := Use::Copy(RValue)      : e.g., a = b            =>   S_a = S_a − a,  S_b = S_b ∪ a
    |  := Ref/AddressOf(RValue)  : e.g., a = &b           =>   S_a = S_a − a,  S_b = S_b ∪ a
    |  := Deref(RValue)          : e.g., a = *(b)         =>   S_a = S_a − a,  S_b = S_b ∪ a
    |  := Fn(Move(RValue))       : e.g., a = Fn(move b)   =>   Update(S_a, S_b)
    |  := Fn(Copy(RValue))       : e.g., a = Fn(b)        =>   Update(S_a, S_b)
```

- Example:

```
Statement 1:  _2 = &_1;          // alias set:{_1, _2}
Statement 2:  _1 = move _4;      // alias sets:{_1, _4}, {_2}
Statement 3:  _3 = &_1;          // alias sets:{_1, _3, _4}, {_2}
```

Approach:

1. Path Extraction ⇨ **2. Alias Analysis** ⇨ 3. Pattern Detection

- Field-sensitive and inter-procedural analysis

```
enum E { A, B { ptr: *mut u8 } }
struct S { b: E }

fn foo(_1: &mut String) -> S:
    _3 = str::as_mut_ptr(_1);  // alias set: {_3, _1}
    ((_2 as B).0: *mut u8) = move _3;   // alias set: {_2.0, _3, _1}
