Beyond k-Limiting: Pointer-Flow-Guided Context Sensitivity for Scalable and Precise Rust Pointer Analysis

Wenyao Chen
UNSW Sydney, Australia

Wei Li[1]
UNSW Sydney, Australia

Jingling Xue[1]
UNSW Sydney, Australia

Abstract

Pointer analysis for Rust faces unique challenges arising from its ownership-based memory model and layered abstractions, which complicate how heap-allocated objects flow across functions. Existing k-limited callsite abstractions – designed for earlier languages – are both imprecise and inefficient on large Rust programs. We present Rceus, a Rust-oriented pointer-analysis technique that mitigates points-to set explosion and resource exhaustion caused by cross-function pointer conflation under deep heap encapsulation, a scalability bottleneck that conventional k-limiting cannot address.

Rceus performs a fast, coarse-grained pointer-flow pre-analysis to identify precision-critical functions and the essential callsites within their calling contexts. This selective context construction distinguishes parameter-derived flows while avoiding unnecessary expansion. As a result, Rceus cleanly partitions intertwined pointer flows, eliminating context explosion and improving both scalability and precision.

On 16 real-world Rust applications, Rceus outperforms state-of-the-art techniques – standard k-limiting, selective k-limiting for Java, and stack-filtered k-limiting for Rust – in both precision and efficiency. The evaluation includes `Wasmtime`, a WebAssembly runtime with 669K lines of code, where the benefits increase with program size. Rceus also composes with existing techniques, providing a practical and extensible foundation for scalable, precise Rust pointer analysis.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Pointer Analysis, Context Sensitivity, Rust

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.1

Supplementary Material *Software (Artifact)*: `https://zenodo.org/records/18502360`
Software (ECOOP 2026 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.12.1.12`

Funding Australian Research Council Grants No. DP240103194.

Acknowledgements We thank the reviewers for their valuable and constructive feedback.

1 Introduction

Problem Statement. Pointer analysis is a foundational technique for reasoning about heap interactions [24, 3], pointer aliasing [47, 22], and memory safety [17, 32, 38]. Its effectiveness, however, is intrinsically tied to the language's memory model and abstraction mechanisms.

Rust's ownership model and layered abstractions introduce pointer-flow patterns that differ substantially from those in languages such as C and Java. In those settings, k-limited callsite sensitivity – which models calling contexts using the k most recent callsites [31] –

[1] Corresponding authors

and its refinements [25, 22, 10, 40] strike a practical balance between precision and efficiency. Although recent work has adapted k-limited abstractions to Rust [19, 18] and observed moderate precision gains with larger k, these approaches remain fundamentally constrained, exhibiting both notable imprecision and poor scalability on large Rust codebases. Our study shows that Rust's unique pointer-flow characteristics call for a dedicated analysis strategy capable of achieving scalable, precise pointer reasoning beyond what k-limiting can provide.

Challenges. Rust's deterministic, ownership-based memory model manages stack values and enforces disciplined access and deallocation of heap data. Heap allocations are typically mediated by smart pointers and containers (e.g., `Box<T>`, `Rc<T>`, `Vec<T>`, and `String`), which are stack-allocated structs holding raw pointers to heap objects. All heap access is funneled through these stack-resident handles, enforcing strict aliasing control [14]. Traits and generics introduce additional indirection through wrapper structs and trait objects [15].

These abstractions ensure memory safety but impose deep heap encapsulation that complicates static analysis. Under k-limited context sensitivity, small k values collapse long call chains and conflate distinct heap objects, inflating points-to sets, reducing precision, and risking memory exhaustion on large programs. Increasing k can, in principle, avoid such conflation by preserving full calling contexts, but the number of contexts grows exponentially with call depth and recursion, making large k values infeasible in real-world Rust codebases.

Thus, Rust pointer analysis requires a strategy that selectively preserves the deep calling contexts needed to avoid spurious merges while avoiding their prohibitive cost. By retaining only semantically meaningful deep contexts and eliminating unnecessary expansion, such a strategy can mitigate points-to set explosion, improve precision, and maintain scalability.

Prior Work. Existing pointer analyses for object-oriented languages like Java (memory-safe) and imperative languages like C (memory-unsafe) do not require specialized modeling of heap abstractions. In Java, heap objects created via `new()` are managed by the JVM, and in C, objects allocated via `malloc()` are treated as unstructured memory. Because neither language employs ownership, borrowing, or lifetime-based encapsulation, prior analyses did not need dedicated strategies for modeling heap objects.

Pointer analyses for Java commonly rely on k-limited context sensitivity, applied either uniformly (*kcs*) [19, 9, 45, 35] or selectively to chosen functions or objects (Sel-*kcs*) [22, 20, 25, 11, 36, 8]. These approaches typically restrict k to 1–2 to avoid exponential growth in context count. Other techniques [10, 12, 41] relax the k-most-recent-callsites restriction by selecting k callsites along each chain based on program features, refining context abstractions without increasing k. Their selection policies, however, are tailored to Java's object and call structure and do not generalize easily to other languages. In contrast, pointer analyses for C often emphasize flow sensitivity [37, 7, 34], reflecting different language semantics.

