Towards Reliable OS: Rust, Design, or Verification?

Hui Xu

Fudan University

2024-06-23

Outline

I. Problem

II. Pros and Cons of Rust

III. Novel Rust OS Design

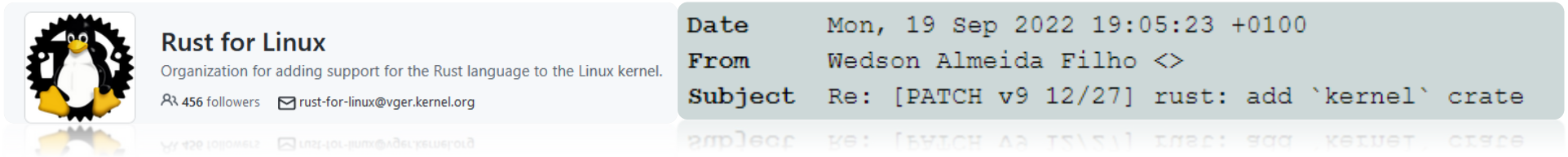
IV. Verification Techniques

V. Summary

I. Problem

Can Rust enable more reliable OS?

When Linux meets Rust...



Wedson A. Filho

*"We generally have **two routes to avoid undefined behavior**: detect at **compile time** (and fail compilation) or at **runtime**..."*

Linus Torvalds

*"**If you can't deal with the rules that the kernel requires, then just don't do kernel programming**. Because in the end it really is that simple. I really need you to understand that **Rust in the kernel is dependent on *kernel* rules**. Not some other random rules that exist elsewhere."*

https://lkml.org/lkml/2022/9/19/1105#1105.php

Eye-Catching Headlines of CVEs “related to” Rust

关于Rust命令注入漏洞(CVE-2024-24576)的安全预警-东南大...

2024年4月11日 Rust标准库中存在命令注入漏洞(CVE-2024-24576,被称为BatBadBut),该漏洞的CVSS评分为10.0,可能在Windows系统上导致命令注入攻击,目前该漏洞的细节已公开。 Rus...

东南大学网络与信息中心

别用Rust了?Win7/8/10系统中发现高危漏洞

2024年4月10日 近日，安全专家发现了一个名为CVE-2024-24576的漏洞。这个漏洞存在于使用Rust编程语言开发的软件中，允许攻击者对Windows系统进行命令注入攻击。该漏洞是由于操作系统命令和参数注...

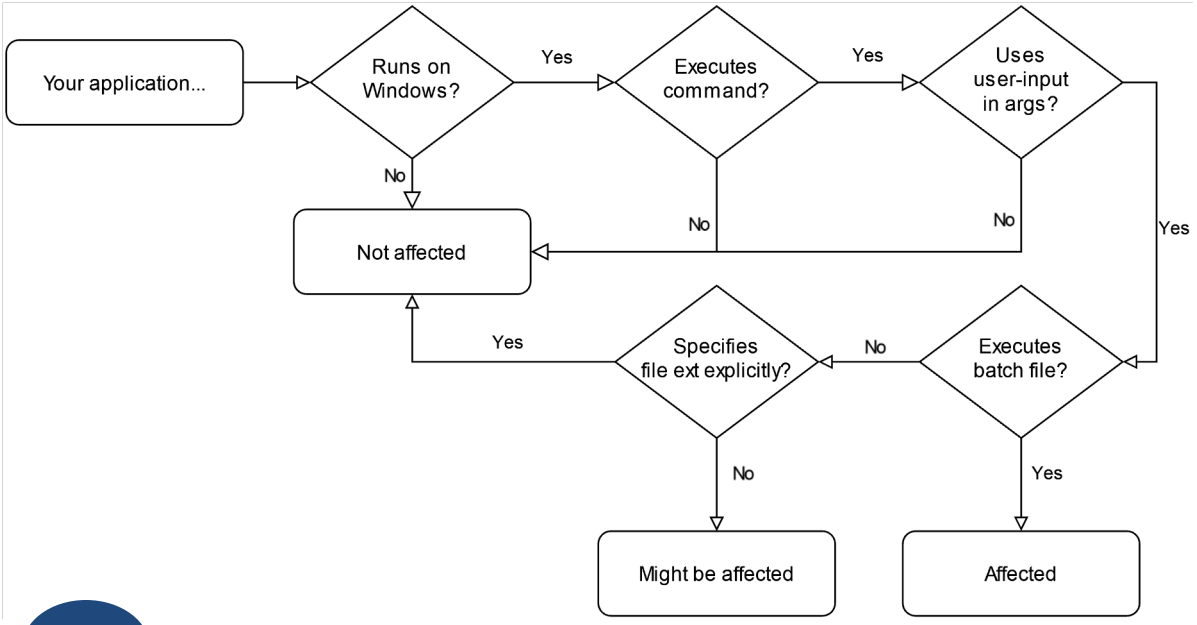
中关村在线

Flatt SECURITY

BatBadBut: You can't securely execute commands on Windows

RyotaK

Truth:
A Windows issue that affects all languages.




```
use std::process::Command;

Command::new("cmd.exe")
    .args(["escape letter", "&calc.exe"])
    .spawn()
    .expect("command failed to start");
```

Users may inject new command via escape letters for cmd.

https://flatt.tech/research/posts/batbadbut-you-cant-securely-execute-commands-on-windows/