    discriminant(_2) = 1;  // instantiate the enum type to variant B
    (_0.0: E) = move _2; // alias sets: {_0.0, _2}, {_0.0.0, _2.0, _3, _1}
    return;

fn main():
    _1 = String::from("string"); // alias set: {_1},
    _2 = &mut _1;   // alias set: {_2, _1},
    _3 = foo(move _2); // alias set: {_3.0.0, _2, _1}
    ...
```

field-sensitive

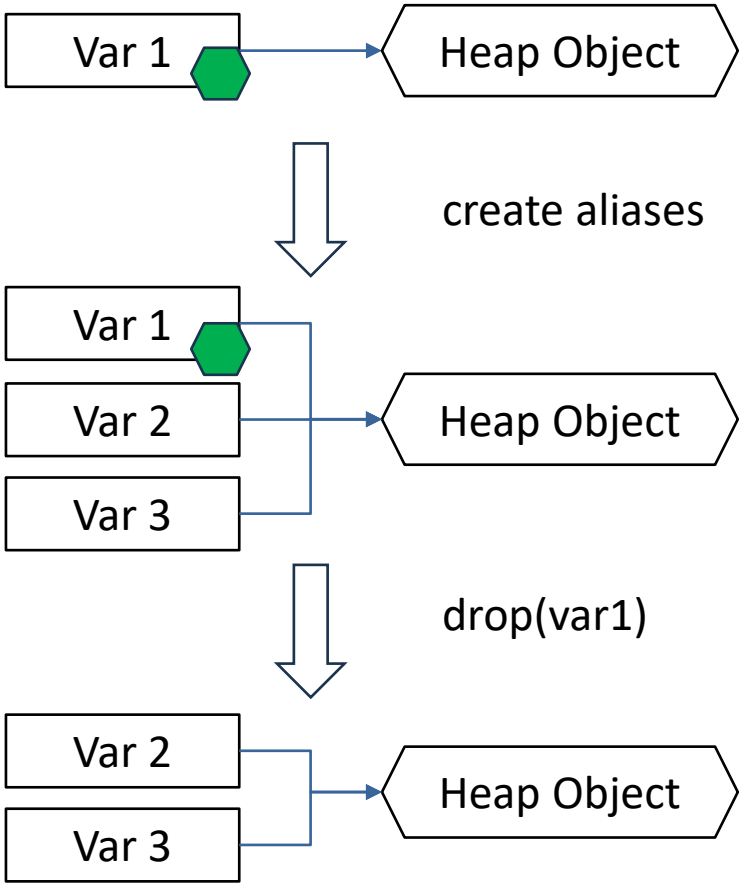
update

{0.0.0, 1}

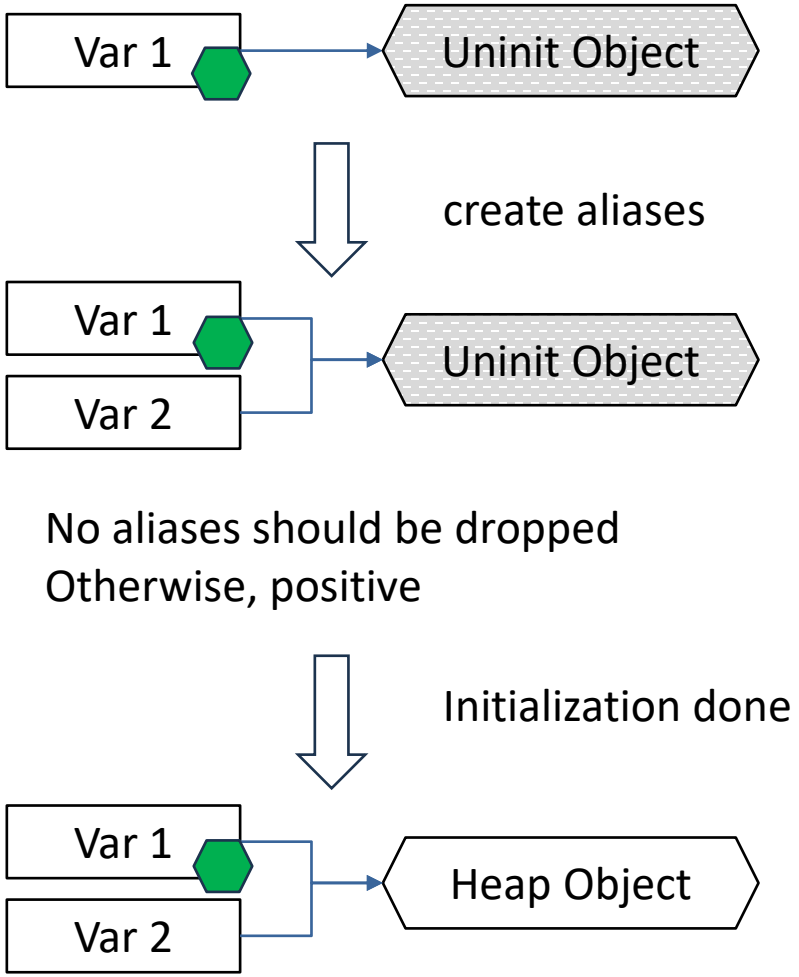
inter-procedural

Approach:

1. Path extraction ⇨ 2. Alias Analysis ⇨ **3. Pattern Detection**



- No aliases should be used or dropped
- Otherwise, positive



- No aliases should be dropped
- Otherwise, positive

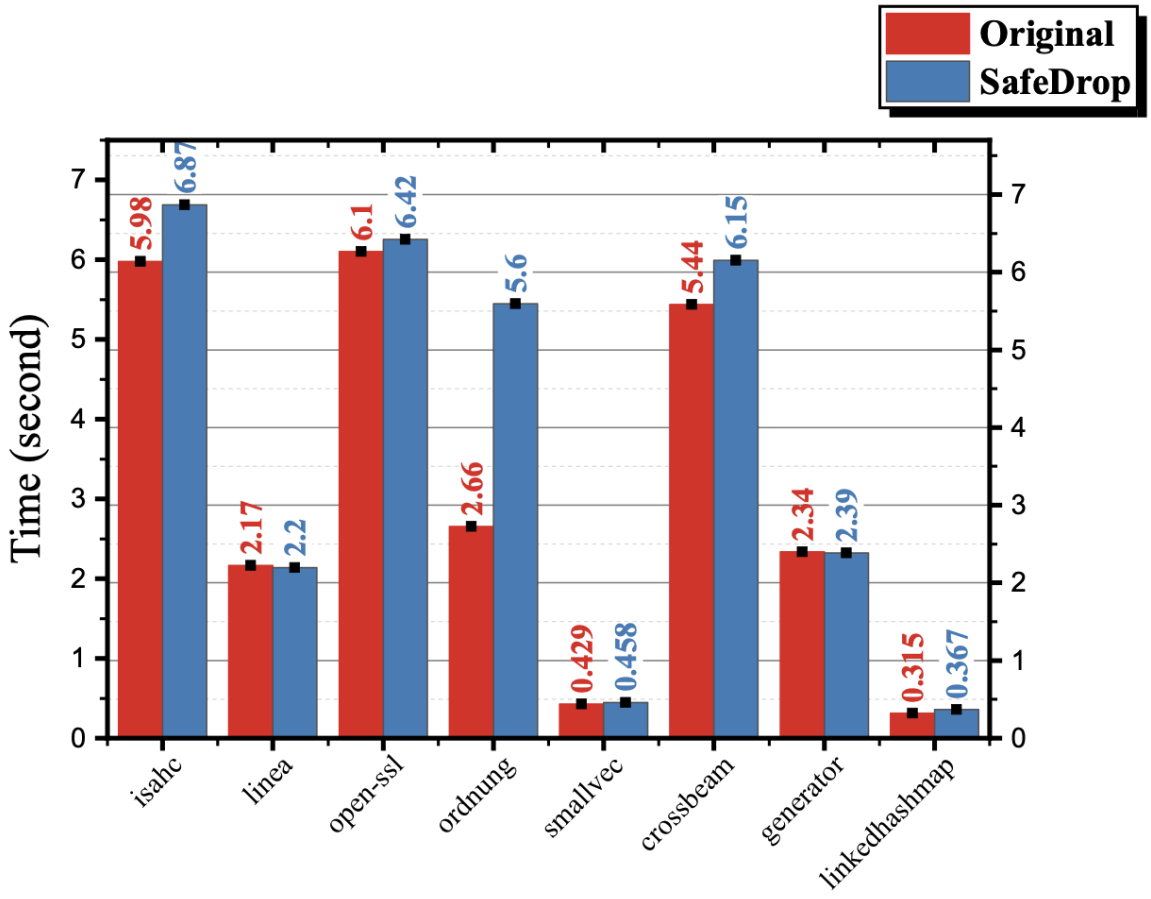
Experimental Results

- Can detect all related CVEs with a low false positive rate
- False positive: compiler intrinsic trait implementations are unavailable
  - Assume aliases among the arguments and return value

Crate			CVE		SafeDrop Report (TP/FP)						Rudra	MirChecker	Miri	
