UnsafeChecker

Finding soundness bugs in Rust safe abstractions

Yin, Zhang, Feng & Xu - Nanjing University
arXiv:2609.09641 [cs.PL]

Reading group - Davsec Lab, UC Davis (Nov 16, 2026)

Presented by Muhammad Hassnain

Part 1

Background

Rust, unsafe, and soundness

Rust gives memory safety without a garbage collector

- **The promise:** no use-after-free, no data races, no buffer overflows - checked at compile time, no runtime GC.

Background: Rust Reference & The Rust Programming Language (rust-lang.org)

Rust gives memory safety without a garbage collector

- **The promise:** no use-after-free, no data races, no buffer overflows - checked at compile time, no runtime GC.
- **The mechanism:** a strict ownership + borrowing system the compiler enforces statically.

Background: Rust Reference & The Rust Programming Language (rust-lang.org)

Rust gives memory safety without a garbage collector

- **The promise:** no use-after-free, no data races, no buffer overflows - checked at compile time, no runtime GC.
- **The mechanism:** a strict ownership + borrowing system the compiler enforces statically.
- **Why it matters:** adopted in Android and the Linux kernel - fast and safe at once.

Background: Rust Reference & The Rust Programming Language (rust-lang.org)

Rust gives memory safety without a garbage collector

- **The promise:** no use-after-free, no data races, no buffer overflows - checked at compile time, no runtime GC.
- **The mechanism:** a strict ownership + borrowing system the compiler enforces statically.
- **Why it matters:** adopted in Android and the Linux kernel - fast and safe at once.
- **The catch (this talk):** those guarantees hold only for code in the safe subset of the language.

Background: Rust Reference & The Rust Programming Language (rust-lang.org)

Safe Rust rests on three invariants the compiler enforces

1 · Unique ownership

Every value has exactly one owner.

Prevents double-free.

Safe Rust rests on three invariants the compiler enforces

1 · Unique ownership

Every value has exactly one owner.

Prevents double-free.

2 · Borrowing & lifetimes

A reference may never outlive its data.

Prevents dangling pointers.

Safe Rust rests on three invariants the compiler enforces

1 · Unique ownership

Every value has exactly one owner.

Prevents double-free.

2 · Borrowing & lifetimes

A reference may never outlive its data.

Prevents dangling pointers.

3 · Memory layout

Pointers stay aligned and in bounds.

Prevents OOB & misalignment.

unsafe unlocks raw pointers - and switches off the checks

- **Why it exists:** some low-level work (hardware access, fast data structures) is too restrictive for the borrow checker.

unsafe unlocks raw pointers - and switches off the checks

- **Why it exists:** some low-level work (hardware access, fast data structures) is too restrictive for the borrow checker.
- **What it does:** the unsafe keyword permits raw pointers (*const T, *mut T) - plain addresses.

unsafe unlocks raw pointers - and switches off the checks

- **Why it exists:** some low-level work (hardware access, fast data structures) is too restrictive for the borrow checker.
- **What it does:** the unsafe keyword permits raw pointers (*const T, *mut T) - plain addresses.

Safe reference &T

Compiler tracks owner, lifetime and layout.
The three invariants are guaranteed.

unsafe unlocks raw pointers - and switches off the checks

- **Why it exists:** some low-level work (hardware access, fast data structures) is too restrictive for the borrow checker.
- **What it does:** the unsafe keyword permits raw pointers (*const T, *mut T) - plain addresses.

Safe reference &T

Compiler tracks owner, lifetime and layout. The three invariants are guaranteed.

Raw pointer *const T

Just an address. Ownership, lifetime, layout all erased - the developer must uphold them by hand.

Libraries wrap unsafe behind safe APIs: “safe abstractions”

- **The pattern:** encapsulate unsafe internals behind a safe public interface, e.g, Vec, Box and Mutex are built this way.

Libraries wrap unsafe behind safe APIs: “safe abstractions”

- **The pattern:** encapsulate unsafe internals behind a safe public interface. Vec, Box and Mutex are built this way.
- **The contract:** the unsafe code must manually uphold the invariants Safe Rust would have enforced.

Libraries wrap unsafe behind safe APIs: “safe abstractions”

- **The pattern:** encapsulate unsafe internals behind a safe public interface. Vec, Box and Mutex are built this way.
- **The contract:** the unsafe code must manually uphold the invariants Safe Rust would have enforced.
- **The promise:** “use me from safe code and nothing bad happens.”

