GOSIM®
GLOBAL OPEN-SOURCE INNOVATION MEETUP

FIGHTING THE HEAP WAR WITH RUST

Hui Xu
School of Computer Science
Fudan University

Outline

i. Research Background
ii. Dangling Pointer Detection
iii. Memory Leakage Detection
iv. Heap Exhaustion Handling
v. Summary

1. Research Background

Rust Ownership and Limitations

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	Improper Neutraliz
17	CWE-119	Improper Restrictio
18	CWE-798	Use of Hard-coded Credentials
19	CWE-918	Server-Side Reques
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

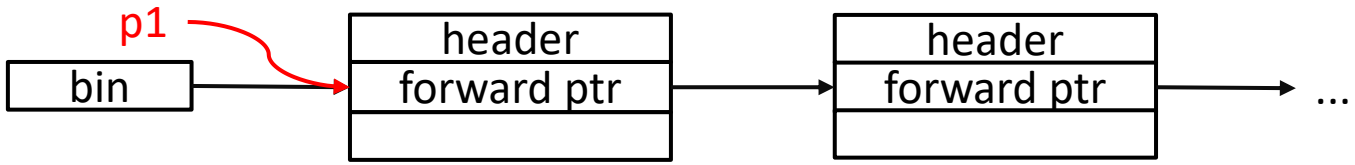
Which vulnerabilities can be mitigated through language design?

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

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

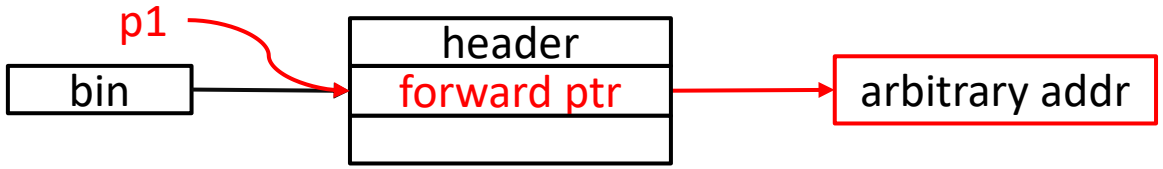
Why Heap Bugs are Dangerous? UAF as an Example

1. free(p1)



Add the block of *p1 to the free list

2. write(p1)

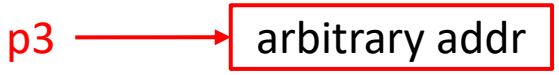


Attackers modify the forward pointer through p1

3. p2 = malloc()

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

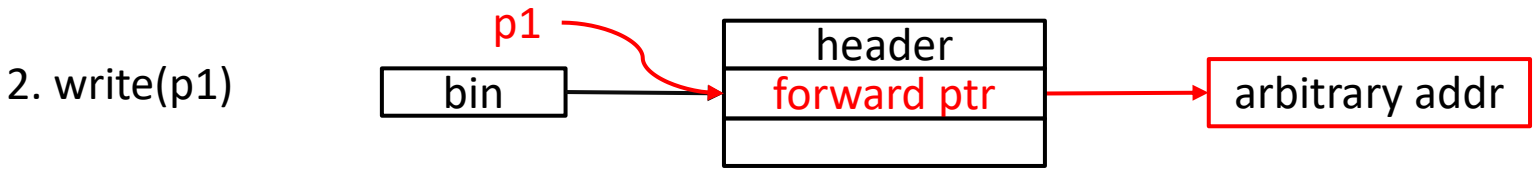
4. p3 = malloc()



p3 points to the arbitrary address

Detecting UAF is Hard: via Allocator Design

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



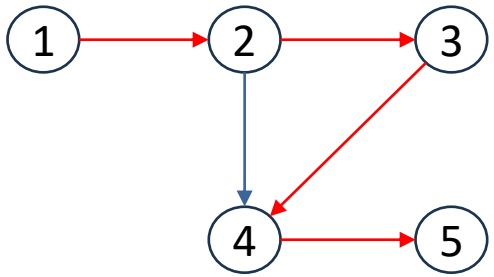
Problem: incur overhead during each pointer access

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

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
```

May-Alias Analysis

****v1 = v5 ?

  - 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

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?

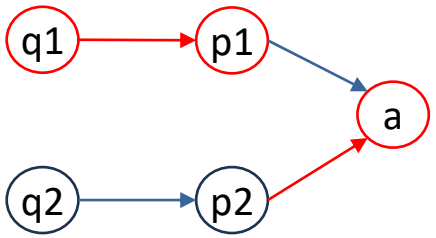
Why the Approach is Efficient?

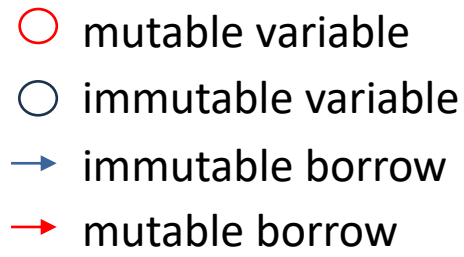
- Compute the 'minimum' liveness of each variable
- Avoid hard alias-analysis problems
  - No need to track multi-level aliases
  - The mutability does not propagate

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

live variables

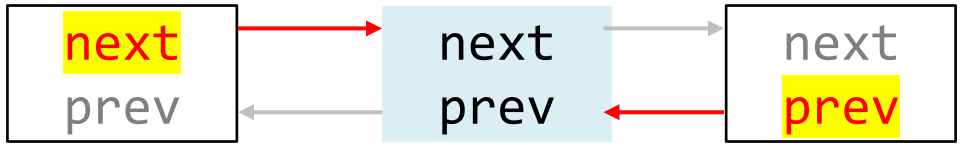
{a}
{a, p1}
{a, p1, p2}
{a,p2}





Limitations of Ownership

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




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

Option 1: Shared Pointer
(with runtime cost)

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

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

Empirical Study of Memory-Safety Bugs in Rust Projects

- 185 bugs reported before 2020-12-31 (all CVEs/Advisory DB + GitHub)
- 35/185 bugs involve bad drop issues caused by unsafe code
- Memory leakage bugs are not included (not memory-safety issues)

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

"Memory-safety challenge considered solved? An in-depth study with all Rust CVEs", TOSEM, 2022.

Our Research Efforts: Fight the Heap War with Rust

- Dangling pointer detection: use-after-free, double free

- Memory leakage detection: the opposite of dangling pointer

- Design a better way to handle heap exhaustion risks

2. Dangling Pointer Detection

SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis, *TOSEM*, 2022.

Motivating Example: Dangling Pointer

```
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.l
        //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