Name	# Methods	LoC	CVE-ID	Type	UAF	DF	DP	IMA	Total	Recall	TP/FP	TP/FP	# Tests	TP/FP
isahc	89	1304	2019-16140	UAF	-	0/1	1/0	-	1/1	100%	0/0	0/0	6	0/0
open-ssl	1188	20764	2018-20997	UAF	1/2	-	0/1	-	1/3	100%	0/0	0/0	0	N.A.
linea	1810	24317	2019-16880	DF	-	1/0	-	10/0	11/0	100%	1/2	0/5	2	0/0
ordnung	145	2546	2020-35891	DF	0/1	-	3/0	-	3/1	100%	1/3	0/2	31	0/0
crossbeam	221	4184	2018-20996	IMA	-	0/1	-	2/0	2/1	100%	0/0	0/0	21	0/1
generator*	158	2608	2019-16144	IMA	-	-	-	1/0	1/0	100%	0/0	0/0	0	N.A.
linkedhashmap	137	1974	2020-25573	IMA	-	-	-	1/0	1/0	100%	0/0	0/0	39	0/1
smallvec*	187	2297	2018-20991 2019-15551	DF DF	-	-	1/2	1/0	2/2	100%	2/1	0/2	48	1/1

Experimental Results: Efficiency

- Very small overhead compared to the original compilation



8) Detecting Memory Leakage Bugs

7) Detecting Dangling Pointer Bugs
8) Detecting Memory Leakage Bugs

[Cui'23b] Mohan Cui, *et al.* "rCanary: Detecting memory leaks across semi-automated memory management boundary in Rust", *arXiv*, 2023.