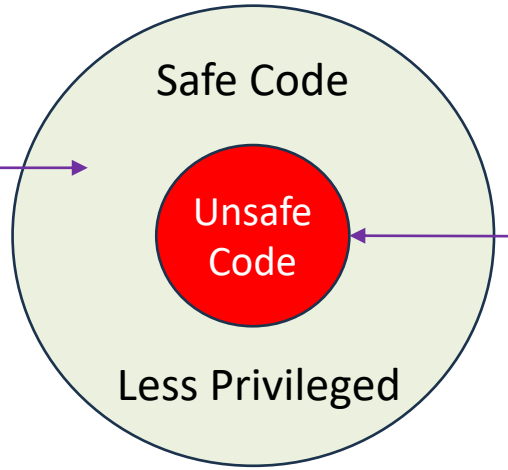
II. Pros and Cons of Rust

Idea of Rust for Security: Security Zone

Rust: Code Privilege Model

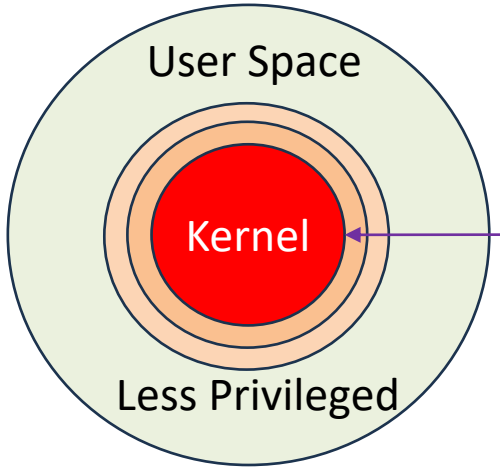
Security objectives of Rust:

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



Interior unsafe: encapsulate privileged code within safe APIs

Operating System: Ring Model



System Call

Safe Rust (Ownership Scheme) ≈ C++ with Enforced Intelligent Pointers

- Each object is owned by one variable
- Ownership can be moved or borrowed (immutable/mutable)
- Exclusive mutability: an object cannot be mutable and shared at one program point

```
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);
```

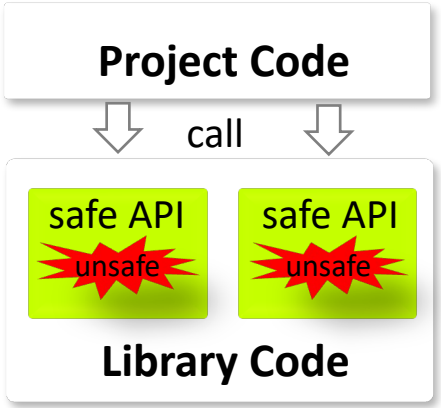
Bob borrows the ownership

Bob is dead; return the ownership

(Unsafe Rust ≈ C) => Interior Unsafe is the Key

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

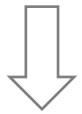
```
impl<T> Vec<T> {
    //safe API encapsulation
    pub fn push(&mut self, value: T) {
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);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



Example from Rust std-lib

Low-level Control: Memory-Mapped IO

```
let vga_buffer = 0xb8000 as *mut u8;
for (i, &byte) in HELLO.iter().enumerate() {
    unsafe {
        *vga_buffer.offset(i as isize * 2) = byte;
        *vga_buffer.offset(i as isize * 2 + 1) = 0xb;
    }
}
```