Safe client code → [safe API] → unsafe internals (raw pointers)

Soundness: a safe API can never let safe code cause UB

> *“Unsafe code that satisfies this property for any safe client is called sound; if unsafe code can be misused by safe code to exhibit undefined behavior, it is unsound.”*

- The Rust Reference

Definition: The Rust Reference, “Behavior considered undefined”

Soundness: a safe API can never let safe code cause UB

> *“Unsafe code that satisfies this property for any safe client is called sound; if unsafe code can be misused by safe code to exhibit undefined behavior, it is unsound.”*

- The Rust Reference

Sound

No safe use of the API can ever trigger undefined behavior.

Definition: The Rust Reference, “Behavior considered undefined”

Soundness: a safe API can never let safe code cause UB

> *“Unsafe code that satisfies this property for any safe client is called sound; if unsafe code can be misused by safe code to exhibit undefined behavior, it is unsound.”*

- The Rust Reference

Sound

No safe use of the API can ever trigger undefined behavior.

Unsound

There exists some safe use of the API that leads to undefined behavior (UB).

Definition: The Rust Reference, “Behavior considered undefined”

Unsoundness shows up in two ways

1 · Direct UB inside unsafe

The unsafe code itself executes an illegal operation under safe inputs.

e.g. invalid dereference, out-of-bounds, reading uninitialized memory.

Unsoundness shows up in two ways

1 · Direct UB inside unsafe

The unsafe code itself executes an illegal operation under safe inputs.

e.g. invalid dereference, out-of-bounds, reading uninitialized memory.

2 · Invalid state leaked out

The API returns something invalid; UB happens later, in the caller.

e.g. a dangling pointer or duplicated ownership handed back to safe code.

Catching the second kind needs reasoning about what escapes a function - not just crashes inside it.

Part 2

The problem

Who checks the unsafe code?

One mistake in internal unsafe code breaks the whole contract

- The compiler does not validate invariants inside unsafe blocks.

One mistake in internal unsafe code breaks the whole contract

- The compiler does not validate invariants inside unsafe blocks.

- So a single implementation error can make the whole abstraction unsound …

One mistake in internal unsafe code breaks the whole contract

- The compiler does not validate invariants inside unsafe blocks.
- So a single implementation error can make the whole abstraction unsound …
- … and then **safe client code can trigger undefined behavior**

So we need a tool that checks safe abstractions for soundness bugs automatically.

Existing tools miss these bugs for two different reasons

C / C++ static analyzers

Flat memory model. Catch buffer overflows, but ignore Rust's safety contracts.

Duplicating a pointer is fine in C - in Rust it can break ownership uniqueness.

Existing tools miss these bugs for two different reasons

C / C++ static analyzers

Flat memory model. Catch buffer overflows, but ignore Rust's safety contracts.

Duplicating a pointer is fine in C - in Rust it can break ownership uniqueness.

Existing Rust tools

Syntactic pattern-matching or generic memory checks.

Lack the semantic modeling to track ownership, lifecycle & layout that unsafe erases.

Neither tracks the semantic context raw pointers erase - so neither can tell whether the contract still holds.

Research question

Can we automatically detect when unsafe code breaks the safety contract its safe API promise

Research question

Can we automatically detect when unsafe code breaks the safety contract its safe API promise

Contributions

- **Framework:** a MIR-level analysis holding ownership, object-validity & layout facts in one shared, flow-sensitive state.

Research question

Can we automatically detect when unsafe code breaks the safety contract its safe API promise

Contributions

- **Framework:** a MIR-level analysis holding ownership, object-validity & layout facts in one shared, flow-sensitive state.
- **Tool:** UnsafeChecker, a rustc compiler plugin that fits existing build pipelines.

Research question

Can we automatically detect when unsafe code breaks the safety contract its safe API promise

Contributions

- **Framework:** a MIR-level analysis holding ownership, object-validity & layout facts in one shared, flow-sensitive state.
- **Tool:** UnsafeChecker, a rustc compiler plugin that fits existing build pipelines.
- **Study:** 46-CVE benchmark + a 100k-crate scan; 114 new bugs found in the wild.

Part 3

Some Examples

Ownership, Lifetime and Layout Bugs

Bug A

```
fn copy_struct<T>(&self, addr: usize) -> Result<T, Error> {
    let mut data = vec![0; size_of::<T>()];  // Vec<u8>, 1-byte align
    self.read(addr, &mut data)?;
    Ok(unsafe { ptr::read(data.as_ptr() as *const T) })
}
```

Listing 1 - remoteprocess crate: a safe copy_struct<T> over unsafe ptr::read

Source: RustSec-derived benchmark, remoteprocess (paper §2.2)

Bug A: one ptr::read duplicates ownership → double-free

```
fn copy_struct<T>(&self, addr: usize) -> Result<T, Error> {
    let mut data = vec![0; size_of::<T>()];  // Vec<u8>, 1-byte align
    self.read(addr, &mut data)?;
    Ok(unsafe { ptr::read(data.as_ptr() as *const T) })
}
```

Listing 1 - remoteprocess crate: a safe copy_struct<T> over unsafe ptr::read

- **Layout:** Vec<u8> gives 1-byte alignment; if T needs 8-byte alignment, the read is misaligned → UB.
- **Ownership:** with no Copy bound, ptr::read bitwise-copies T, creating a second owner → double-free.