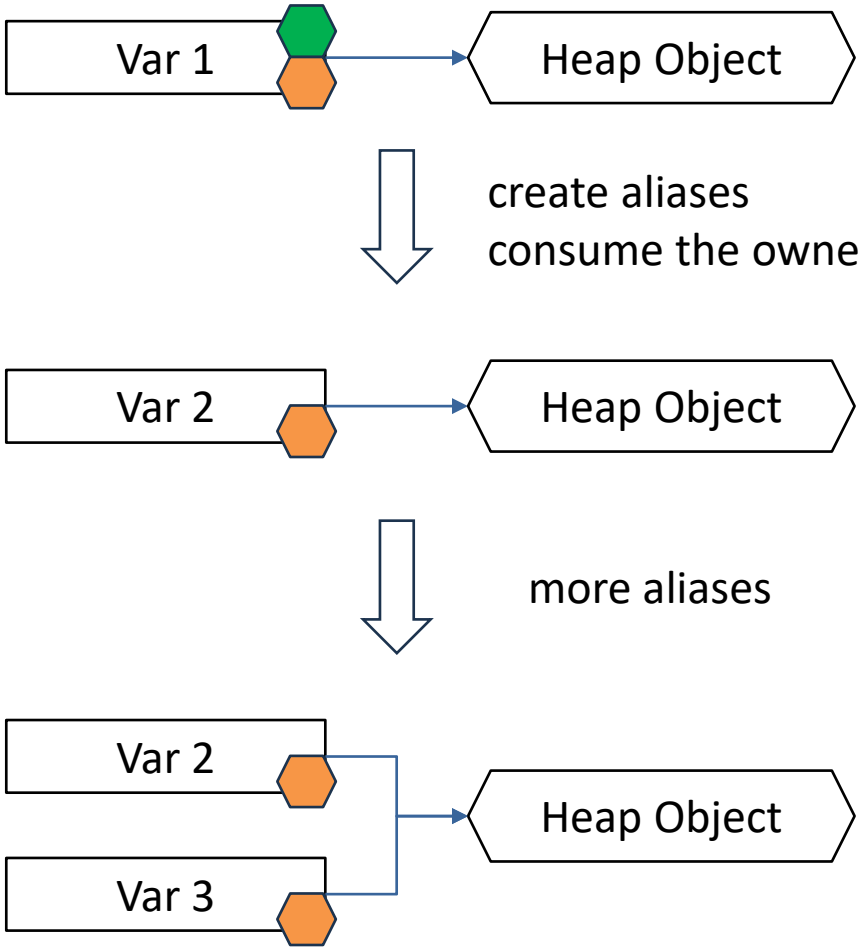
Memory Leakage Problem

- Consuming an ownership (leak) is safe in Rust
- Taking an ownership or manual deallocation is unsafe

```
  let mut buf = Box::new("buffer");
  let ptr = Box::into_raw(buf); //consume the ownership
+ //fix 1: unsafe { let _ = *ptr; }
+ //fix 2: unsafe { drop_in_place(ptr); }
```

buf is not owned by anyone

Abstraction of Bug Patterns



At least one variable should be properly dropped
Otherwise, positive (leakage detected)

ownership

resource token

Practical Cases are More Complicated

- Field-sensitivity issues
- Correctness of Drop trait implementation

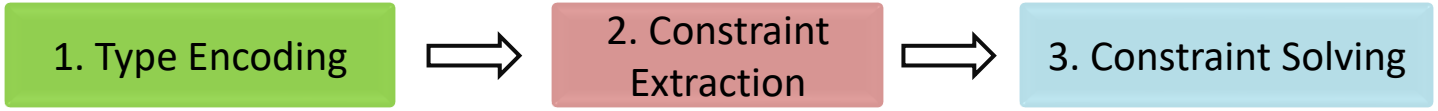
```
pub struct WString { ptr: NonNull<WStr>, capacity: usize, }
impl WString {
    unsafe fn steal_buf(&mut self)
                -> ManuallyDrop<Units<Vec<u8>, Vec<u16>>> { ... }
}
impl Drop for WString {
    fn drop(&mut self) {
        let _ = unsafe { self.steal_buf() };
-       unsafe {
            let mut buf = self.steal_buf();
+           ManuallyDrop::drop(&mut buf);
+       };
    }
}
```

steal_buf() returns a ManuallyDrop object