```
lazy_static! {
    pub static ref WRITER: Writer = Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };
}
```


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

III. Novel Rust OS Design

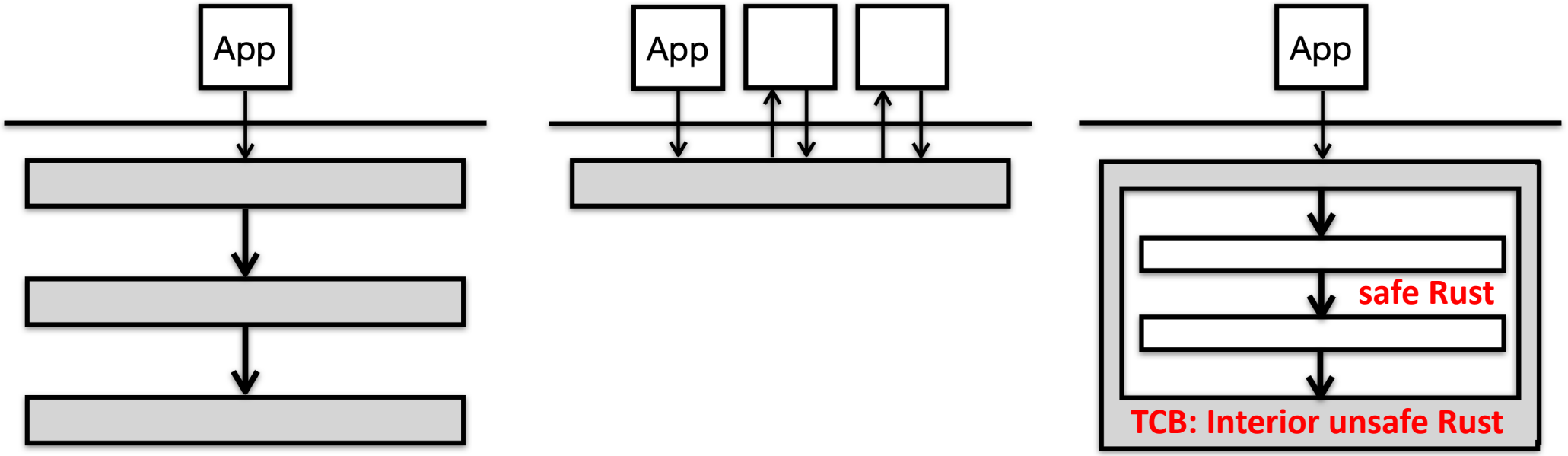
Theseus

Asterinas

Asterinas: Forbid Unsafe Code via Framekernel

蚂蚁集团 ANT GROUP

- Kernel services: developed with safe Rust only
- Framekernel: provides a TCB (apis) with interior unsafe code



(a) A monolithic kernel (e.g., Linux)

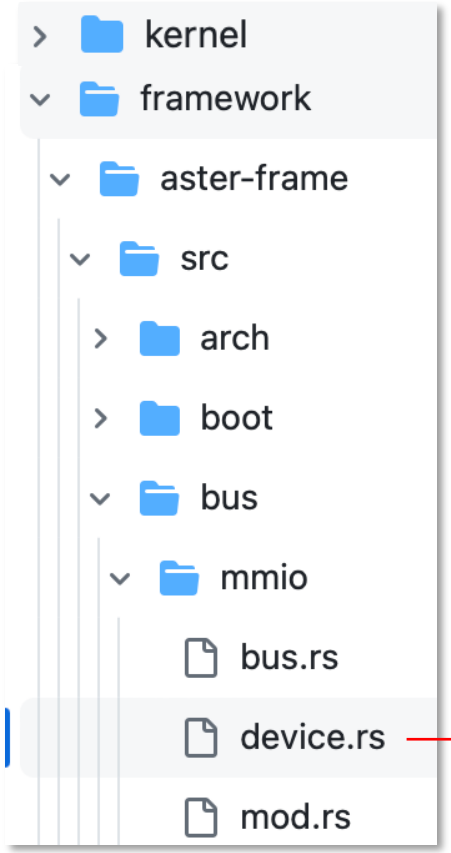
(b) A microkernel (e.g., seL4)

(c) A framekernel (e.g., Asterinas)

https://asterinas.github.io/book/kernel/the-framekernel-architecture.html

Asterinas

Example: Memory-Mapped IO