Rust combines Java's memory safety with C's low-level control, yet remains relatively underexplored in pointer analysis. Rupta [19] introduced the first k-callsite-sensitive analysis for Rust (*kcs*), later extended by Stack Filtering (SF-*kcs*) [18], which uses a lifetime-based pre-analysis to eliminate spurious stack targets and is therefore never less precise than *kcs*. While SF-*kcs* improves precision and efficiency, it focuses only on stack objects and leaves heap imprecision unresolved – a notable limitation, since heap data without explicit lifetimes is common in real-world Rust codebases. Imprecision and inefficiency caused by conflated heap objects thus remain a key challenge for scalable and precise Rust pointer analysis.

This Work. We present RCEUS, a new context-sensitive pointer-analysis approach for Rust based on a key insight: in Rust, precise reasoning about heap objects requires preserving only the calling contexts that carry distinct pointer-flow origins, rather than uniformly applying k-limited callsite sensitivity. RCEUS leverages interprocedural pointer-flow information to select these contexts, enabling scalable and precise analysis of heap objects and naturally improving precision for stack objects through the same context abstraction.

RCEUS begins with a fast, coarse-grained pointer-flow pre-analysis that identifies precision-critical functions – those whose return values may depend on their parameters, directly or transitively – and determines, for each such function, the callsites that must be preserved in its calling contexts. These functions often include methods of heap-managing container types (e.g., `Box<T>`, `Rc<T>`, `Vec<T>`, `String`), where accurate separation of allocation origins is necessary to avoid conflation. Using the callsites selected during pre-analysis, RCEUS constructs contexts by choosing exactly one callsite per context from each call chain leading to a precision-critical function. These callsites, derived from interprocedural pointer-flow paths, identify where distinct allocation origins first enter the function's computation. By preserving contexts using singleton callsites at these specific sites – rather than uniformly or through heuristics as in k-limiting and its refinements – RCEUS separates flows from different heap objects while avoiding unnecessary context expansion.

Built on the open-source RUPTA [19] framework over Rust's Mid-level Intermediate Representation (MIR), RCEUS outperforms state-of-the-art k-limited techniques – RUPTA (standard k-limiting, *kcs*) [19], SF-*kcs* (stack-filtered k-limiting) [18], and SEL-*kcs* (selective k-limiting for Java) [22] – in precision, efficiency, and scalability across 16 Rust projects with $k \in \{1, 2\}$. These include `Wasmtime`, a WebAssembly runtime of over 669K LOC, where the benefits amplify with program size. Compared with *kcs*, SF-*kcs*, and SEL-*kcs* at $k = 1$ across all applications, RCEUS reduces average points-to set sizes by 86.6%, 68.9%, and 87.9%, and achieves 4.9×, 2.4×, and 2.2× speedups on average, all while using less memory.

In summary, our work makes the following contributions:

- We show that Rust's ownership model both necessitates and enables a new form of interprocedural reasoning in which context sensitivity follows pointer-flow structure to achieve scalable and precise analysis of heap objects.
- We present RCEUS, an open-source framework operating on Rust MIR that constructs contexts by separating pointer flows with different allocation origins using pointer-flow-guided callsite selection, thereby avoiding redundant context expansion.
- We evaluate RCEUS on 16 real-world Rust applications and show that it substantially outperforms state-of-the-art k-limited techniques in scalability, efficiency, and precision, with benefits increasing on larger programs.

Although selective sensitivity – previously explored for Java [22, 20, 25, 11, 36, 8] – can in principle reduce precision, our evaluation (Section 4) shows that RCEUS achieves strong precision improvements over existing techniques. Beyond these gains, RCEUS offers a practical, scalable foundation for Rust pointer analysis: it integrates with existing optimizations such as stack filtering (SF-*kcs*) [18] and provides a robust platform for future research.

2 Motivation

Rust's ownership model and deeply layered abstractions create pointer-flow patterns that differ fundamentally from those in C, C++, or Java. These patterns lead to substantial call-graph depth and extensive reuse of heap and stack objects across many layers of trait indirection, causing traditional k-limited context-sensitive analyses to either conflate pointer flows or suffer exponential context growth. In this section, we examine these challenges in

detail and identify the key insight that motivates RCEUS: context sensitivity is required only for precision-critical functions, and for these functions it suffices to preserve their flow-entry callsites; all other functions can be analyzed context-insensitively without loss of precision.

2.1 Rust's Ownership and Layered Abstractions

Ownership, Borrowing, and Lifetimes. Rust enforces memory safety at compile time through its *ownership* model [43]. Each value has a unique owner, and the value is automatically deallocated when the owner's *lifetime* ends. Any subsequent access attempts trigger a compile-time error, preventing use-after-free. In the example given in Figure 1, the variable `v` (line 3) owns the vector, and its lifetime ends at line 10, when the value is dropped.

```
1 fn bar(s: String) {}
2 fn main(){
3     let mut v = vec![1, 2];
4     let r = &v;
5     let it = v.iter_mut();
6     // println!("{:?}", r); // Compile Error
7     let s = String::from("hello");
8     bar(s);
9     // println!("{:?}", s); // Compile Error
10 }
```

Figure 1 An example illustrating ownership, borrowing, and lifetimes in Rust.

Rust supports two key mechanisms: *moves* and *borrowing*. A move transfers ownership (line 8), invalidating the previous owner and triggering a compile-time error on subsequent use (line 9). Borrowing provides temporary access through immutable (`&v`) or mutable (`&mut v`) references, with the compiler ensuring their validity to prevent data races. Immutable borrows allow shared read-only access, whereas mutable borrows require exclusive access for modification. For example, the immutable reference `r` (line 4) conflicts with the mutable iteration `v.iter_mut()` (line 5), resulting in a compile-time error (line 6).