https://github.com/ruffle-rs/ruffle/pull/6528

Overall Idea: Model Checking

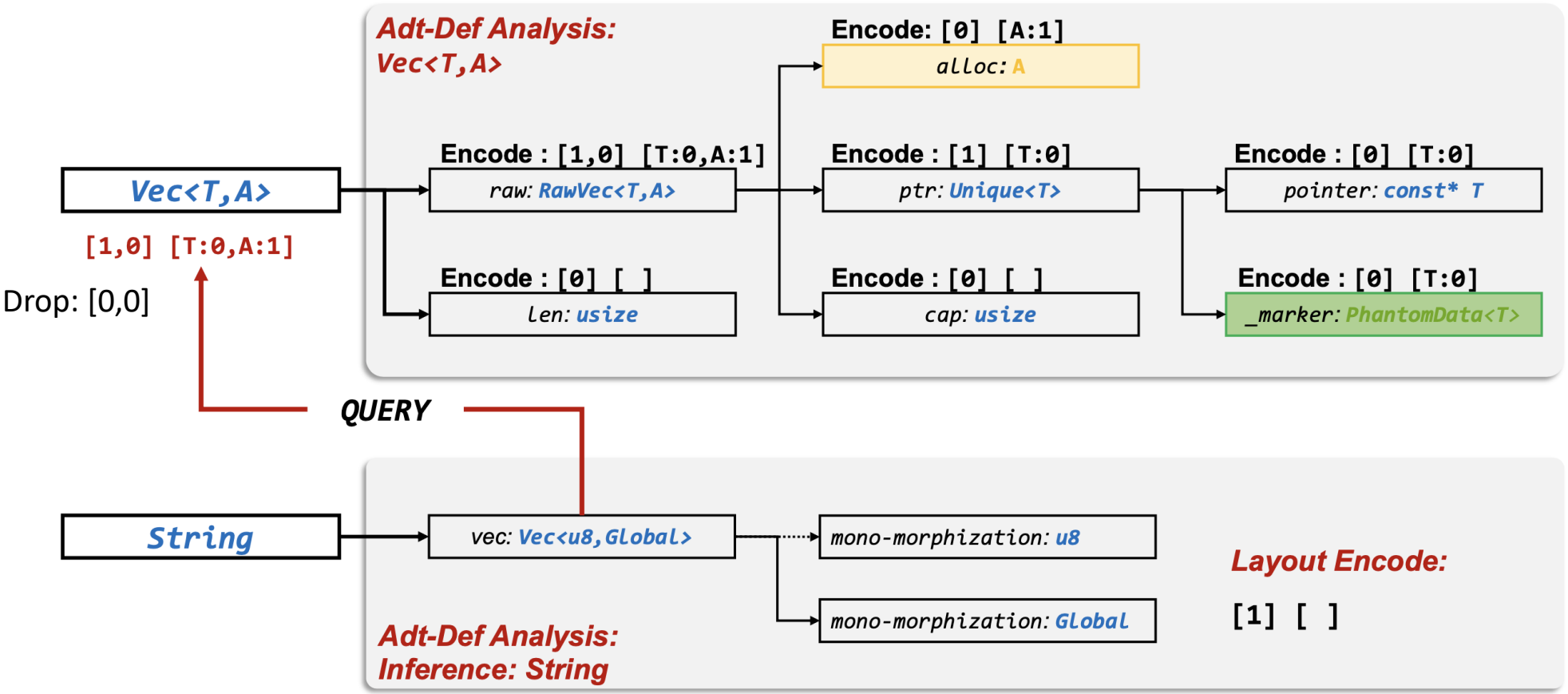
- Type encoding: abstract data types wrt heap resource holding
- Constraint extraction: path-insensitive data-flow analysis
  - fast, less false positives
- Constraint solving: based on Z3



Approach

1. Type Encoding ⇒ 2. Constraint Extraction ⇒ 3. Constraint Solving

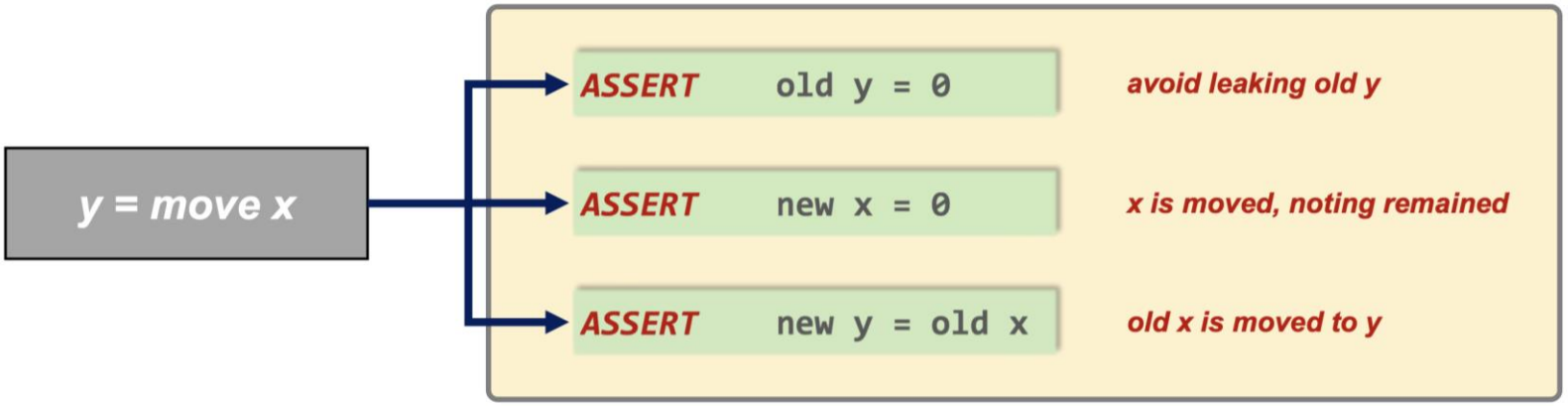
- Encode each type as a one-level bit vector (field-sensitive)
- Also encode the corresponding drop function as a one-level bit vector



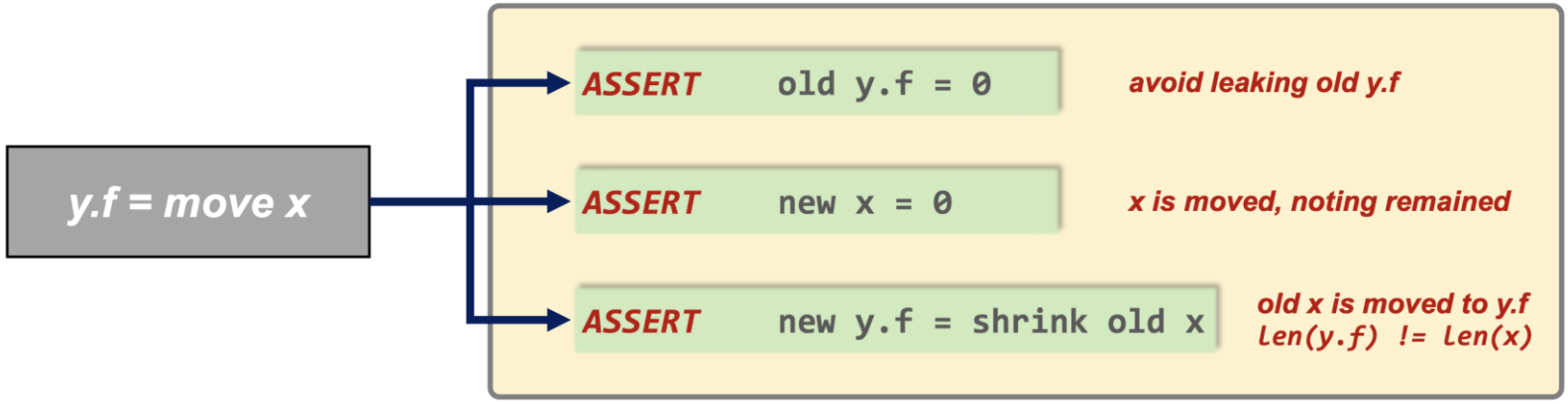
Approach

1. Type Encoding ⇒ **2. Constraint Extraction** ⇒ 3. Constraint Solving

$$\frac{M \vdash x:\overrightarrow{\sigma_x},\ y:\overrightarrow{\sigma_y} \qquad \overrightarrow{\sigma_x}',\ \overrightarrow{\sigma_y}'\ \textbf{new} \qquad C' = C \wedge \{\overrightarrow{\sigma_y} = \vec{0}\} \wedge \{\overrightarrow{\sigma_x}' = \vec{0}\} \wedge \{\overrightarrow{\sigma_y}' = \overrightarrow{\sigma_x}\}}{O; M; C \vdash y = \textbf{move}\ x \quad \Rightarrow \quad M[y \mapsto \overrightarrow{\sigma_y}',\ x \mapsto \overrightarrow{\sigma_x}']; C'}$$