Source: RustSec-derived benchmark, remoteprocess (paper §2.2)

Bug B

```
fn next(&mut self) -> Option<usize> {
    while self.pos < self.max {
        let opt = unsafe { ptr::read(self.head.add(self.pos)) };
        if opt.is_some() { self.pos += 1; return Some(self.pos - 1); }
        self.pos += 1;
    }  // opt (a String) is dropped here - frees the map's buffer
    None
}
```

Listing 2 - emap crate: next() copies bytes out of the internal buffer

Bug B: ptr::read in an iterator leaves a dangling pointer

```
fn next(&mut self) -> Option<usize> {
    while self.pos < self.max {
        let opt = unsafe { ptr::read(self.head.add(self.pos)) };
        if opt.is_some() { self.pos += 1; return Some(self.pos - 1); }
        self.pos += 1;
    }  // opt (a String) is dropped here - frees the map's buffer
    None
}
```

Listing 2 - emap crate: next() copies bytes out of the internal buffer

- ptr::read copies the String into opt; at scope end opt is dropped and frees the heap buffer.
- But the map still marks the slot occupied → later m.get(0) is a use-after-free.

Key insight: track the three semantics unsafe erases

Plain value-based dataflow treats a pointer as just an address - it can't explain why ptr::read duplicates ownership or returns freed memory. So track all three, flow-sensitively:

Ownership

Who owns what, and which pointers alias the same resource.

Key insight: track the three semantics unsafe erases

Plain value-based dataflow treats a pointer as just an address - it can't explain why ptr::read duplicates ownership or returns freed memory. So track all three, flow-sensitively:

Ownership

Who owns what, and which pointers alias the same resource.

Lifecycle

Runtime object state: live, moved, dropped, uninitialized…

Key insight: track the three semantics unsafe erases

Plain value-based dataflow treats a pointer as just an address - it can't explain why ptr::read duplicates ownership or returns freed memory. So track all three, flow-sensitively:

Ownership

Who owns what, and which pointers alias the same resource.

Lifecycle

Runtime object state: live, moved, dropped, uninitialized…

Layout

Byte-level bounds, offset, alignment, nullability, mutability.

These dimensions interact - e.g. dropping an owner invalidates every alias to its resource.

Part 4

Methodology

How UnsafeChecker analyzes Rust MIR

The pipeline: crate → MIR → shared state → warnings



σ_A Ownership

points-to, ownership hierarchy, aliasing

σ_O Object-validity

live / moved / dropped / uninit…

σ_L Layout

base, size, offset, align, null, mut

Propogate transfer functions while checking instruction-level UB (during) and boundary-level violations (at exit).

Redrawn from Fig. 2 (paper §3.1)

Step 1: abstract MIR places are divided into three kinds of locations

SO · stack owners

Non-Copy roots of ownership (Vec, Box).

May manage heap resources.

Step 1: abstract MIR places are divided into three kinds of locations

SO · stack owners

Non-Copy roots of ownership (Vec, Box).

May manage heap resources.

P · pointers

References & raw pointers (&T, *mut T).

Access objects without owning them.

Step 1: abstract MIR places are divided into three kinds of locations

SO · stack owners

Non-Copy roots of ownership (Vec, Box).

May manage heap resources.

P · pointers

References & raw pointers (&T, *mut T).

Access objects without owning them.

V_copy · Copy data

Other Copy locals (usize, bool).

Feed arithmetic in the layout domain.

- **Field-sensitive:** x.f is distinct from x - partial ownership and disjoint borrows are tracked.
- **Heap:** allocation-site abstraction keeps the heap finite.

Ownership: track who owns, and who aliases

- **Points-to (Pt):** may-points-to map - which objects each pointer might reference.

Ownership: track who owns, and who aliases

- **Points-to (Pt):** may-points-to map - which objects each pointer might reference.
- **Ownership hierarchy (Own):** which heap resources an object is responsible for (RAII / “Vec owns its buffer”).