```rust
impl IoMem {
    pub(crate) unsafe fn new(range: Range<Paddr>) -> IoMem {
        IoMem { virtual_address: paddr_to_vaddr(range.start), limit: range.len(), }
    }
}
impl VmIo for IoMem {
    fn write_bytes(&self, offset: usize, buf: &[u8]) -> crate::Result<()> {
        self.check_range(offset, buf.len())?;
        unsafe { core::ptr::copy(...); }
        Ok(())
    }
}
```

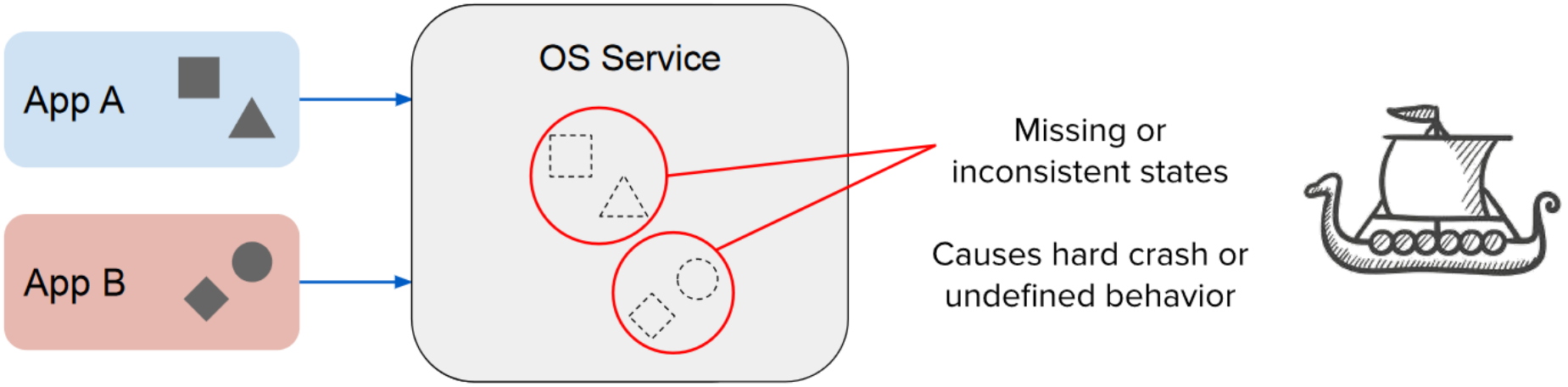
```rust
impl MmioCommonDevice {
    pub(super) fn new(paddr: Paddr, handle: IrqLine) -> Self {
        ...
        // SAFETY: This range is virtio-mmio device space.
        let io_mem = unsafe { IoMem::new(paddr..paddr + 0x200) };
        let res = Self { io_mem,  irq: handle, };
        res
    }
}
```

? Is this interior unsafe method sound?

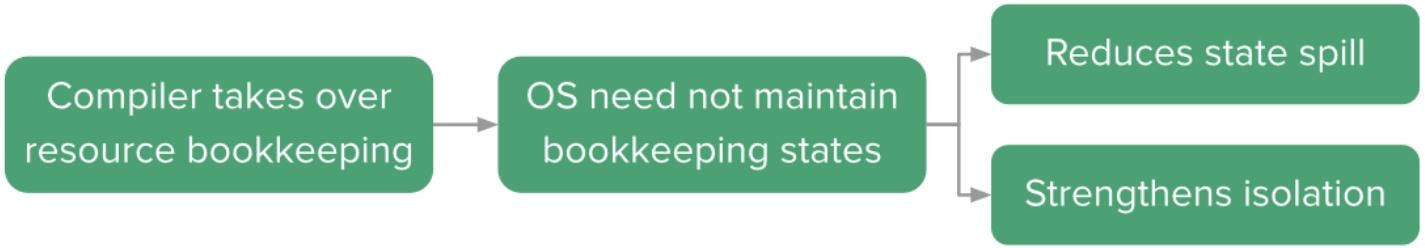
Probably yes. Leveraging visibility, kernel cannot directly access the function.

Theseus: Intralingual Approach to Enforce Invariants about OS Semantics

- To mitigate the faults of state spill: *e.g.,* process management, inter-entity collaborations



- Characteristics of Theseus OS:
  - Single address space
  - Single privilege level
  - Single allocator instance

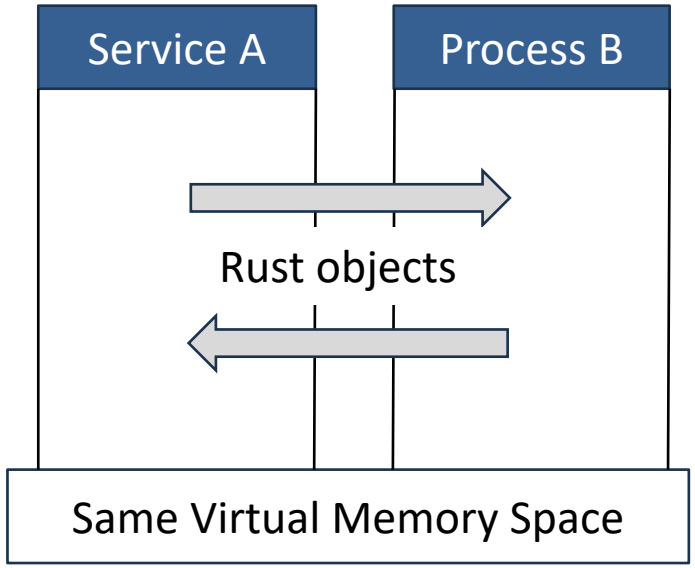


Yale FUTUREWEI Technologies

Kevin Boos, *et al*. "A characterization of state spill in modern operating systems." EuroSys. 2017.
Kevin Boos, *et al*. "Theseus: an experiment in operating system structure and state management." *OSDI* 2020.

Example: Task Management

- Multi-tasking: similar as multi-threading
- Server can safely relinquish its state to client



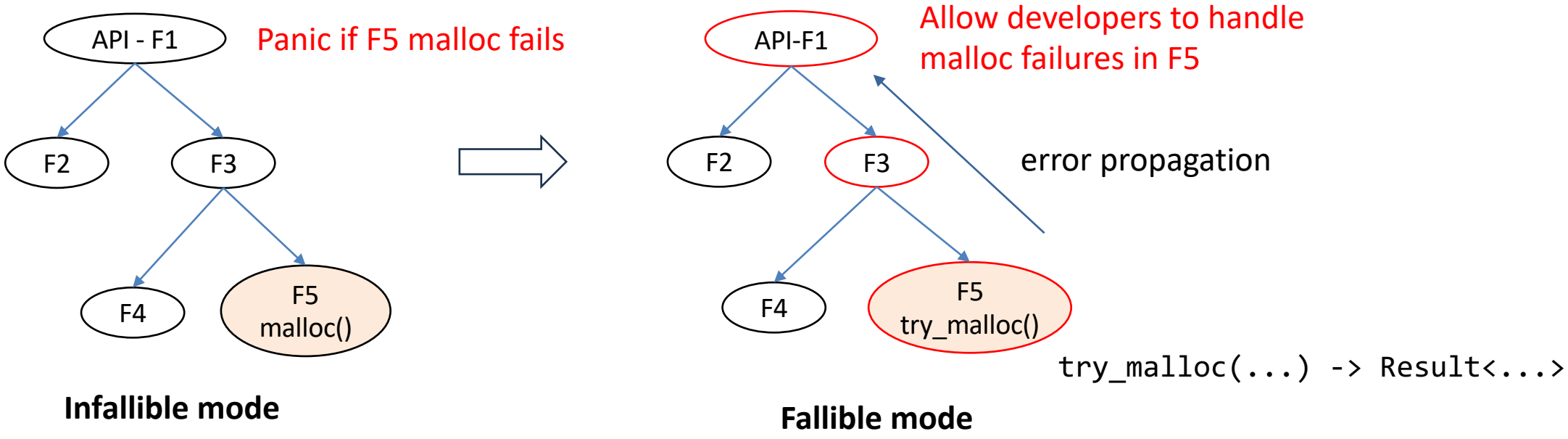

```
pub fn create(task: Task,
              cleanup: FailureCleanupFunction)
       -> JoinableTaskRef
```