$$\frac{M \vdash x:\overrightarrow{\sigma_x},\ y.f:\sigma_f \qquad \overrightarrow{\sigma_x}',\ \sigma_f'\ \textbf{new} \qquad C' = C \wedge \{\sigma_f = 0\} \wedge \{\overrightarrow{\sigma_x}' = \vec{0}\} \wedge \{\sigma_f = Srk(\overrightarrow{\sigma_x})\}}{O; M; C \vdash y.f = \textbf{move}\ x \quad \Rightarrow \quad M[y.f \mapsto \sigma_f' x \mapsto \overrightarrow{\sigma_x}']; C'}$$



Example

```
struct Proxy<T> { ptr: *mut T }

impl<T> Drop for Proxy<T> {
    fn drop(&mut self) {
        // user should manually deallocate the buffer in drop
    +   unsafe { Box::from_raw(self.ptr); }
    }
}
fn main() {
    let mut buf = Box::new("buffer");
    // transform the heap item 'buf' into the orphan object
    let ptr = Box::into_raw(buf);
    let proxy = Proxy { ptr };
    // leak due to lacking releasing 'proxy.ptr' in drop
}
```

Encoded the destructor as [1] since it does not drop ptr

ASSERT buf = [1,0]

ASSERT ptr = buf, buf'= [0,0]

ASSERT proxy = SRK(ptr), ptr'= [0,0]

ASSERT drop(proxy): proxy' = proxy ∩ [1]

ASSERT return: proxy', ptr', buf' = 0

Experimental Results

- Can detect all issues of a dataset collected from GitHub