Ownership: track who owns, and who aliases

- **Points-to (Pt):** may-points-to map - which objects each pointer might reference.
- **Ownership hierarchy (Own):** which heap resources an object is responsible for (RAII / “Vec owns its buffer”).
- **Ownership aliasing (Alias):** which owners claim the same underlying resource.

Safe Rust keeps |Alias(o)| = 1. Two live owners in one alias set is the structural precursor to a double-free.

DOMAIN 2 OF 3

Object-validity: each object is sorted into one of these

Live

Initialized & valid.

Uninit

Allocated, not written. Read = UB.

Moved

Ownership transferred. Use = error.

Dropped

Destructor ran. Use = use-after-free.

ManuallyDropped

Valid, auto-drop suppressed.

Forgotten

mem::forget - moved, resource leaked.

At control-flow joins, the state is a set (e.g. {Live, Uninit}) - so their tool can report “maybe uninitialized on some path.”

Paper §3.4 - runtime object state, distinct from compile-time lifetimes

DOMAIN 3 OF 3

Layout: byte-level facts about every pointer

```
σ_L(p) = ( base, size, offset[l,u], align, null, mut )
```

- **base** = target block; **size** = guaranteed capacity; **offset** = position interval [l,u].

DOMAIN 3 OF 3

Layout: byte-level facts about every pointer

```
σ_L(p) = ( base, size, offset[l,u], align, null, mut )
```

- **base** = target block; **size** = guaranteed capacity; **offset** = position interval [l,u].
- **align, null, mut** = alignment guarantee, nullability, mutability.

DOMAIN 3 OF 3

Layout: byte-level facts about every pointer

```
σ_L(p) = ( base, size, offset[l,u], align, null, mut )
```

- **base** = target block; **size** = guaranteed capacity; **offset** = position interval [l,u].
- **align, null, mut** = alignment guarantee, nullability, mutability.
- **Conservative by design:** lower-bound guarantees, over-approximated offsets; widening at loop headers keeps it terminating.

Catches out-of-bounds, misalignment, null deref, and bad pointer arithmetic - the largest bug class.

Paper §3.5 - layout domain

Propagate the combined state to a fixpoint

- **Composite domain:** Σ = (σ_A, σ_O, σ_L); order & join are component-wise - a conservative may-approximation.

Propagate the combined state to a fixpoint

- **Composite domain:** Σ = (σ_A, σ_O, σ_L); order & join are component-wise - a conservative may-approximation.
- **Transfer functions** model MIR operations - each updates only the facts it touches:

Propagate the combined state to a fixpoint

- **Composite domain:** Σ = (σ_A, σ_O, σ_L); order & join are component-wise - a conservative may-approximation.
- **Transfer functions** model MIR operations - each updates only the facts it touches:

```
T-Alloc     fresh heap object; Live; ownership + layout initialized
T-Move      transfer ownership topology; mark source Moved
T-ReadBits  ptr::read → new owner + merge alias sets
T-Drop      recursive drop: mark object AND all aliases Dropped
T-Offset    propagate bounds & alignment through pointer arithmetic
```

Propagate the combined state to a fixpoint

- **Composite domain:** Σ = (σ_A, σ_O, σ_L); order & join are component-wise - a conservative may-approximation.
- **Transfer functions** model MIR operations - each updates only the facts it touches:

```
T-Alloc     fresh heap object; Live; ownership + layout initialized
T-Move      transfer ownership topology; mark source Moved
T-ReadBits  ptr::read → new owner + merge alias sets  (duplication!)
T-Drop      recursive drop: mark object AND all aliases Dropped
T-Offset    propagate bounds & alignment through pointer arithmetic
```

Propagate the combined state to a fixpoint

- **Composite domain:** Σ = (σ_A, σ_O, σ_L); order & join are component-wise - a conservative may-approximation.
- **Transfer functions** model MIR operations - each updates only the facts it touches:

```
T-Alloc     fresh heap object; Live; ownership + layout initialized
T-Move      transfer ownership topology; mark source Moved
T-ReadBits  ptr::read → new owner + merge alias sets  (duplication!)
T-Drop      recursive drop: mark object AND all aliases Dropped
T-Offset    propagate bounds & alignment through pointer arithmetic
```

Functional analysis unto to 3 layers deep - does not see FFI

Paper §3.6

Warnings fire at two granularities

Rule	Detects	When
E-AliasRet	Double ownership returned	Boundary
E-Dangling	Dangling pointer escapes	Boundary
E-Layout	Invalid container (ptr/len/cap)	Boundary
E-Leak	Resource leak at exit	Boundary
E-Access	UAF / uninit / use-after-move	Instruction
E-DoubleFree	Dropping an invalid resource	Instruction
E-OOB / MA	Out-of-bounds / misalignment	Instruction
E-NullDeref	Dereference of maybe-null ptr	Instruction