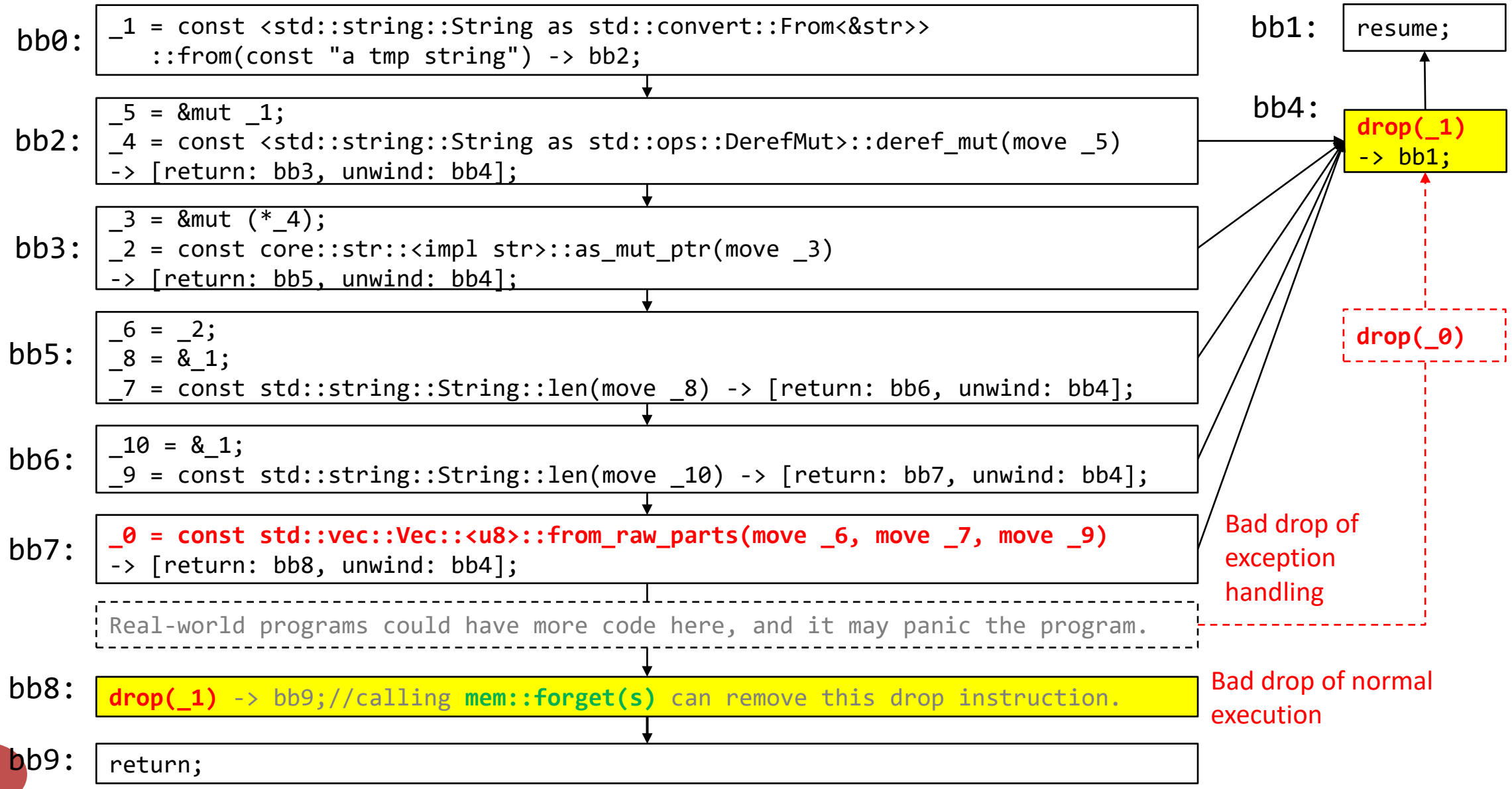
create a Vec v pointing to s via unsafe

return v

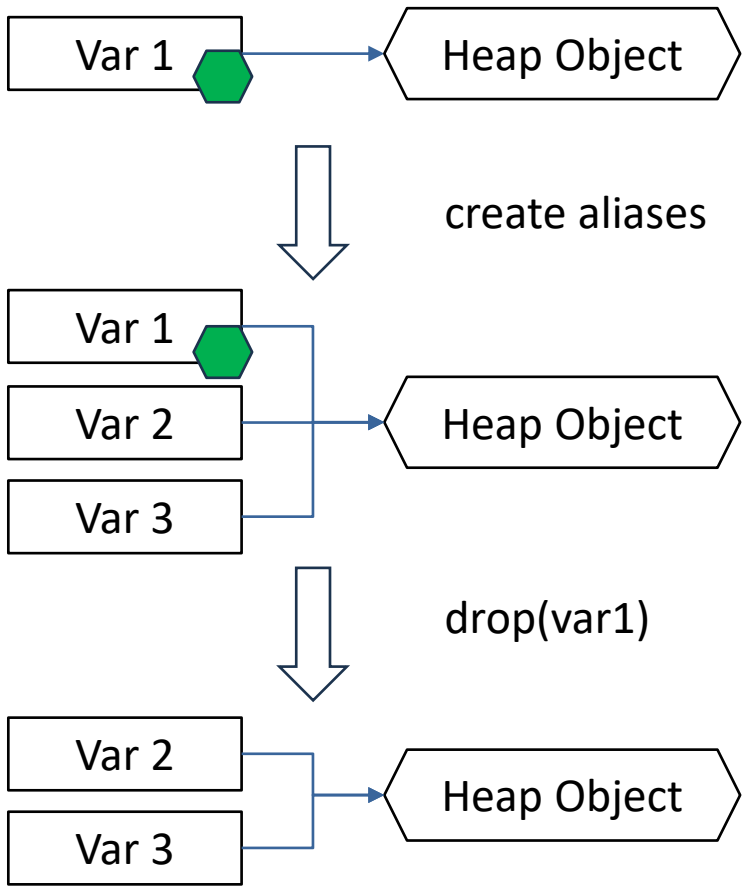
drop s; v becomes dangling

accessing v causes use-after-free

Bug Analysis with Rust MIR

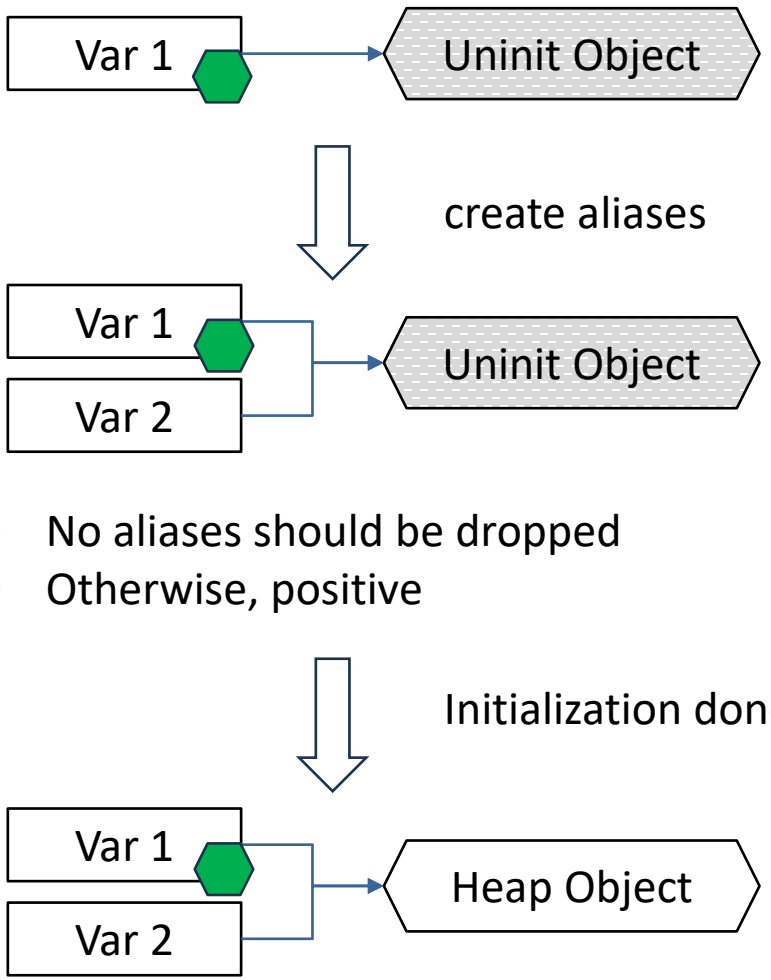


Abstraction of Bug Patterns