- A small number of false positives

Package						Issue		rCanary		FFIChecker	Saber	ASAN	Miri	libFuzzer
Name	Functions	LoC	Tests	AdtDef	Ty	PR	Pattern	TP	FP	TP/FP	TP/FP	TP/FP	TP/FP	TP/FP
napi-rs	1428	20.9k	14	724	2408	#1230	PT	1+18	4	0/0	0/5	FAIL	FAIL	FAIL
rust-rocksdb	3295	30.2k	43	503	1593	#658	OO	1+16	0	1/0	0/8	0/13	FAIL	1/0
arrow-rs	6164	149.6k	1202	1293	7452	#1878	OO	1+0	0	FAIL	FAIL	0/1	2/8	FAIL
arma-rs	179	2.2k	66	243	858	#22	OO	1+0	0	0/0	0/6	0/1	0/0	0/0
ruffle	6474	135.8k	915	11301	67092	#6528	PT	1+0	0	0/0	0/12	FAIL	FAIL	FAIL
flaco	22	0.4k	0	258	729	#12	PT	2+0	0	FAIL	0/3	0/0	0/0	FAIL
pprof-rs	110	2.3k	9	162	508	#84	PT	1+0	0	1/0	0/2	FAIL	0/0	1/0
rowan	406	4.4k	5	118	442	#112	PT	1+0	3	1/0	0/7	0/0	0/0	0/0
Fornjot	688	11.8k	40	304	13141	#646	PT	1+0	0	0/0	FAIL	0/0	0/0	1/0

Failed: FFIChecker: internal errors (self abort due to panic); Saber: the minimum Rust version supported (LLVM14) conflicts with the maximum SVF support (LLVM13); ASAN: cannot integrate LeakSanitizer into cargo tests; Miri: does not support to test `extern` FFI calls; libFuzzer: cannot generate fuzz targets through `cargo-fuzz`.