```
pub struct Task {
    pub id: usize,
    pub name: String,
    pub mmi: Arc<Mutex<MemoryManagementInfo,
                       DisableIrq>, Global>,
    pub is_an_idle_task: bool,
    pub app_crate: Option<Arc<AppCrateRef, Global>>,
    pub namespace: Arc<CrateNamespace, Global>,
    /* private fields */
    ...,
}
```


https://www.theseus-os.com/Theseus/book/subsystems/task.html

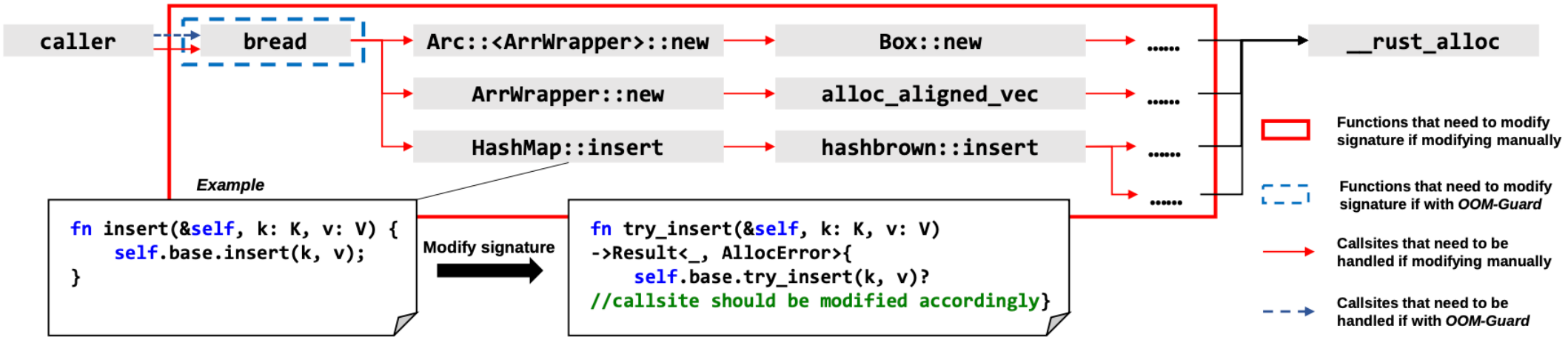
Our Efforts to Ease Out-Of-Memory Handling: OOM-Guard

- Rust employs infallible mode by default
- Switching to falliable mode (nightly Rust) requires much exception handling efforts
- OOM-Guard:
  - Reserve a large enough heap space (prediction) by the top-level API
  - Subsequent allocations reusing the space would not fail



Chengjun Chen, *et al.* OOM-Guard: Towards Improving the Ergonomics of Rust OOM Handling via a Reservation-Based Approach, FSE, 2023

OOM-Guard: Demonstration of Usage



(a) The usability comparison between OOM-Guard and existing fallible mode.

```
+ impl From<TryReserveError> for BentoError{
+   fn from(err: TryReserveError) -> BentoError {
+       BentoError::alloc_fail(String::from("..Error_Message"))
+      //need allocation, second OOM may occur
+   }
+ }
fn bread(&self, bno: u64) -> Result<BufferHead, BentoError> {
    ...//allocation-free instructions
    - let bh_buf = ArrWrapper::new(...)?; //allocation
    - let new_arc = Arc::new(bh_buf); //allocation
    - cache_lock.insert(bno, Arc::downgrade(&new_arc));
    //cache_lock is a Hashmap and need allocation when expending
    + let bh_buf = ArrWrapper::try_new(...)?;
    + let new_arc = Arc::try_new(bh_buf)?;
    + cache_lock.try_insert(bno, Arc::downgrade(&new_arc))?;
    return Ok(BufferHead::new(new_arc, bno));
}
```

(b) Manual modification with fallible APIs.

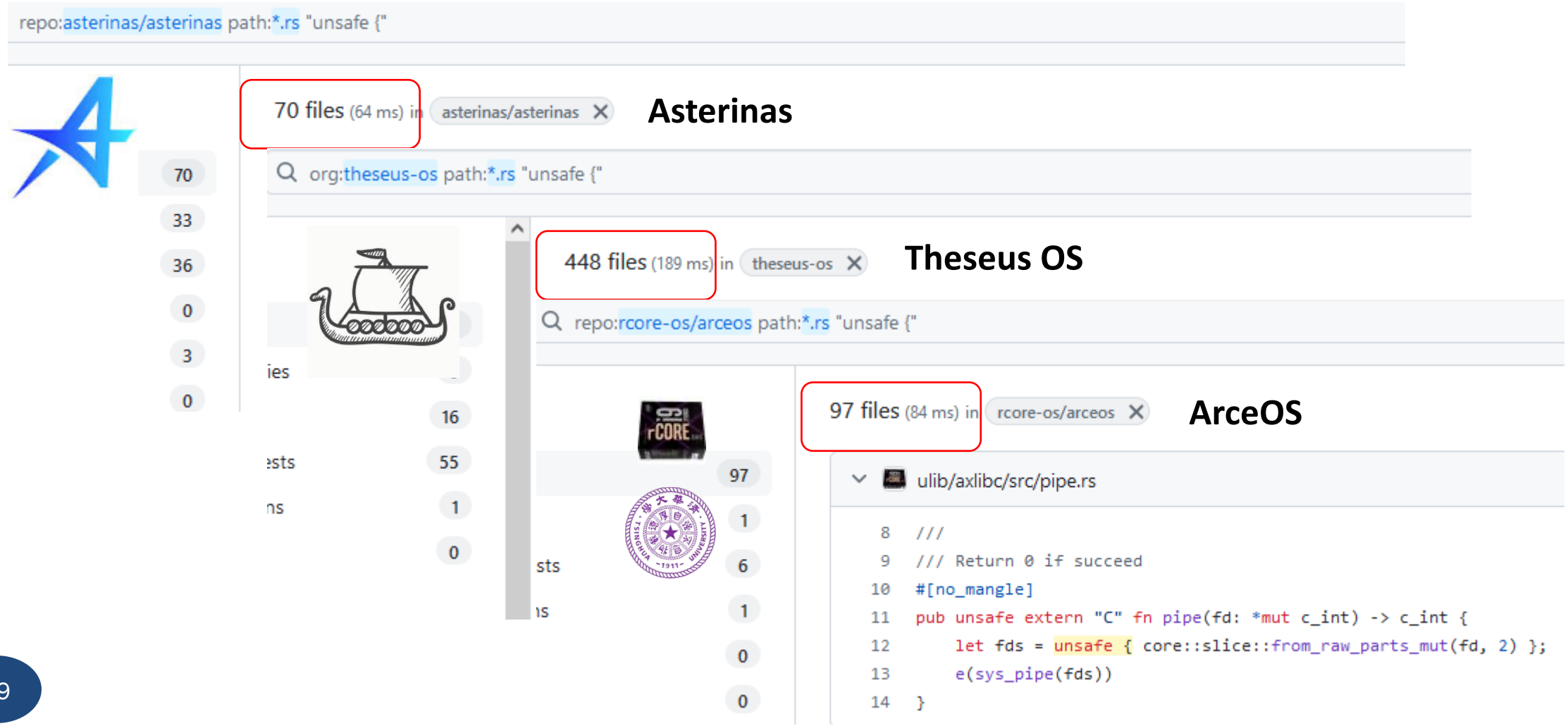
```
#[global_allocator]
pub static ALLOCATOR = OOMGuardAllocator::new(&DefaultAllocator);