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

ownership



- No aliases should be dropped
- Otherwise, positive

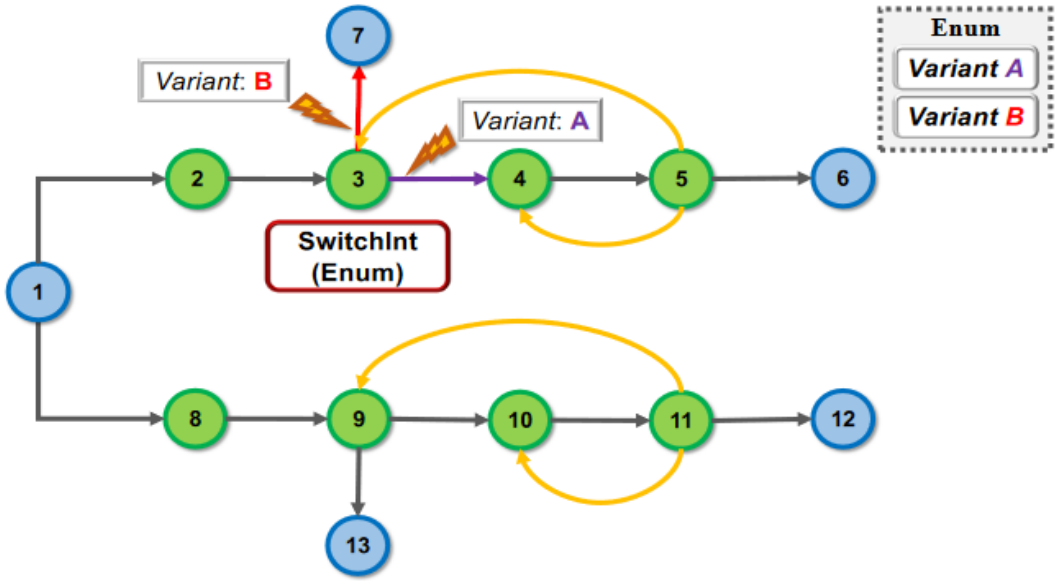
Approach for Dangling Pointer Bug Detection

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

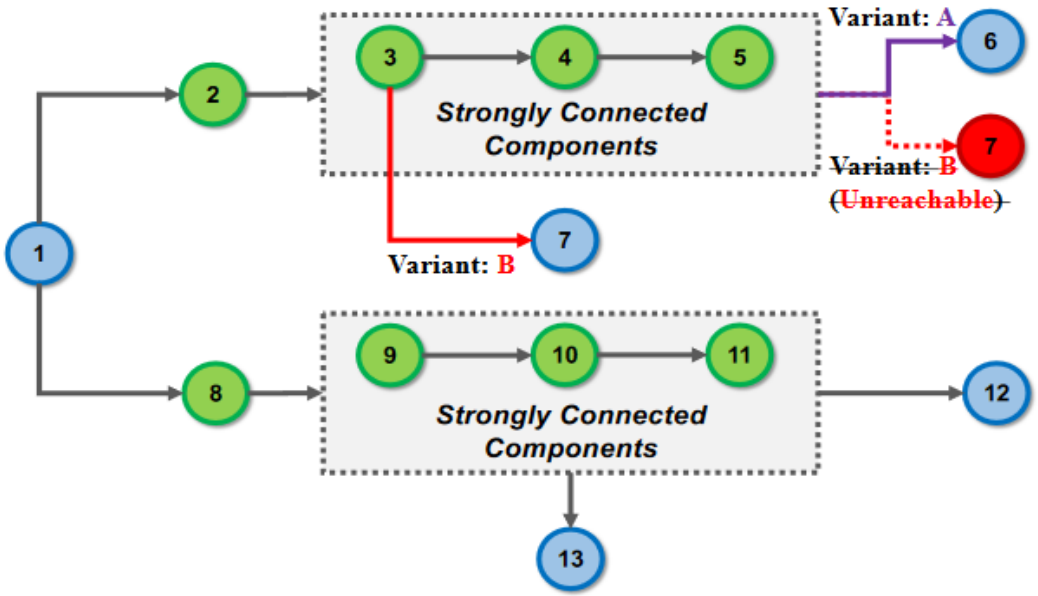
Approach:

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

- 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

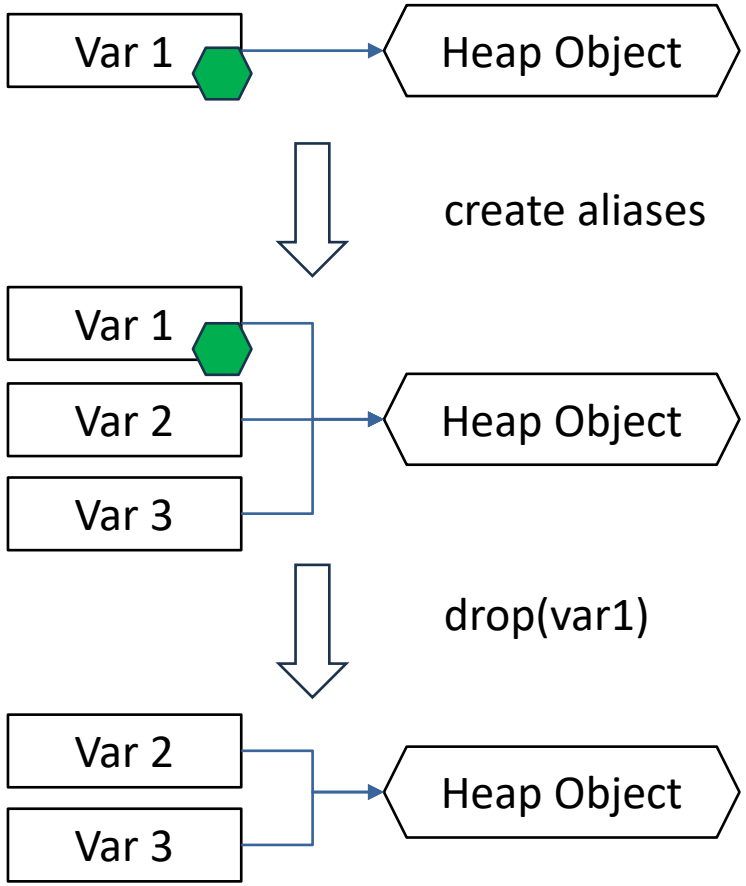
update

{0.0.0, 1}

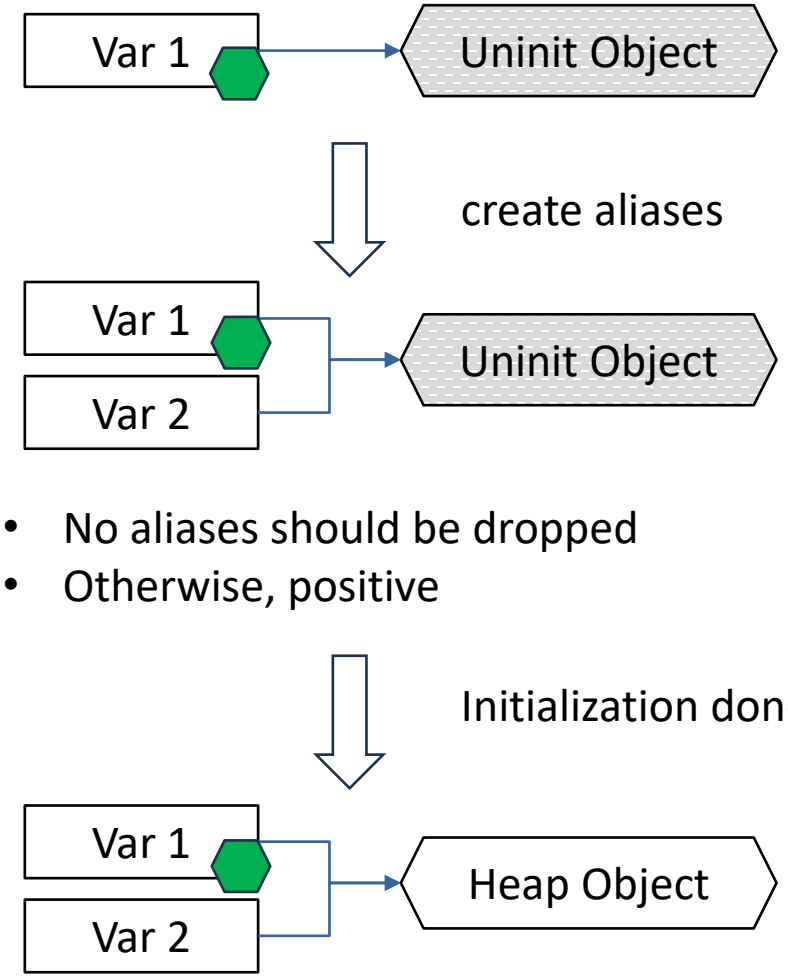
inter-procedural

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

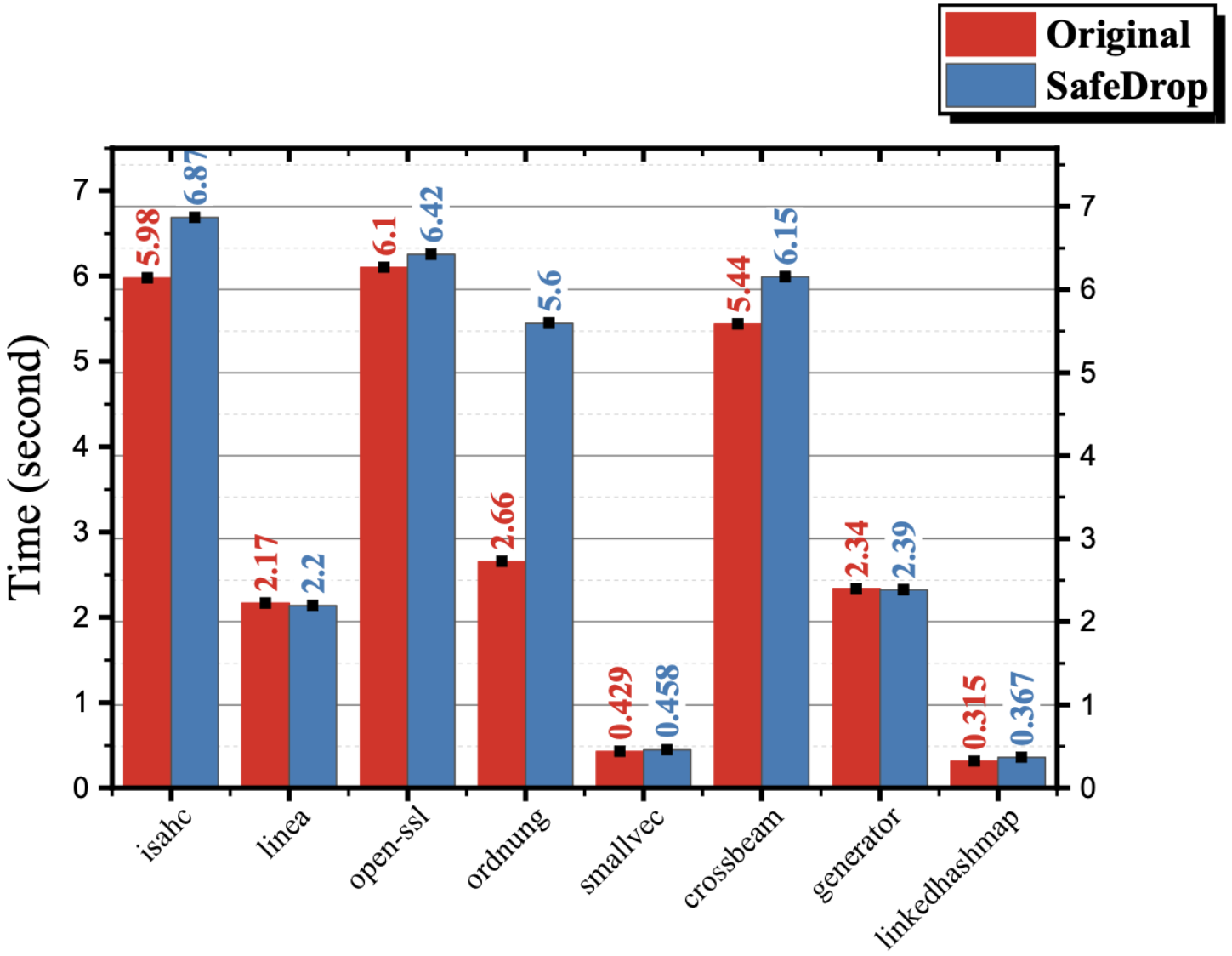
Experimental Results

- 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



3. Memory Leakage Detection

rCanary: Detecting memory leaks across semi-automated memory management boundary in Rust, *arXiv*, 2023.