Large-Scale Experiments

- Scan 1.2k real-world Rust crates in 96 minutes (8.4s per crate)
- Find 19 crates with potential leak issues

Crate	Functions	LoC	Location	Pattern	Summary	Description
basic_dsp #52	1998	30.3k	mod.rs	PT	LeakedDropImpl	Struct `VectorBox<B,T>` leaks the proxy field *argument* in its `Drop impl`.
symsynd #15	35	0.5k	cabi.rs	PT	LeakedDropImpl	Field *message* stored a boxed slice in struct `CError` should be freed in `Drop impl`.
rustfx #3	263	4.8k	core.rs*	PT	LeakedDropImpl	Proxy field *host* in struct `OfxHost` and *String* in enum `PropertyValue`.
signal-hook #150	234	5.4k	raw.rs	OO	Overwriting	Calling `init()` multiple times will leak `AtomicPtr` in *slot* and trigger panicking.
rust-vst2 #42	207	4.1k	host.rs	OO	Overwriting	Global `LOAD_POINTER` in `call_main()` will leak `host` for multiple assignments.
synthir #1	489	7.9k	emulator.rs	OO	Overwriting	`MASKS_ALL` and `MASKS_BIT` may leak the pointed `Vec<BigUint>`.
relibc (redox-os) #180	1709	32.0k	mod.rs*	OO	Overwriting	Calling `init()` will overwrite `PTHREAD_SELF`; *packet_data_ptr* may be leaked.
tinyprof #1	26	0.5k	profiler.rs	OO	Overwriting	Assign by dereferencing to `PROFILER_STATE_SENDER` will leak the front chunck.
teaclave-sgx #441	7558	147.4k	func.rs*	OO	Overwriting	*session_ptr* and *p_ret_ql_config* may be leaked if overwrites to it.
tor_patches #1	288	4.1k	tor_log.rs	OO	Overwriting	Rewriting to `LAST_LOGGED_FUNCTION` and `LAST_LOGGED_MESSAGE` will leak box.
log #314	438	5.0k	lib.rs	OO	Finalization	`set_boxed_logger` leaks a boxed `Log` to `LOGGER` along with potential spin loop.
tracing #1517	3375	61.0k	dispatch.rs	OO	Finalization	Intentional leak towards a global `GLOBAL_DISPATCH`, the relevant issue in #1517.
rust-appveyor #1	43753	733.4k	thread_local.rs	OO	Finalization	`KEYS` and `LOCALS` need finalization function to clean up.
ServoWsgi	11606	236.2k	opts.rs	OO	Lazy	`DEFAULT_OPTIONS` needs finalization to clean up, especially for multi threads exit.
rux	585	9.9k	lazy.rs	OO	Lazy	The current lazy static uses the *'static* that cannot be freed in multi threads.
next_space_coop	20477	317.1k	lazy.rs	OO	Lazy	The current lazy static uses the *'static* that cannot be freed in multi threads.
rio #52	95	3.0k	lazy.rs	OO	PanicPath	If the assertion goes into unwinding, the *value* will be leaked inside a panic.
sled #1458	1350	32.8k	lazy.rs	OO	PanicPath	If the assertion goes into unwinding, the *value* will be leaked inside a panic.

The item having * indicates vulnerabilities located in multiple files. Additionally, we did not open an issue specifically for scenario `Lazy`, due to it being a common design flaw in lazy_static implementations.

Thanks

Q & A