Heap Management with Raw Pointers and Safe Abstractions. Rust permits explicit heap access through *raw pointers*, which mediate all heap operations. Raw pointers are plain memory addresses without lifetime or borrowing guarantees, resembling C/C++ pointers. Unlike references, they bypass compiler safety checks and may lead to undefined behavior; dereferencing therefore requires an `unsafe` block.

To confine such unsafe operations, Rust adopts *encapsulated unsafety*: low-level pointer manipulation is placed inside small, well-tested abstractions that expose safe interfaces. The standard library provides smart pointers (e.g., `Box<T>`, `Rc<T>`) and containers (e.g., `Vec<T>`, `String`), which wrap raw pointers while preserving invariants through the type system. For example, `Vec<T>` and `Box<T>` encapsulate similar internal structures, as shown in Figure 2.

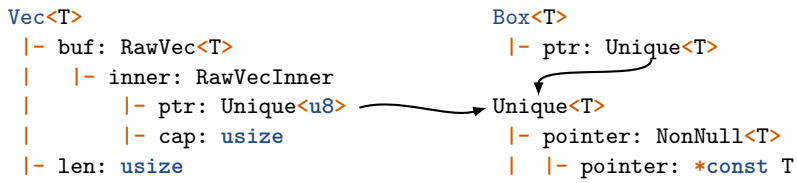

Figure 2 Structures of `Vec<T>` and `Box<T>`.

Both `Box<T>` and `Vec<T>` rely on the same underlying pointer representation. Internally, each stores a `Unique<T>` pointer built from `NonNull<T>`, which wraps a raw pointer (`*const T`). `NonNull` enforces non-nullness, while `Unique` encodes unique ownership by preventing other writable aliases. `Box<T>` uses this pointer to own exactly one heap allocation, whereas `Vec<T>` uses it to own a contiguous buffer whose length and capacity are tracked separately.

In Rust, `Box<T>` and `Vec<T>` expose safe interfaces that respect ownership and borrowing, allowing heap access through references. For example, immutable and mutable access to a boxed value is provided through safe methods such as:

```
1 fn as_ref(&self) -> &T { &**self }
2 fn as_mut(&mut self) -> &mut T { &mut **self }
```

These methods return immutable (`&T`) or mutable (`&mut T`) references, enforcing borrowing rules and preserving aliasing invariants:

```
1 let mut b = Box::new(42);
2 let r = b.as_ref(); // Immutable borrow
3 *b += 1;            // Mutable borrow (conflict)
4 println!("{r}");    // Compiler Error
```

Here, the immutable reference `r` (line 2) conflicts with the mutable borrow on line 3, causing the compile-time error on line 4. Although `Box<T>` relies on unsafe internals, its safe API preserves Rust's guarantees through compiler checking.

2.2 Why k-Limiting Falls Short

Rust's ownership model and heap abstractions give rise to layered designs: safe heap manipulation typically proceeds through chains of functions that unwrap abstractions, perform low-level operations, and rebuild safe interfaces. These patterns simplify programming but shift complexity to static analysis. We identify three Rust-specific sources of this complexity: layered heap abstractions, layered slice-centric memory access, and layered trait-driven operations. Together, they render traditional k-limited context-sensitive approaches (with context insensitivity as the special case $k = 0$) – originally developed for languages such as Java [19, 9, 45, 35] – both imprecise and inefficient, thereby limiting scalability.

1) Layered Heap Abstraction

Rust's heap encapsulation naturally leads to layered heap construction and access.

Heap Construction Through Abstraction Layers. As shown earlier in Figure 2, constructing a `Vec<T>` unfolds across multiple abstraction layers.

Figure 3 illustrates this process by creating two vectors, `v1` and `v2`, at lines 2 and 7 using the `vec![]` macro. As shown in the two groups of comments (lines 2–3 and 7–8), each macro expansion conceptually produces a `Box<[i32]>` (`b1` and `b2`) owning heap-allocated values (`o1` and `o2`). Each `Box` holds a raw pointer to the allocation, which is then passed to `into_vec()` for vector construction.

Ownership of the heap objects transfers from `b1` and `b2` to the parameter `b` in `into_vec()` (line 12). Inside `into_vec()`, the raw pointer `ptr` is extracted via `into_raw_with_allocator()` (line 14) and converted into a vector by `Vec::from_raw_parts()` (line 15). This step risks creating two owners of the same allocation: the original `Box` and the new `Vec`. To prevent a double free, the `Box` is wrapped in `ManuallyDrop`, which suppresses its destructor. This guarantees that ownership is transferred safely to the `Vec`, the allocation is freed exactly once, and Rust's memory safety is preserved.