#[oom_guard]
fn bread(&self, bno: u64) -> Result<BufferHead, BentoError> {
    + ...calculative statements;
    + let reserve_array = [...];
    + let guard_life_time = ALLOCATOR.reserve(&reserve_array)?;
    //automatically generated during macro expansion
    ...//allocation-free instructions
    let bh_buf = ArrWrapper::new(...)?;
    let new_arc = Arc::new(bh_buf);
    cache_lock.insert(bno, Arc::downgrade(&new_arc));
    return Ok(BufferHead::new(new_arc, bno));
}
```

(c) Modification with OOM-Guard.

IV. Verification Techniques

Key Problem: Soundness Verification of Interior Unsafe Code

- Interior unsafe is an advocated paradigm in system software development with Rust.
- How to verify the soundness of interior unsafe code? Either by human or automated.



Verification Techniques

	De/Inductive Verification	Model Checking	Static Analysis/ Dynamic Analysis
Specification	Theorem Functional Correctness	Contract Properties	
Proof	**Manual** function code + proof code	**Automated** abstract interpretation /symbolic execution	Alias analysis /lattice-based /pattern-based /…
Theorem Provers	**Interactive** HOL/Isabella/Iris/Coq	**Automatic** CVC/Z3	
Example Work	seL4/RustBelt	Kani/Prusti/Verus/RustHorn	Rudra/SafeDrop/Semgrep

Lightweight Formal Method

Verus: Model Checker

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

- Usage: automated + require contract annotations (oracle)
- Not directly applicable for OS verification, especially the soundness of using unsafe code
- Limitations: feature/precision issues (*e.g.*, heap modeling, loop handling)