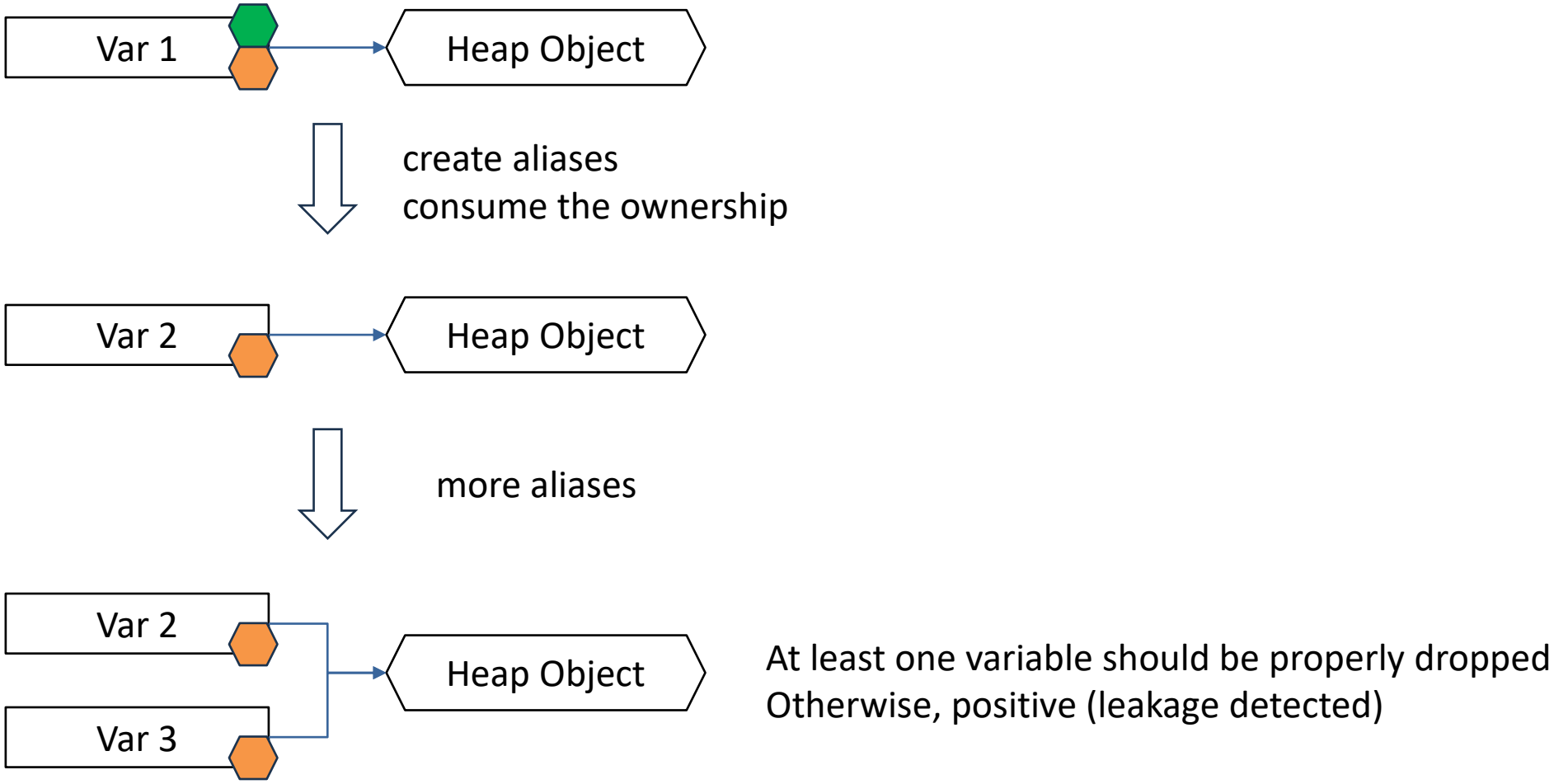
Memory Leakage Problem

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

```
    let mut buf = Box::new("123");
    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

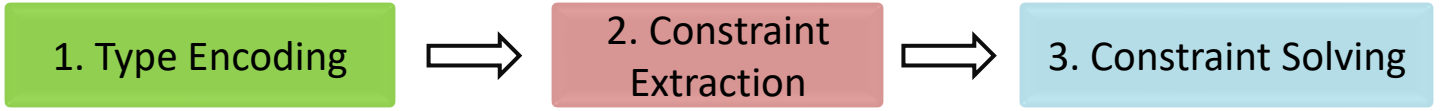
```
pub struct WString { ptr: NonNull<WStr>, capacity: usize, }
impl WString {
    unsafe fn steal_buf(&mut self)
                -> ManuallyDrop<Units<Vec<u8>, Vec<u16>>> { ... }
}
impl Drop for WString {
    fn drop(&mut self) {
-       let _ = unsafe { self.steal_buf() };
        unsafe {
+      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

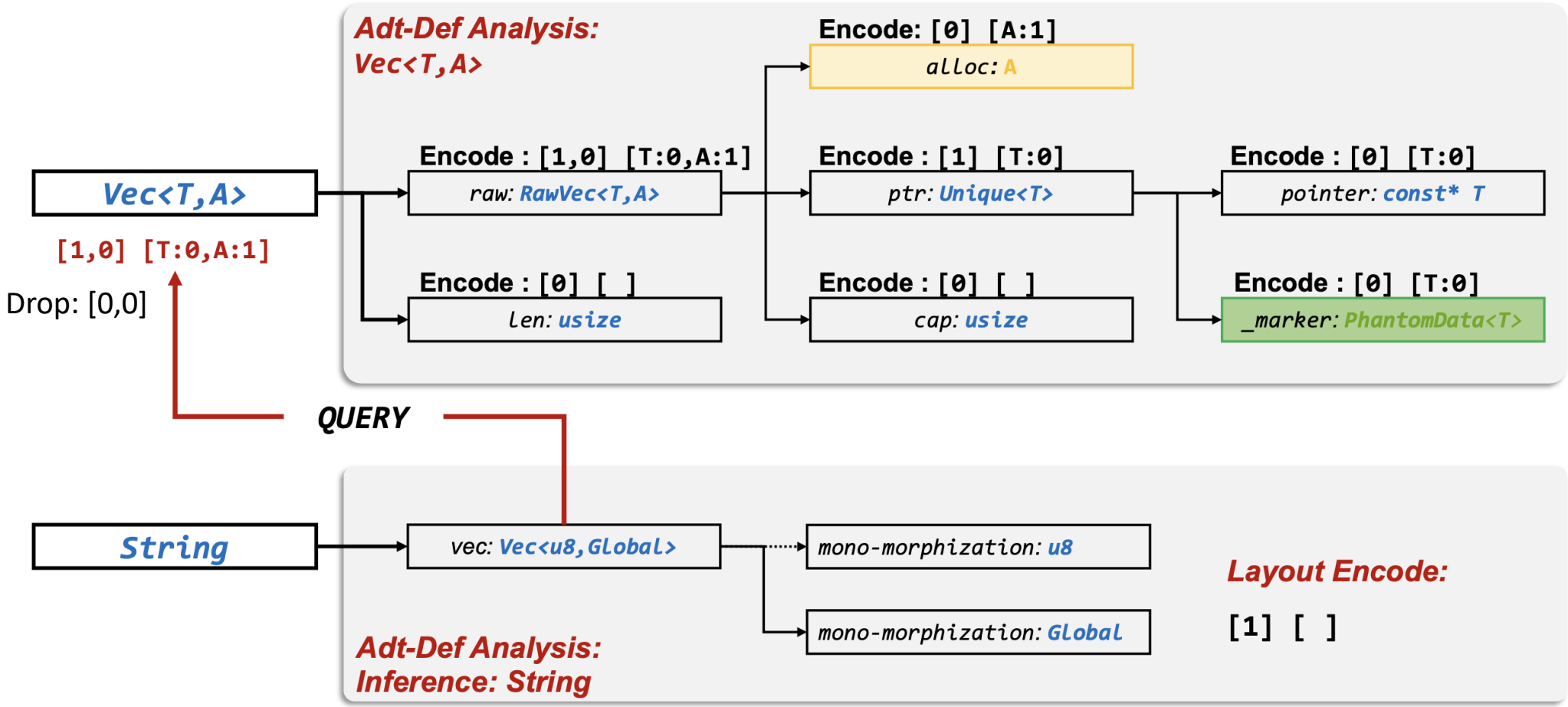
- Type encoding: abstract data types wrt heap resource holding
- Constraint extraction: path-insensitive data-flow analysis
  - fast, less false positives
- Constraint solving: based on Z3