The raw pointer `ptr` extracted at line 14 traverses five abstraction layers before becoming a vector at line 15. It is first wrapped in `NonNull` and then in `Unique` for non-null alignment and exclusive ownership. Next, `RawVecInner` couples the pointer with capacity metadata, followed by `RawVec`, which manages allocator state and growth logic. Finally, `Vec` combines the buffer with a length field to expose a safe user interface.

```
1  fn main(){ // f0
2    let v1 = vec![1, 2];  // b1 = Box::new(); // o1
3                          // v1 = into_vec(b1);
4    let mut it1 = v1.iter();
5    let val1 = it1.next();
6
7    let v2 = vec![1, 2];  // b2 = Box::new(); // o2
8                          // v2 = into_vec(b2);
9    let mut it2 = v2.iter();
10   let val2 = it2.next();
11 }
12 fn into_vec(b: Box<[i32]>)->Vec<[i32]>{ // f1
13     unsafe {
14         let mut ptr = into_raw_with_allocator(b);
15         Vec::from_raw_parts(ptr as *mut i32)
16     }
17 }
18 fn into_raw_with_allocator(b: Box<[i32]>)
   ↪ ->*mut [i32]{ // f2
19     let b = ManuallyDrop::new(b);
20     &raw mut **b
21 }
22 fn ManuallyDrop::new(b: Box<[i32]>)
   ↪ ->ManuallyDrop<Box<[i32]>>{ // f3
23     ManuallyDrop {b}
24 }
25 fn Vec::from_raw_parts(ptr)->Vec<i32>{ // f4
26     Vec {buf: RawVec::from_raw_parts_in(ptr),..}
27 }
28 fn RawVec::from_raw_parts_in(ptr)->RawVec<i32>{
   ↪ // f5
29     RawVec {inner:
       ↪ RawVecInner::from_raw_parts_in(ptr),..}
30 }
31 fn RawVecInner::from_raw_parts_in(ptr)
   ↪ ->RawVecInner{ // f6
32     RawVecInner {ptr:
       ↪ Unique::new_unchecked(ptr),..}
33 }
34 fn Unique::new_unchecked(ptr)->Unique<u8>{ // f7
35     Unique {pointer: NonNull::new_unchecked(ptr)}
36 }
37 fn NonNull::new_unchecked(ptr)->NonNull<u8>{
   ↪ // f8
38     NonNull {pointer: ptr as *const u8}
39 }
```

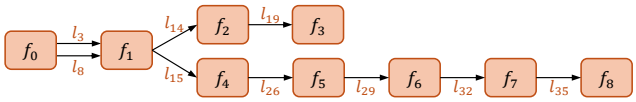

Figure 3 A Rust program demonstrating the insufficiency of k-limiting due to layered heap abstraction. `main()` creates two vectors, and `into_vec()` is simplified to highlight heap operations. Functions are labeled f_0–f_8. The `vec!` macro expands at lines 2–3 and 7–8. In the call graph below the code, $f_i \xrightarrow{l_c} f_j$ denotes a call from f_i to f_j at line l_c.

Figure 3 also shows the call graph for the example program. For clarity, the long function names in the code are abbreviated as f_0–f_8. An edge $f_i \xrightarrow{l_c} f_j$ indicates a call from f_i to f_j at line l_c. Using this call graph, we illustrate why k-limiting (with k typically restricted to small values for scalability) fails to distinguish `v1` → `o1` from `v2` → `o2`, leading to conflation.

Under context-insensitive analysis ($k = 0$), `o1` and `o2` collapse into a single abstract object when `into_vec()` is analyzed. This conflated abstraction $\{\texttt{o1}, \texttt{o2}\}$ then propagates to both `v1` and `v2` (lines 2 and 7) and to their iterators (lines 4–5 and 9–10), causing imprecision. In large Rust programs, heap accesses frequently flow through shared library functions such as `Vec` initialization, so context-insensitive analysis typically collapses many allocation sites into one abstract target. With N allocations, each points-to set grows to $O(N)$, incurring excessive propagation cost and memory overhead.

Context-sensitive analysis improves precision by distinguishing calling contexts. Callsite sensitivity [31] with k-limiting defines each context by the k most recent callsites. In our example, separating the two heap allocations requires tracking deep call chains: three callsites for `ManuallyDrop` wrapping ($l_3 \to l_{14} \to l_{19}$ and $l_8 \to l_{14} \to l_{19}$), and six callsites for vector construction ($l_3 \to l_{15} \to l_{26} \to l_{29} \to l_{32} \to l_{35}$ and $l_8 \to l_{15} \to l_{26} \to l_{29} \to l_{32} \to l_{35}$).

While 6-callsite sensitivity (i.e., full sensitivity in this example) would preserve precision, uniformly increasing k (as in kcs) quickly causes a combinatorial explosion of contexts. Selective context sensitivity (Sel-kcs) [25, 20] mitigates this by applying k-limited sensitivity only to precision-critical functions and analyzing the rest context-insensitively. In practice, however, Sel-kcs restricts $k \leq 2$ for tractability, which cannot capture the deeper contexts required by Rust's layered heap encapsulation and long call chains. Although larger k

could theoretically improve precision, increasing k rapidly becomes infeasible due to severe context explosion and memory overhead, even at $k = 2$ (Section 4). Stack filtering [18], a Rust-specific technique for pruning spurious pointed-to stack objects in k-limited analysis, does not address this issue, as it applies only to stack objects.