```
verus! {
    fn octuple(x1: i8) -> (x8: i8)
        requires -16 <= x1 < 16,
        ensures x8 == 8 * x1,
    {
        let x2 = x1 + x1;
        let x4 = x2 + x2;
        x4 + x4
    }
    fn main() {
        let n = octuple(10);
        assert(n == 80);
    }
}
```

Contract: precondition

Contract: postcondition

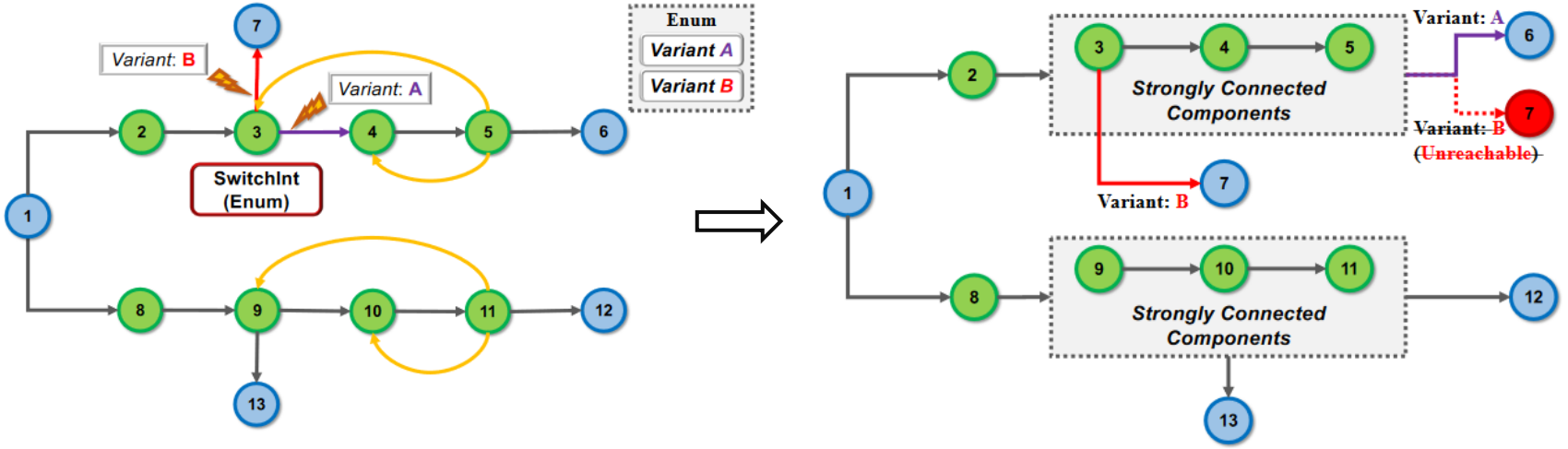
https://github.com/verus-lang/verus

Andrea Lattuada, *et al.* "Verus: Verifying rust programs using linear ghost types." OOPSLA, 2023.

SafeDrop: Static Analysis for Dangling Pointer Bug Detection

- Limitations: do not support other UBs; false positives

1. Path Extraction



Control-flow Graph

Spanning Tree

2. Alias Analysis

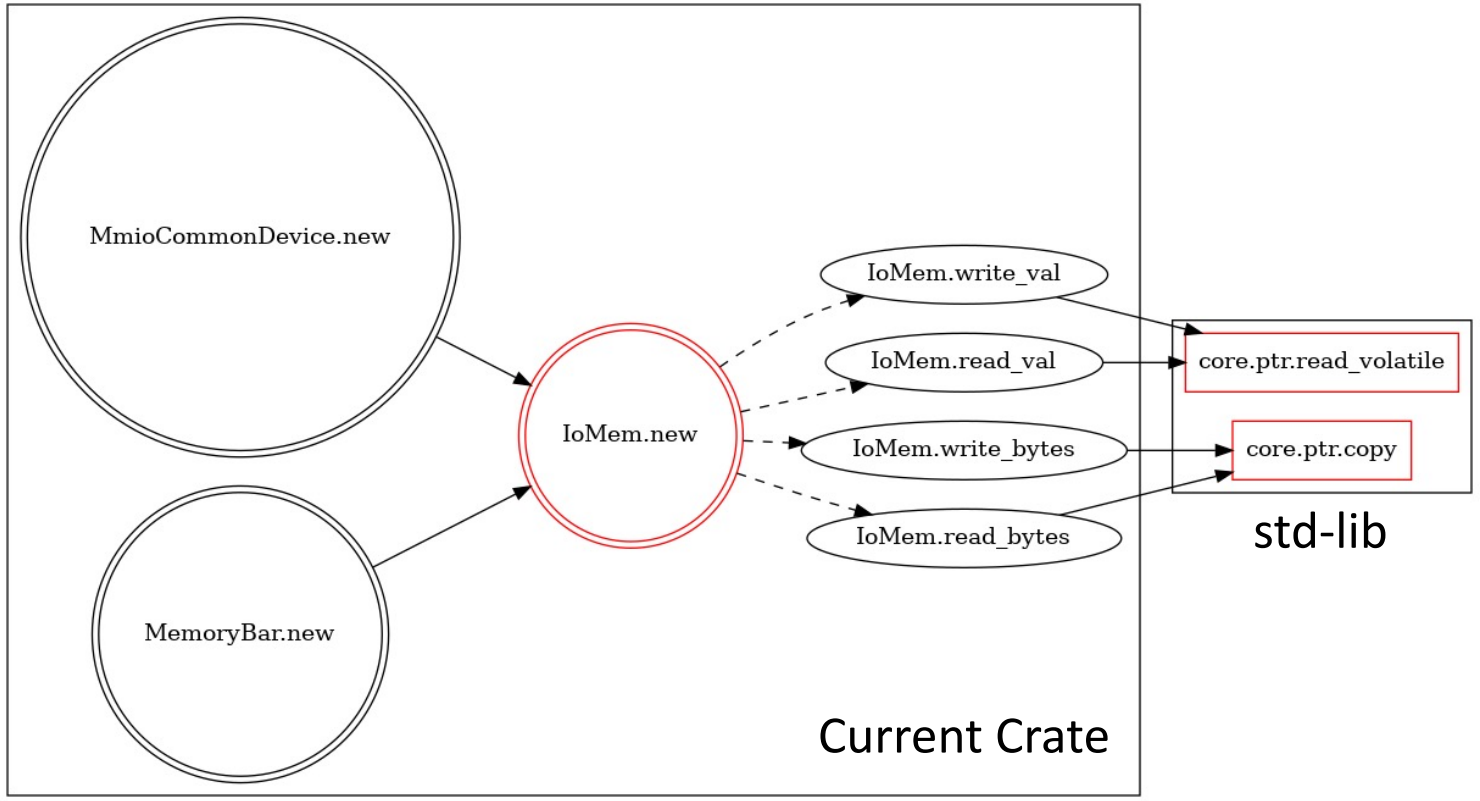
3. Pattern Detection