Boundary-level *checks run at function exit on values that escape to safe callers.* ***Instruction-level*** *checks run during propagation on unsafe operations.*

Part 5

Evaluation

A CVE benchmark, a 100k-crate scan, and four questions

Setup

Dataset A - ground truth

46 RustSec CVEs
53 ground-truth bugs

Setup

Dataset A - ground truth

46 RustSec CVEs
53 ground-truth bugs

Dataset B - the wild

100k+ crates scanned
~10k manually inspected
83 crates with real bugs

Setup

Dataset A - ground truth

46 RustSec CVEs
53 ground-truth bugs

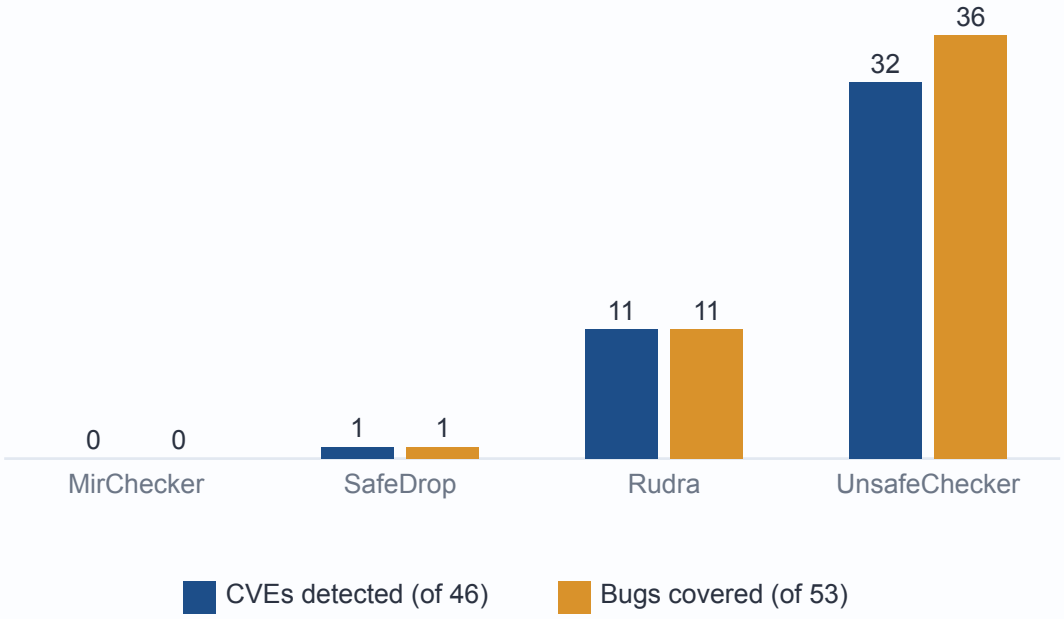
Dataset B - the wild

100k+ crates scanned
~10k manually inspected
83 crates with real bugs

- **Baselines:** MirChecker (MIR symbolic), Rudra (pattern-matcher), SafeDrop (dealloc bugs).
- RQ1 effectiveness · RQ2 precision/recall · RQ3 efficiency · RQ4 discovery in the wild.

Dataset B rule: a bug counts only if a Miri-checked PoC triggers UB via safe public APIs

RQ1: UnsafeChecker detects 32 of 46 CVEs - ~3× the best baseline



- **69.6% CVE recall** vs 23.9% for Rudra, the best baseline.
- **67.9% bug recall** (36 / 53)
- **20+ vulnerabilities** missed by all baselines.
- UnsafeChecker Reasons over object state & derived pointers unlike others which look at fixed API signatures.

Table 1 (paper §5.1). MirChecker 0 · SafeDrop 1 · Rudra 11 · UnsafeChecker 32 CVEs

Three cases every baseline missed - and why

tracing

RUSTSEC-2023-0078

mem::forget(self) then reads field pointers. Caught as use-after-move via the Forgotten state.

Paper §5.1 - representative cases

Three cases every baseline missed - and why

tracing

RUSTSEC-2023-0078

mem::forget(self) then reads field pointers. Caught as use-after-move via the Forgotten state.

temporary

RUSTSEC-2018-0022

mem::uninitialized read for entropy. Caught by the Uninit lifecycle state.