Heap Access Through Abstraction Layers. Rust's memory access also unfolds through deep call chains. Among Rust's heap abstractions, `Box<T>` (Section 2.1) is a shallow case: a boxed value `b` can be dereferenced simply via `*b`, and the compiler lowers this to a dereference of the raw pointer stored inside `Box<T>`, reached through its `Unique` and `NonNull` wrappers. This direct lowering explains the simplicity of `Box<T>`'s `as_ref()` and `as_mut()` implementations. In contrast, other heap-managing structures access their allocations through multiple intermediate layers. For example, `Vec<T>` reaches its heap buffer through `Unique`'s and `NonNull`'s implementations of `as_ptr()` and `as_mut_ptr()`:

```
1 fn Unique<T>::as_ptr(self) -> *mut T { self.pointer.as_ptr() }
2 fn NonNull<T>::as_ptr(self) -> *mut T { mem::transmute::<Self, *mut T>(self) }
```

As with the layered heap construction described earlier (Section 2.1), `Vec<T>`'s accessors traverse a five-layer hierarchy – `Vec`, `RawVec`, `RawVecInner`, `Unique`, and `NonNull` – with each layer delegating to its own `as_ptr()` implementation. This structure causes `Vec<T>`'s `as_ptr()` and `as_mut_ptr()` methods to require at least 5-callsite sensitivity for precise points-to reasoning. Moreover, operations within these layers may themselves invoke `as_ptr()` to obtain raw pointers, further contributing to conflation in k-limited analyses.

Such a layered heap abstraction provides ergonomics and memory safety by minimizing the use of `unsafe` code, but it obscures access paths for pointer analysis. Consequently, k-limiting becomes inadequate and causes conflation of points-to sets, even when high-level types differ. Because user-defined types follow the same design idioms, these deep and compositional access paths are pervasive in real-world Rust programs.

Rust monomorphizes all generic types: each generic definition is instantiated with concrete type parameters. For example, `Vec<u8>` and `Vec<i32>` are compiled into separate code copies [43]. However, monomorphization does not prevent severe points-to conflation, as pointers stored inside high-level types sharing the same generic type can still be conflated.

Rust's standard library further compounds this through compositional design. For example, `String` contains a `Vec<u8>` buffer with UTF-8 invariants; `OsString` uses a platform-specific vector internally (`Vec<u8>` on Unix, `Vec<u16>` on Windows); `CString` wraps a `Box<[u8]>` with C-compatible guarantees; and `PathBuf` is a thin wrapper over `OsString`. As a result, many distinct high-level types ultimately store their data in buffers of the same underlying type, causing their heap access paths to converge on the same low-level operations. This design promotes reuse and abstraction but significantly deepens and unifies the hierarchy, exacerbating points-to conflation in analysis.

2) Layered Slice-Centric Memory Access

Rust's slice abstraction exemplifies a layered memory-access mechanism that upholds borrowing and lifetime rules. Rust provides a unified abstraction for contiguous memory through its Dynamically Sized Types (DSTs), most notably slices (`[T]`) and string slices (`str`) for UTF-8 text. A slice is a safe view of any contiguous region of memory – heap, stack, or static – and is internally represented by a fat pointer containing a data pointer and metadata (e.g., length). Slice references such as `&[T]` and `&str` therefore offer lightweight, zero-cost access to contiguous data without copying or reallocating it.

Slices serve as Rust's canonical abstraction for contiguous memory. The standard library implements most contiguous-data operations – indexing, iteration, splitting, searching, and pattern matching – directly on slices. High-level containers expose their underlying memory as slice references (e.g., `&[T]` for `Vec<T>` and `&str` for `String`), inheriting this functionality rather than reimplementing it. This design centralizes memory access in one abstraction and ensures that safe operations on contiguous data follow consistent slice-based semantics.

Containers provide methods that return slice references explicitly, such as `as_slice()`, `as_mut_slice()`, and `as_str()`, yielding `&[T]`, `&mut [T]`, or `&str`, respectively. In idiomatic Rust, however, slice references are more often obtained implicitly through the `Deref` and `DerefMut` traits: taking a reference to a container automatically coerces it into the appropriate slice reference. In both cases, the conversion requires a layered reconstruction of the slice's fat pointer from the container's internal raw pointer and associated metadata.

Figure 4 illustrates Rust's use of slices. Consider the `foo()` (f_9) and `bar()` (f_{10}) functions, where `g` and `v` own the heap-allocated objects `o3` and `o4`. Because `str::is_ascii()` and `slice::get()` expect `&str` and `&[T]`, the compiler automatically performs a *deref* coercion: taking `&g` and `&v`, invoking their `deref()` implementations, and producing `&str` and `&[u8]`. These implicit `deref()` calls are inserted at lines 3 and 8, respectively.

The conversion to a slice at line 8 calls `Vec::deref()` (line 21), which invokes `Vec::as_slice()` (line 24). This retrieves the raw pointer via `as_ptr()` and reconstructs a slice reference using `slice::from_raw_parts()` (line 26). The pointer obtained at line 25 traverses several internal layers for runtime safety checks before forming the final fat pointer that combines the data pointer and length. The conversion at line 3 follows a similar pattern. `String::deref()` (line 17) obtains a `&[u8]` slice from the string's internal `Vec<u8>` buffer by calling `g.vec.as_slice()` (line 18), implemented by `Vec::as_slice()`. It then reinterprets this slice as a `&str` at line 19.