```
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}
```

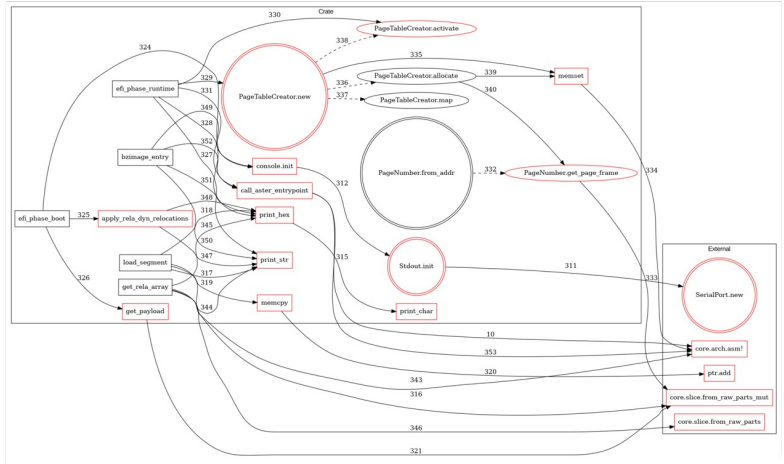

Mohan Cui, *et al*. "SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis." TOSEM. 2023.
https://github.com/Artisan-Lab/RAP

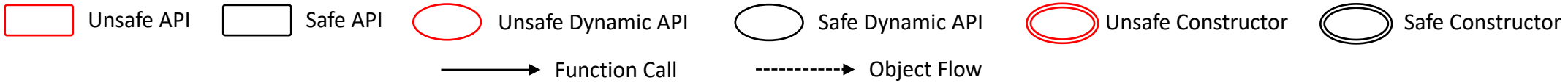
Real-world Rust Project Verification: Track Unsafety Propagations

Unsafety Isolation Graph: MMIO Example from Asterinas



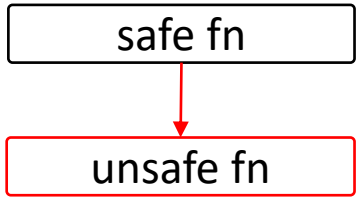
The graph could be huge



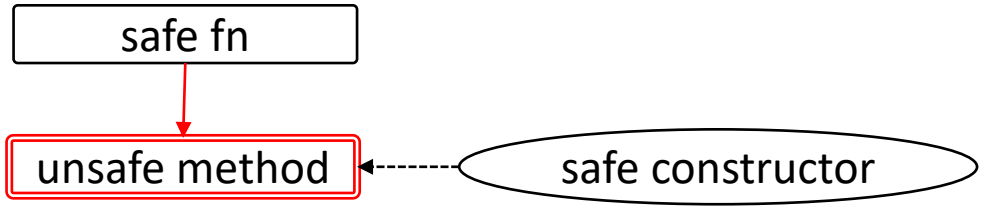


Zihao Rao, *et al*. "Characterizing Unsafe Code Encapsulation In Real-world Rust Systems." *arXiv preprint* (2024).

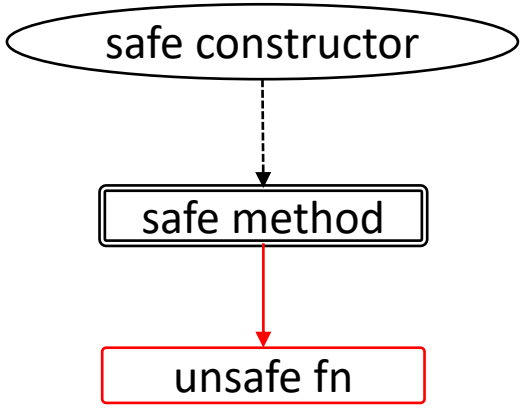
Split the Graph into Small Audit Units based on Patterns



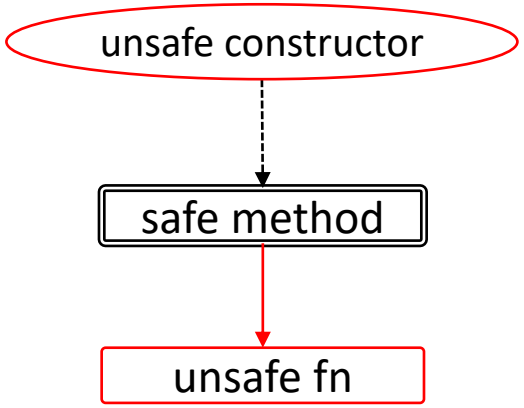
$RS_{uf} \in VS_{sf}$



$RS_{um} \in VS_{sf}$



$RS_{uf} \in VS_{sm} + VS_{sc}$



$RS_{uf} \in VS_{sm} + RS_{uc} + VS_{uc}$

RS: Required Safety Property
VS: Verified Safety Property

V. Summary

Summary

- Safe Rust ≈ C++ with enforced intelligent pointers
- The magic of Rust lies in interior unsafe or unsafe code encapsulation
- Possible benefits for Rust towards reliable OS:
  - Asterinas: forbid unsafe code via framekernel
  - Theseus: intralingual approach to enforce invariants about OS semantics
- Verification for interior unsafe code is critical for achieving reliable Rust OS

Thanks! Q&A