Paper §5.1 - representative cases

Three cases every baseline missed - and why

tracing

RUSTSEC-2023-0078

mem::forget(self) then reads field pointers. Caught as use-after-move via the Forgotten state.

temporary

RUSTSEC-2018-0022

mem::uninitialized read for entropy. Caught by the Uninit lifecycle state.

wrflib

RUSTSEC-2025-0072

ptr.add(offset) with no bounds check. Caught by layout bounds propagation.

Paper §5.1 - representative cases

Three cases every baseline missed - and why

tracing

RUSTSEC-2023-0078

mem::forget(self) then reads field pointers. Caught as use-after-move via the Forgotten state.

temporary

RUSTSEC-2018-0022

mem::uninitialized read for entropy. Caught by the Uninit lifecycle state.

wrflib

RUSTSEC-2025-0072

ptr.add(offset) with no bounds check. Caught by layout bounds propagation.

Each needs semantic reasoning over object state or derived pointers.

Paper §5.1 - representative cases

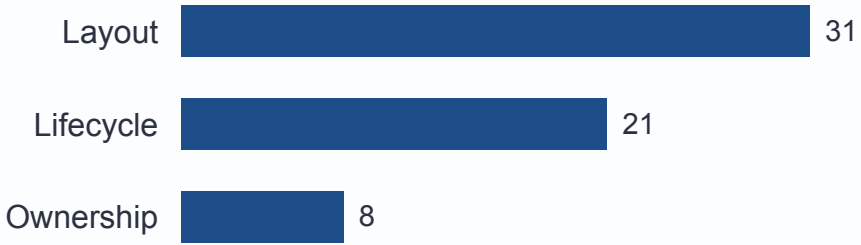
RQ2: 67.9% bug recall at 51.6% alert-level precision

124
total alerts

64
true positives

60
false positives

Where false positives come from



- **Layout dominates** (52%): conservative bounds + interval widening.
- **Missed:** 14 CVEs - mostly unmodeled library APIs.