As shown in the call graph for this example, slice conversions introduce their own deep call chains in addition to those from layered heap access via `as_ptr()`. In our example, dereferencing `Vec<u8>` into `&[u8]` at line 8 involves five calls: $f_{10} \xrightarrow{l_8} f_{13} \xrightarrow{l_{22}} f_{14} \xrightarrow{l_{26}} f_{21} \xrightarrow{l_{55}} f_{22} \xrightarrow{l_{59}} f_{23}$. Similarly, dereferencing `String` into `&str` at line 3 also follows a five-call chain: $f_9 \xrightarrow{l_3} f_{12} \xrightarrow{l_{18}} f_{14} \xrightarrow{l_{26}} f_{21} \xrightarrow{l_{55}} f_{22} \xrightarrow{l_{59}} f_{23}$.

Rust's centralized slice operations apply uniformly to all contiguous memory regions – heap, stack, and static – further exacerbating points-to conflation beyond heap objects alone. In `baz()` (line 12), the reference `&a` points to the stack-allocated array `o5`. The indexing expression `&a[0..5]` is compiled into the array-specific `array::index()` (line 15), which delegates to slice-indexing utilities: it performs raw-pointer arithmetic and constructs the resulting slice's fat pointer via `ptr::slice_from_raw_parts()`. A similar indexing path is used by `slice::get()` (line 10) when slicing a `Vec<u8>`. Both `array::index()` and `slice::get()` ultimately rely on the same `SliceIndex<[u8]>::get_unchecked()` implementation and converge on `ptr::slice_from_raw_parts()` to build the subslice's fat pointer.

Based on the call graph shown in Figure 4, we see that when $k < 4$, the context-sensitive analysis fails to separate the three memory objects `o3`, `o4`, and `o5`, leading to conflated points-to information. This conflation propagates to `g_slc`, `v_slc`, the return value of `slice::get()`, and the return value of `array::index()`.

3) Layered Trait-Driven Operations

Trait-driven operations introduce additional abstraction layers beyond raw pointer manipulation. Rust's abstraction model is fundamentally trait-oriented: shared behaviors are defined once and reused across many types without relying on inheritance or dynamic dispatch.

Example Code

```
1  fn foo(){ // f9
2      let g = String::from("example"); // o3
3                                       // g_slc = String::deref(&g);
4      g.is_ascii(); // str::is_ascii(g_slc);
5  }
6  fn bar(){ // f10
7      let v: Vec<u8> = vec![1, 2, 3]; // o4
8                                      // v_slc = Vec::deref(&v);
9                                      // r = Range {start: 0,end: 2};
10     v.get(0..2); // slice::get(v_slc, r);
11 }
12 fn baz(){ // f11
13     let a = [0u8; 15]; // o5
14                        // r = Range {start: 0,end: 5};
15     &a[0..5]; // array::index(&a, r);
16 }
```

Deref Coercion

```
17 fn String::deref(g: &String)->&str{ // f12
18      let g: &[u8] = g.vec.as_slice();
19      g as &str
20 }
21 fn Vec::deref(v: &Vec<u8>)->&[u8]{ // f13
22      Vec<u8>::as_slice(v)
23 }
24 fn Vec::as_slice(v: &Vec<u8>)->&[u8]{ // f14
25     let p: *const u8 = v.as_ptr();
26     slice::from_raw_parts(p, v.len())
27 }
```

Indexing Utilities

```
28 fn slice::get(slc: &[u8], r: Range)
   ↪ ->Option<&[u8]>{ // f15
29     SliceIndex<[u8]>::get(r, slc)
30 }
31 fn array::index(a: &[u8; 15], r: Range)
   ↪ ->&[u8]{ // f16
32     let a_slc = a as &[u8];
33     Index<Range>::index(a_slc, r)
34 }
```

Index Trait

```
35 fn Index<Range>::index(slc: &[u8], r: Range)
   ↪ ->&[u8]{ // f17
36     SliceIndex<[u8]>::index(r, slc)
37 }
```

SliceIndex Trait

```
38 fn SliceIndex<[u8]>::get(r: Range, slc: &[u8])
   ↪ ->Option<&[u8]>{ // f18
39     slc = SliceIndex<[u8]>::get_unchecked(r, slc);
40     Option::Some(slc)
41 }
42 fn SliceIndex<[u8]>::index(r: Range, slc: &[u8])
   ↪ ->&[u8]{ // f19
43     // skip details of length check
44     SliceIndex<[u8]>::get_unchecked(r, slc)
45 }
46 fn SliceIndex<[u8]>::get_unchecked(r: Range,
   ↪ slc: &[u8])->&[u8]{ // f20
47     let new_len = r.end - r.start;
48     ptr::slice_from_raw_parts(
49         slc.as_ptr().add(r.start),
50         new_len
51     )
52 }
```

Slice Construction

```
53 fn slice::from_raw_parts(p: *const u8, len:
   ↪ usize)->&[u8]{  // f21
54     // skip details of runtime safety checks
55     unsafe{ &*ptr::slice_from_raw_parts(p, len) }
56 }
57 fn ptr::slice_from_raw_parts(p: *const u8, len:
   ↪ usize)->*const [u8]{ // f22
58     p = p as *const ();
59     ptr::from_raw_parts(p, len)
60 }
61 fn ptr::from_raw_parts(p: *const (), len: usize)
   ↪ ->*const [u8]{ // f23
62     let ptr = PtrComponents::<[i32]> {
63         data_pointer: p, metadata: len };
64     ptr as *const [u8]
65 }
```

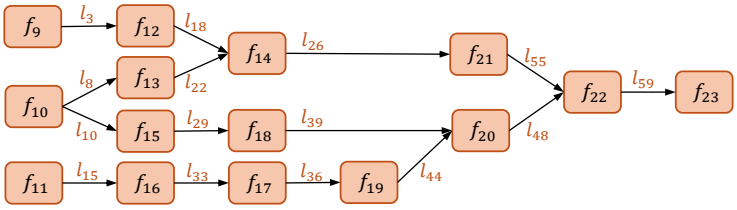

Figure 4 A Rust program illustrating the insufficiency of k-limiting under Rust's slice-centric memory-access model. The example is simplified to highlight deref coercions and the layered slice-indexing and slice-construction calls that produce slice fat pointers. Functions are labeled f_9 – f_{23}. In the call graph below the code, $f_i \xrightarrow{l_c} f_j$ denotes a call from f_i to f_j at line l_c.