Approach

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

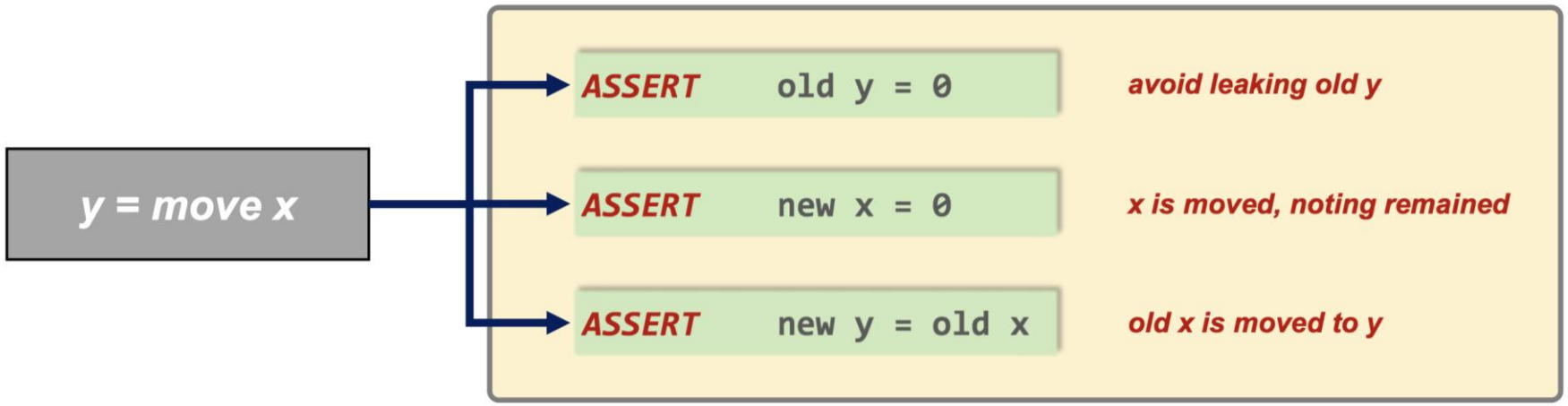
- Encode each type as a one-level bit vector (field-sensitive)
- Also encode the corresponding drop function as a one-level bit vector



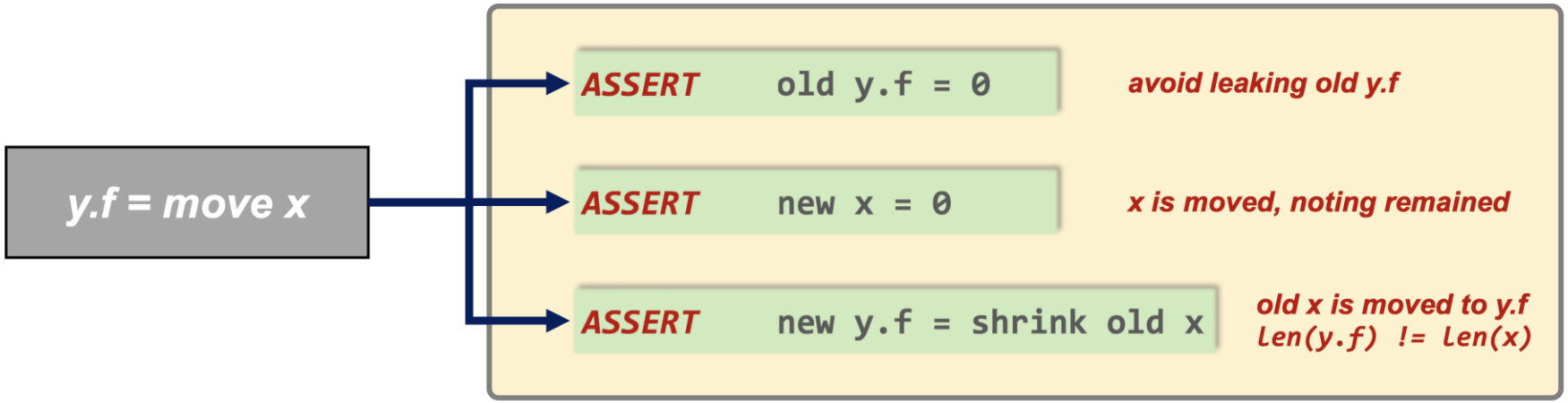
Approach

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.

4. Heap Exhaustion Handling

OOM-Guard: Towards improving the ergonomics of Rust OOM handling via a reservation-based approach, *FSE*, 2023.

Problem of Out-Of-Memory

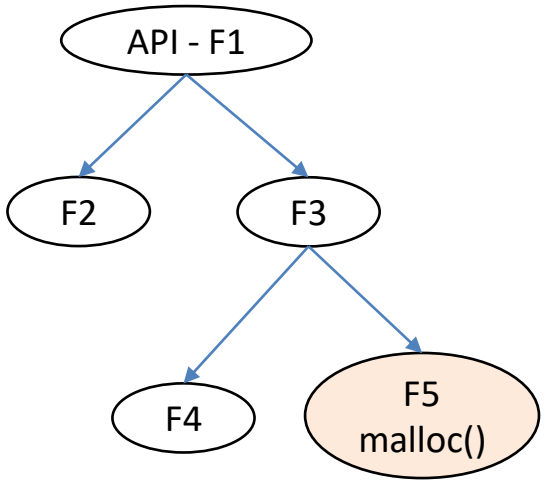
- Rust adopts an infallible mode:
  - No way for developers to handle OOM
  - Terminate the process if OOM
- Problem: many APIs involve heap allocations underneath

```
pub fn sort(&mut self)                                    source
where
    T: Ord,
```

Current implementation

The current algorithm is an adaptive, iterative merge sort inspired by timsort. It is designed to be very fast in cases where the slice is nearly sorted, or consists of two or more sorted sequences concatenated one after another.

Also, it allocates temporary storage half the size of `self`, but for short slices a non-allocating insertion sort is used instead.

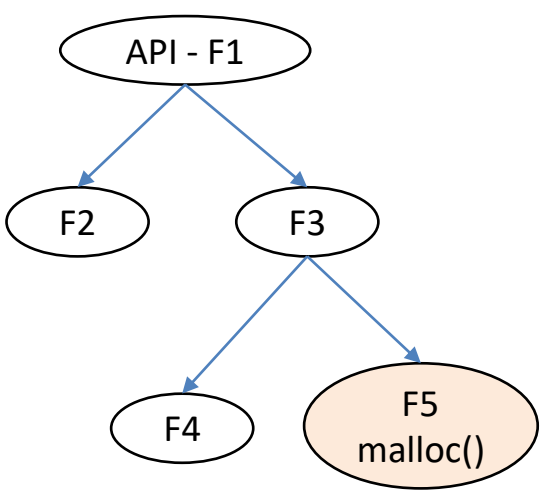


Sample Call Graph

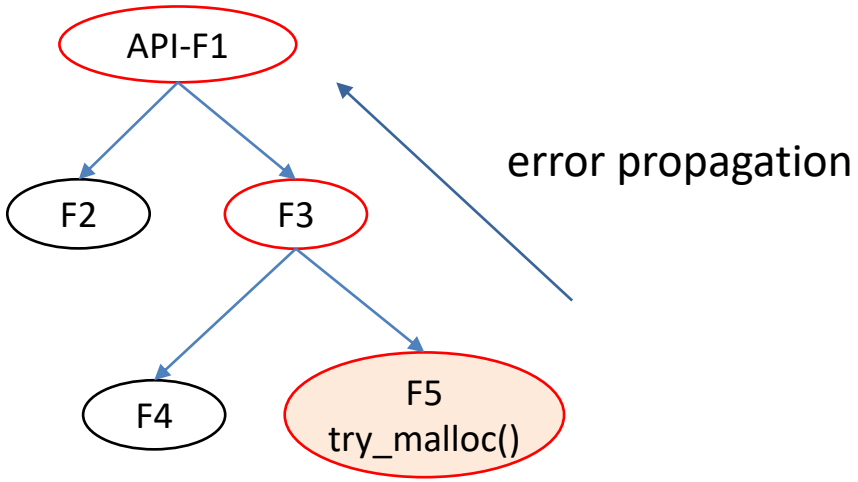
https://doc.rust-lang.org/std/primitive.slice.html#method.sort