Paper §5.2 - FP sources: layout 31, lifecycle 21, ownership 8

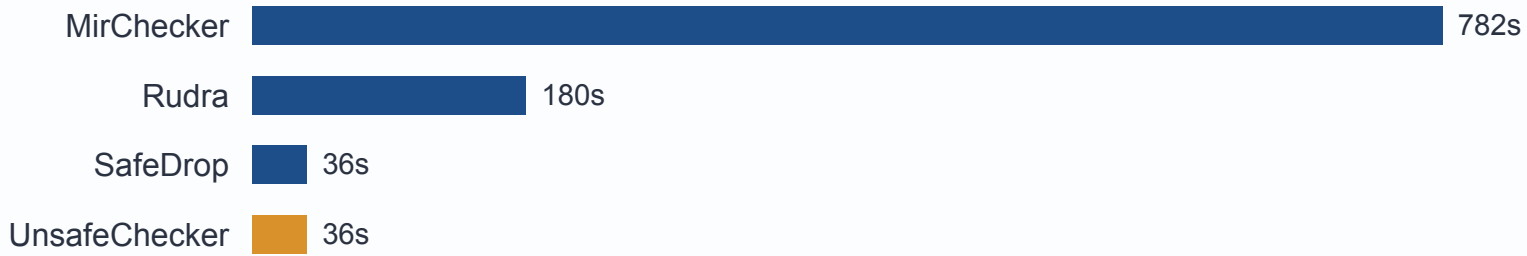
RQ3: fast enough for daily use

11.4s
median end-to-end time per crate

100k+
crates analyzed in two days

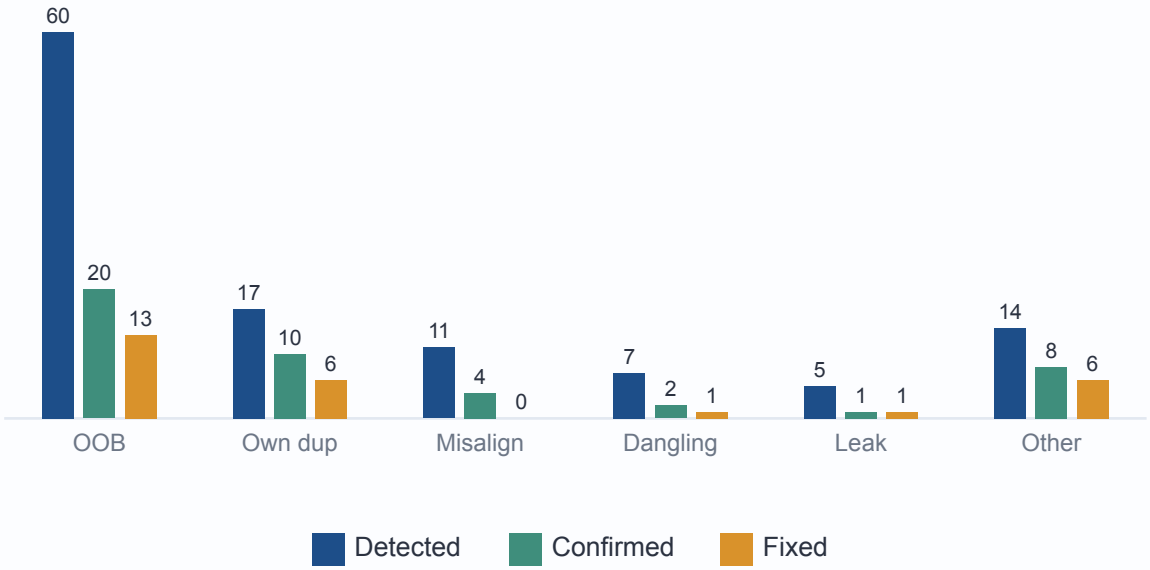
0.78s
average per CVE case

Time to analyze all 46 CVE cases (seconds, lower is better)



Paper §5.3 - 5× faster than Rudra, >20× faster than MirChecker; median memory 475 MB, peak 4.3 GB

RQ4: 114 new bugs in the wild - 45 confirmed, 27 fixed



- **83 crates** affected, 30M+ downloads.
- **OOB dominates:** 53% of findings.
- **11 findings** got public vuln IDs.
- **Not just ptr::read:** 129 sites across 21 APIs.

Table 3 (paper §5.4) - “Other” folds in integer-overflow, invalid/null pointer, div-by-zero, UTF-8, uninit

Case study

```
fn with_shared_mut_ref<T, F>(t: &mut T, f: F) {
    unsafe {
        let obj = std::ptr::read(t);
        let obj_rc = Rc::new(obj);
        f(obj_rc.clone());
        std::ptr::write(t, ...);
    }
}
```

Listing 5 - multiversx_chain_vm

Case study: a panic-triggered double-free (132k downloads)

```
fn with_shared_mut_ref<T, F>(t: &mut T, f: F) {
    unsafe {
        let obj = std::ptr::read(t);      // [1] duplicate ownership
        let obj_rc = Rc::new(obj);
        f(obj_rc.clone());  // [2] if f panics: Rc dropped → Free #1
        std::ptr::write(t, ...); // [3] restore - skipped on panic
    }
}
```

Listing 5 - multiversx_chain_vm

- ptr::read makes two owners of obj; if the closure f panics, unwinding drops the Rc (free #1) and skips the restore , the caller later frees the stale copy (free #2).

Why one shared state beats pattern matching

- **Real soundness bugs span multiple steps and invariants:** e.g, duplicating a pointer and then freeing one of them.

Why one shared state beats pattern matching

- **Real soundness bugs span multiple steps and invariants:** e.g, duplicating a pointer and then freeing one of them.
- **Pattern matchers** flag a risky API in isolation; they can't connect the steps that cause UB.

Why one shared state beats pattern matching

- **Real soundness bugs span multiple steps and invariants:** duplicate a pointer here, free the resource there.
- **Pattern matchers** flag a risky API in isolation; they can't connect the steps that cause UB.
- **UnsafeChecker** keeps ownership, validity & layout in one state, so each rule consumes exactly the facts it needs.

Why one shared state beats pattern matching

- **Real soundness bugs span multiple steps and invariants:** duplicate a pointer here, free the resource there.
- **Pattern matchers** flag a risky API in isolation; they can't connect the steps that cause UB.
- **UnsafeChecker** keeps ownership, validity & layout in one state, so each rule consumes exactly the facts it needs.

Scope and limitations

Scope

Sequential memory safety only.

No concurrency / data races; no Sync/Send reasoning.

Paper §3.1 and §6.3 - threats to validity and future work

Scope and limitations

Scope

Sequential memory safety only.

No concurrency / data races; no Sync/Send reasoning.

Precision

Conservative → false positives.

Interval layout weak on non-linear arithmetic; FFI over-approximated.

Paper §3.1 and §6.3 - threats to validity and future work

Scope and limitations

Scope

Sequential memory safety only.

No concurrency / data races; no Sync/Send reasoning.

Precision

Conservative → false positives.

Interval layout weak on non-linear arithmetic; FFI over-approximated.

Modeling

Key APIs modeled (Vec, Box, raw ptrs).

Not the whole std lib; custom allocators & deep RefCell unmodeled.

It is a bug-finder, not a proof: the claim covers the memory-safety cases, MIR operations, and APIs it models.