Traits provide static, compositional interfaces that the compiler aggressively specializes via monomorphization, enabling zero-cost reuse of shared logic across different types.

Leveraging this architecture, the Rust standard library implements many high-level operations as sequences of delegated trait calls. A method defined on a container forwards to a trait implementation, which may in turn invoke helper traits that encode reusable semantics, forming layered chains of trait-driven computation.

The same example in Figure 4 also illustrates how trait-driven operations introduce their own layered call chains. Slices provide the implementation of the `Index` trait for range-based indexing. The `Index` trait defines the public interface for indexing and delegates the actual

slicing logic to the subslicing machinery. For the operator `&a[0..5]` (line 15) in `baz()` (f_{11}), the compiler constructs a `Range` object (line 14) and routes the array and range through the array's `Index` implementation (f_{16}). The array reference is coerced into a slice reference so that slice-based indexing can be applied. Specifically, `Index<Range>::index()` (f_{17}) forwards the operation to the helper trait `SliceIndex` via `SliceIndex<[u8]>::index()` (f_{19}), which centralizes range-based indexing across slice-backed types. The final subslice is produced by `SliceIndex<[u8]>::get_unchecked()` (f_{20}).

As shown in the call graph, this delegation passes through four intermediate layers ($l_{15} \to l_{33} \to l_{36} \to l_{44}$), starting at f_{11} and ending at f_{20}, before reaching `ptr::slice_from_raw_parts()` (f_{22}) at line 48 (l_{48}) for subslice construction.

Such layering is not unique to slice operations; many core Rust APIs follow the same trait-driven delegation. For instance, (1) the `Iterator` API uses an adaptor pattern in which each adaptor (e.g., `Enumerate`, `Zip`, `Flatten`, `Rev`) wraps the previous iterator, forming a nested chain of adaptor layers. Any call to `next()`, `next_back()`, or `nth()` must traverse this entire stack, triggering a cascade of trait-method dispatches. (2) Error propagation also proceeds through a sequence of trait calls rather than a language-level exception mechanism: the `Try` trait separates success from failure, and the `FromResidual` trait adapts the error type before early return, introducing at least two layers of calls.

These high-level layers frequently converge on shared low-level implementations, further amplifying the conflation effects already introduced by Rust's heap abstractions and slice-centric memory model.

2.3 Pointer-Flow-Guided Context Sensitivity Beyond k-Limiting

Existing context-sensitive pointer analyses rely on k-limiting, applied uniformly [19, 9, 45, 35] or selectively [22, 20, 25, 11, 36]. Several Java-oriented techniques [10, 12, 41] further choose k "important" callsites rather than the k-most-recent ones (Section 1). These methods depend on heuristics tied to Java's object and call-structure regularities, whereas Rust's ownership, borrowing, and deep compositional pointer-flow patterns follow fundamentally different structures. As a result, both uniform k-limiting and Java-based callsite-selection heuristics remain inadequate for Rust, particularly for precise heap reasoning.

To address this gap, we introduce Rceus, a Rust-oriented context-sensitivity approach guided by interprocedural pointer-flow information from a lightweight pre-analysis. This pre-analysis identifies the functions whose points-to results depend on calling contexts and determines the flow-relevant (i.e., flow-entry) callsites that distinguish distinct pointer-flow origins. The subsequent main analysis preserves only these callsites – one per calling context – avoiding large k or uniform context expansion while preventing points-to conflation, yielding scalable and precise context sensitivity aligned with Rust's abstraction patterns.

1) Precision-Critical Functions

Rust's multi-layered pointer operations form chains of functions that take pointers as input and produce new pointers, propagating information across abstraction levels. At the lower layers of these chains are routines such as `Unique::new_unchecked()` and `NonNull::new_unchecked()` (see Figure 3), which directly wrap raw pointers. Because many higher-level abstractions eventually call these routines to initialize or reconstruct internal pointers, they become convergence points for diverse pointer flows. Such functions must therefore be analyzed under distinct contexts to avoid conflating unrelated points-to sets; a context-insensitive treatment would merge flows arriving along different call paths.

In this paper, we identify a function as *precision-critical* if its return value depends on pointer flows originating from its parameters, as determined by a lightweight interprocedural pointer-flow pre-analysis. Such functions require context sensitivity to avoid merging unrelated pointer flows arriving along different call paths. Concretely, RCEUS classifies a function as precision-critical when its return value depends on one of its parameters through either (1) intraprocedural flows or (2) interprocedural flows.