Fallible Mode in Nightly Rust

- Enable developers to handle allocation failures
- Require much programming efforts



Infallible mode



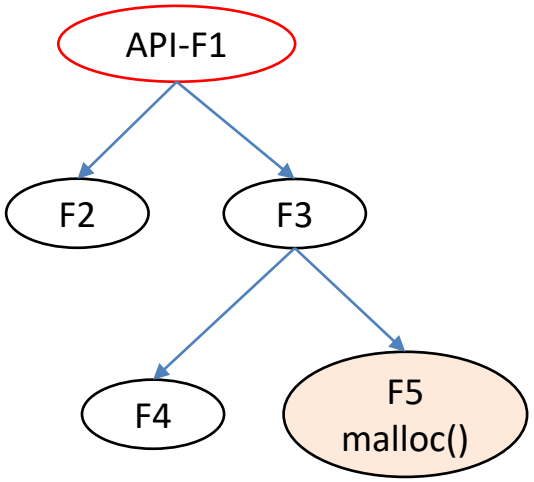
Fallible mode

```
try_malloc(...) -> Result<...>
```

https://rust-lang.github.io/rfcs/2116-alloc-me-maybe.html

Overall Idea: A Convenient Way to Handle OOM

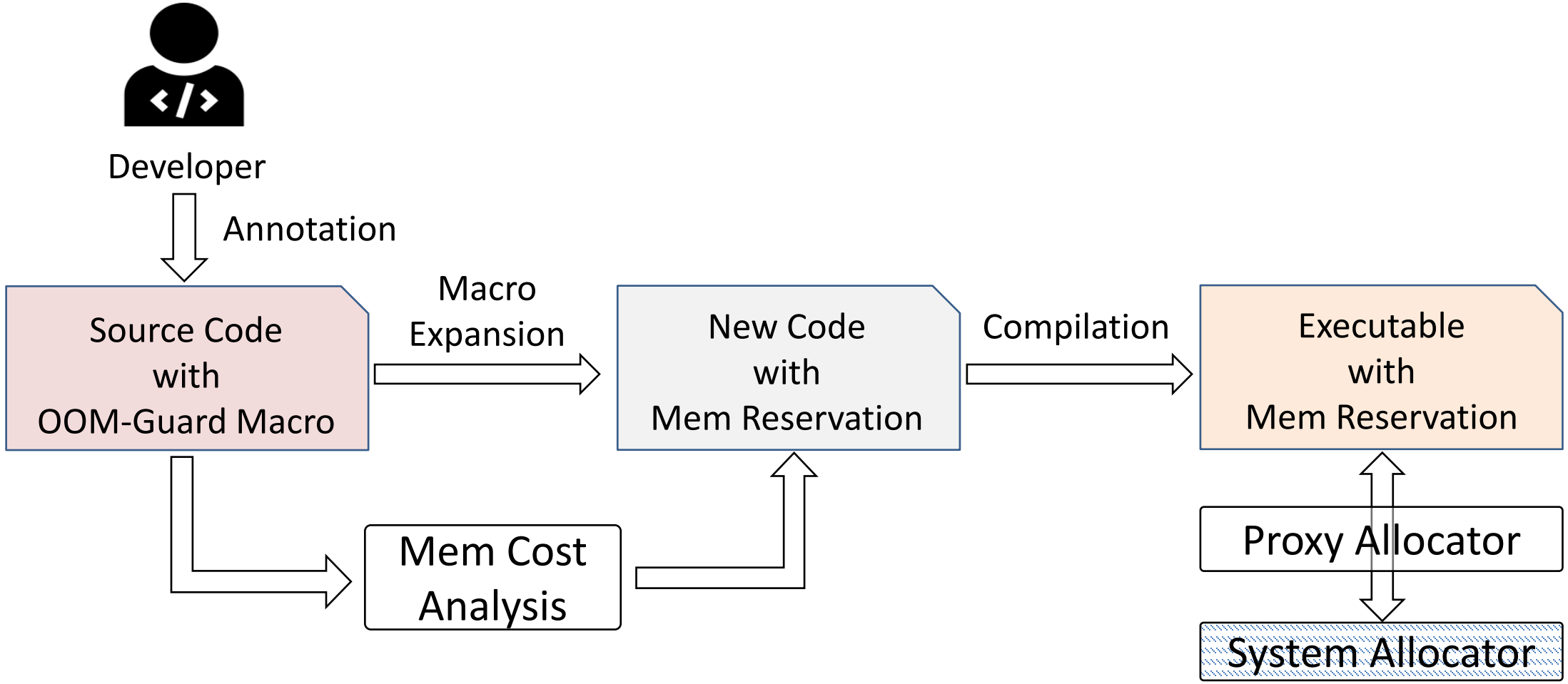
- Reserve a large enough heap space by the top-level API
- Subsequent allocations reusing the space would not fail




```
fn F1(...) -> Result<...>{
   // reservation stmts
   let r2 = F2(...);
   let r3 = F3(...);
   ...
}
```

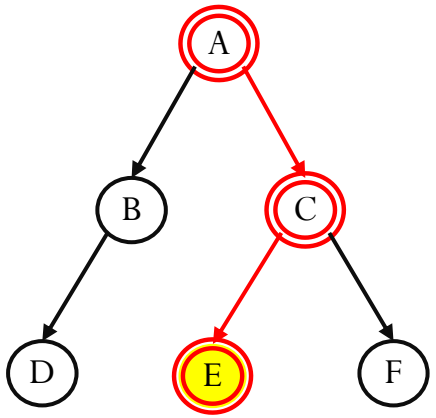
- How to automate the process?
  - Compute the memory size needed for reservation
  - Insert the reservation statements at the function entry
  - Hook subsequent allocations to use the reserved memory

Framework of Our Solution



Memory Cost Analysis

When A is called, reserve max (x, 8) + 8



E is function with allocation sites

```
fn A(x:i8){ B(); C(x);}

fn C(y:i8){
    let z = match y {
        0 => 8,
        _ => y,
    };
    E(z); F();
}

fn E(n:i8){__rust_alloc(n + 8);}
```

Cost Expression of Function A:

Phi(x, 8) + 8

⇧ replace y with x

Cost Expression of Function C:

Phi(y, 8) + 8

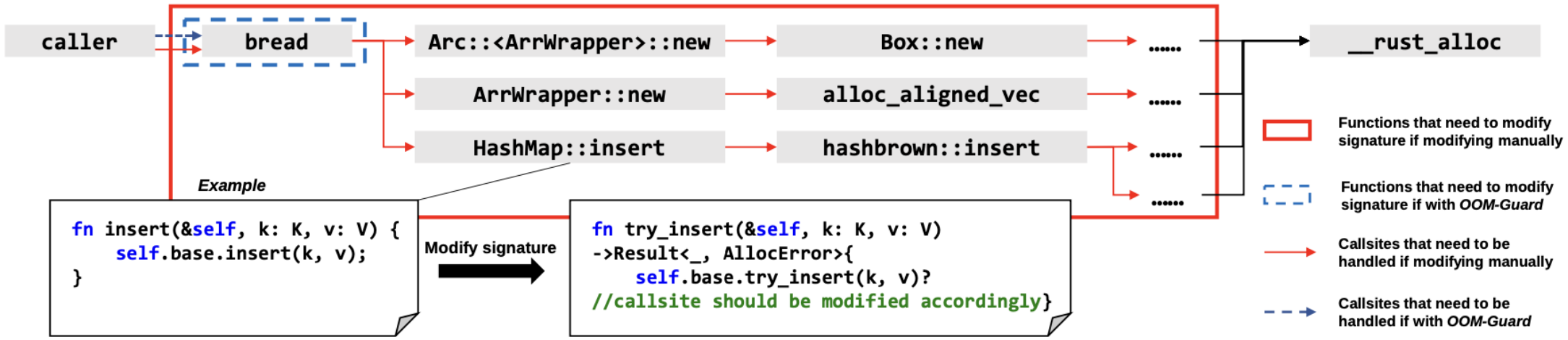
⇧ replace n with Phi(y, 8)

Cost Expression of Function E:

n + 8

- Cases that cannot be analyzed: need more annotations
  - Implicit loop bound: bound annotation
  - Loop variant allocation size: sub-level reservation
  - Dynamic dispatch/function pointers: sub-level reservation

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.

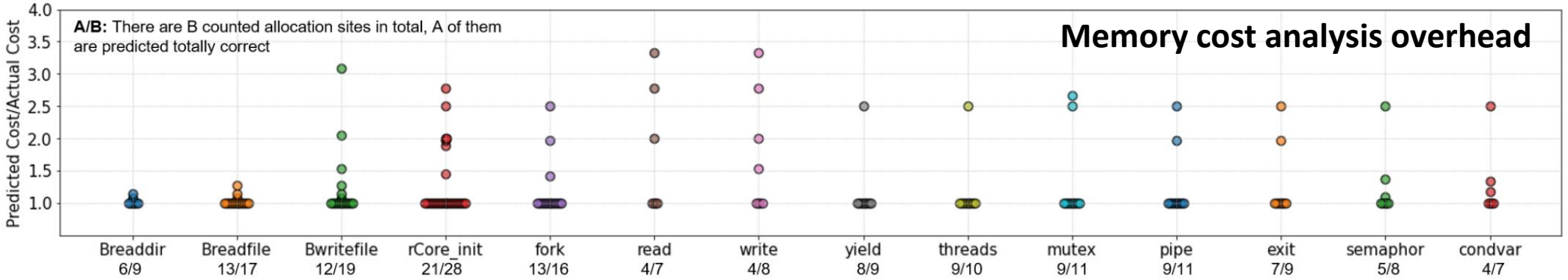
Experiments: Effectiveness

- Target Rust projects:
  - rCore: an operating system
  - Bento-fs: a file system
- Effectiveness: less code needed (1/5), no crash of OOM

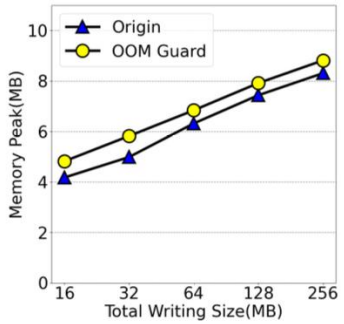
System	# Annotation	# Func Sign	Callsite	Extra
Bento.Try (baseline)	-	243	567	-
Bento.OOM-Guard	13 + 17	-	58	9
rCore.Try (baseline)	-	236	321	-
rCore.OOM-Guard	18 + 23	-	69	9

System	Setting		Crash	# Successful Cases	
	# round	# cases		before OOM	after OOM
Bento.Origin (baseline)	10	100	10	542	0
Bento.OOM-Guard	10	100	0	519	96
rCore.Origin (baseline)	10	100	10	506	0
rCore.OOM-Guard	10	100	0	452	112

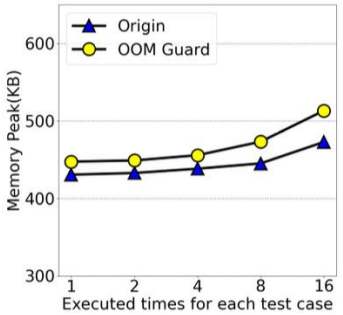
Experimental Results: Efficiency



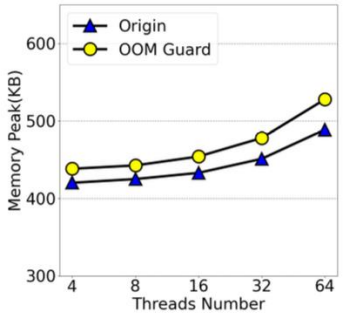
Memory cost analysis overhead



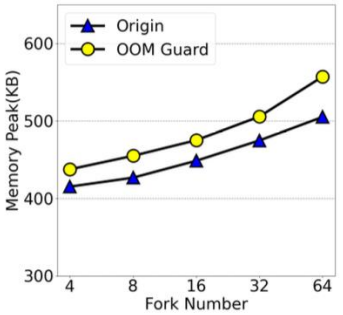
(c) Peak memory usage experiment in Bento-fs.



(d) Peak memory usage experiment in rCore.

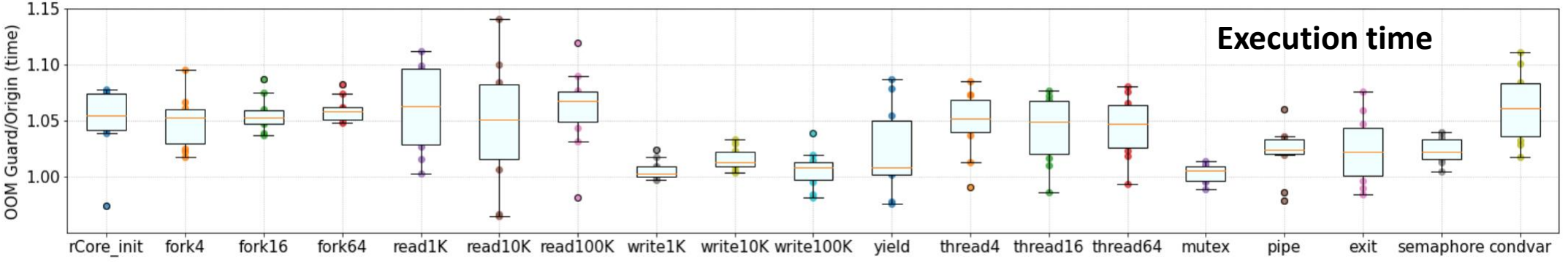


(e) Peak memory usage of thread test in rCore.



(f) Peak memory usage of fork test in rCore.

Peak memory usage comparison



Execution time

5. Summary

Summary and Takeaways

- Limitation of Rust ownership:
  - Ineffective for unsafe code
  - Ignores memory leakage
- Our approaches to detect heap bugs:
  - Dangling pointer: path-sensitive
  - Memory leakage: model checking
- Rust lacks convenient heap exhaustion handling: infallible/fallible
- Our reservation-based approach (hybrid mode) with better ergonomics

GOSIM®
GLOBAL OPEN-SOURCE INNOVATION MEETUP

THANK YOU

xuh@fudan.edu.cn