Paper §3.1 and §6.3 - threats to validity and future work

Conclusions

- **1. Soundness bugs come from three semantics unsafe erases.** Ownership, lifecycle and layout - in one shared, flow-sensitive MIR state.
- **2. Two-granularity checking catches what others miss.** Instruction-level UB and boundary-level violations that escape through safe APIs.
- **3. It finds real bugs.** 32 / 46 CVEs (~3× the best baseline), and 114 new bugs in the wild; 45 confirmed, 27 fixed.

Paper: arXiv:2609.09641 · Yin, Zhang, Feng & Xu, Nanjing University · fengyang@nju.edu.cn

References

Yin, Zhang, Feng & Xu (2026). UnsafeChecker: Finding Soundness Bugs in Rust Safe Abstractions. arXiv:2609.09641 [cs.PL].

Bae, Kim, Askar, Lim & Kim (2021). Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. SOSP '21.

Li, Wang, Sun & Lui (2021). MirChecker: Detecting Bugs in Rust Programs via Static Analysis. CCS '21.

Cui, Chen, Xu & Zhou (2023). SafeDrop: Detecting Memory Deallocation Bugs via Static Data-flow Analysis. TOSEM 32(4).

Astrauskas, Matheja, Poli, Müller & Summers (2020). How Do Programmers Use Unsafe Rust? Proc. ACM PL 4, OOPSLA.

Jung, Dang, Kang & Dreyer (2019). Stacked Borrows: An Aliasing Model for Rust. Proc. ACM PL 4, POPL.

Jung, Jourdan, Krebbers & Dreyer (2018). RustBelt: Securing the Foundations of the Rust Programming Language. Proc. ACM PL 2, POPL.

The Rust Project (2025). The Rust Reference - Behavior Considered Undefined.

The Rust Project (2025). Miri: An Undefined Behavior Interpreter for Rust.

Xu, Chen, Sun, Zhou & Lyu (2021). Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. TOSEM 31(1).

Appendix

Backup slides

Transfer rules, detection rules, and extra cases for questions

How MIR operations update the shared state

```
T-Alloc     x=Alloc(T):  fresh heap h; Live; Own(x)={h};
            Pt(p_x)={h}; layout=(h,size,[0,0],align,NonNull,Mut)
T-Move      y=x (!Copy): Transfer topology x→y; source Moved
T-ReadBits  y=ptr::read(p): new owner y + AliasMerge with Pt(p)
T-Drop      drop(x): RecursiveDrop - x and all aliases Dropped
T-Forget    mem::forget(x): x (and fields) → Forgotten
T-ManDrop   y=ManuallyDrop(x): x Moved; y valid, auto-drop off
T-AddrOf    p=&x: Pt(p)={x}; layout from x
T-Cast      p=q as _: copy Pt & layout; update mutability only
T-Offset    p=q.offset(i): shift bounds; align=gcd(align, size_of T)
```

Paper Fig. 3 - intra-procedural transfer functions

What each rule actually checks

- **Boundary (at exit, on escaping values V_out):**
- **E-AliasRet** two return values share an Own set → double-free risk.
- **E-MutAlias** two returned refs alias, ≥1 mutable → breaks uniqueness.
- **E-Dangling** an escaping pointer targets Dropped/Moved/Forgotten/Uninit memory.
- **E-Layout / E-Leak** bad container invariants; live-but-unreachable heap.
- **Instruction (during propagation):**
- **E-Access, E-DoubleFree, E-RefCreation, E-OOB/MA, E-NullDeref,**
- **E-PtrArith, E-Proj, E-SliceInvalid, E-SizeMismatch, E-MutateImmut.**

Paper Figs. 4-5 - boundary-level & instruction-level checks

Three more real-world bugs, one per domain

doubly

Ownership

pop_back uses ptr::read then drops the Box → double free of the value.

auto_vec

Layout

Iterator does (&children[0]).sub(1) - pointer moves before the allocation → OOB UB.

bytes-kman

Lifecycle / leak

Box::leak then ptr::read copies bits - leaked heap is never freed.

The 14 missed CVEs are modeling gaps, not rule failures

- **7** need library/API semantics not yet modeled (e.g. Layout::from_size_align, Cell, debug_assert).
- **3** need more precise numeric / loop / iterator reasoning than interval abstraction gives.
- **2** need per-element array / initialization-state tracking finer than the current memory model.
- **2** depend on external or cross-method logic state (cache behavior, raw-pointer state across calls).

Each miss points to a concrete engineering extension - the warning rules themselves stay valid.

Paper §5.2.2 - false-negative analysis