In Figure 3, functions f_3 (`ManuallyDrop::new()`) and f_8 (`NonNull::new_unchecked()`) illustrate the first case, with return values derived directly from their parameters. By contrast, all other functions (except f_0, `main()`) fall into the second case, where parameter-derived pointer information propagates through one or more callees before contributing to the function's return.

This pointer-flow-based strategy identifies precision-critical functions approximately but effectively. Although it may misclassify a function in either direction in theory, in practice applying k-limited context sensitivity only to these functions achieves precision nearly identical to uniform k-limiting while substantially reducing computational cost (as validated in our evaluation), demonstrating the practical accuracy of the classification.

2) Flow-Entry Callsites

Rust's deeply layered abstractions make traditional k-limited context-sensitive pointer analyses ineffective: small k values fail to preserve the calling structure needed to separate distinct pointer flows, whereas increasing k quickly becomes prohibitively expensive. To address this, RCEUS preserves only the flow-entry callsites – one per context – that distinguish how parameter-derived pointer flows reach each precision-critical function, providing the necessary separation while keeping the analysis both scalable and precise.

In Rust's layered abstractions, pointer flows descend from higher-level precision-critical functions into lower-level ones and eventually return upward through nested abstraction layers. The first callsite where a pointer enters a precision-critical function provides the only context that distinguishes flows reaching the same callee from different callers. Once a pointer has propagated into deeper layers, downstream callsites no longer encode its origin and thus are ineffective as context boundaries for separating flows from different call paths.

Motivated by this observation, we define a *flow-entry callsite* as a callsite where a pointer first enters an upper-layer precision-critical function. For each function transitively reached downstream along this pointer flow, we preserve the corresponding flow-entry callsite as its context with respect to that precision-critical function. Propagating this flow-entry callsite along the entire call chain consistently encodes the pointer's origin, keeping flows from different callers distinguishable even deep within lower-layer routines.

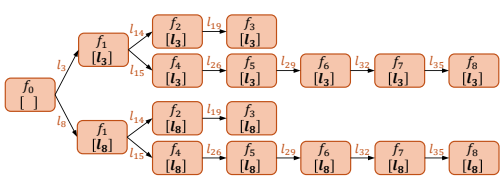

Figure 5 Context-sensitive call graph generated by RCEUS for the example in Figure 3.

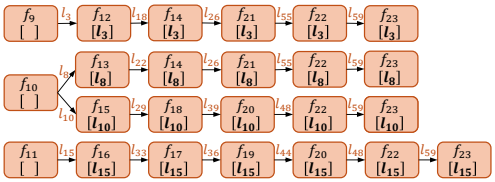

Figure 6 Context-sensitive call graph generated by RCEUS for the example in Figure 4.

In Figure 3, f_1 (`into_vec()`) is the topmost precision-critical function. Two pointer flows, originating from `b1` and `b2`, enter it at callsites l_3 and l_8 in f_0 (`main()`), respectively. Each flow then descends to f_3 along $l_{14} \to l_{19}$, binds the returned value to `ptr`, and continues

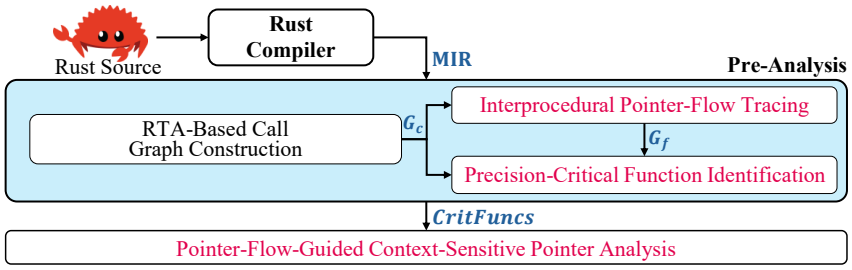

Figure 7 Workflow of RCEUS with new components shown in red.

through f_4 – f_8 along $l_{15} \to l_{26} \to l_{29} \to l_{32} \to l_{35}$ before returning upward. These two flows ultimately reach `v1` and `v2` separately, making l_3 and l_8 the flow-entry callsites for the two distinct pointer flows. Figure 5 shows the context-sensitive call graph generated by RCEUS, where each context is written as $[\ldots]$. Call chains rooted at l_3 are annotated with context $[l_3]$, and those rooted at l_8 with context $[l_8]$. By preserving these flow-entry callsites, RCEUS keeps the two pointer-propagation paths separated throughout the analysis, preventing conflation of the points-to information for `v1` and `v2`.

Similarly, in Figure 4, f_{12} (`String::deref()`), f_{13} (`Vec::deref()`), f_{15} (`slice::get()`), and f_{16} (`array::index()`) serve as the topmost precision-critical functions. Their callsites l_3, l_8, l_{10}, and l_{15} are therefore identified as the flow-entry callsites. As shown in Figure 6, the context-sensitive call graph produced by RCEUS preserves these entry contexts, allowing the analysis to cleanly separate pointers to both heap-allocated and stack-allocated objects.

3 The RCEUS Design

RCEUS, shown in Figure 7, is integrated into the Rust compiler as an MIR analysis pass. We first define a core Rust subset for pointer analysis (Section 3.1). A lightweight pre-analysis (Section 3.2), based on Rapid Type Analysis [1, 18], over-approximates the call graph, traces pointer flows through Rust's layered abstractions, and identifies the precision-critical functions requiring context-sensitive treatment. RCEUS then builds on the RUPTA framework [19] to generate pointer-flow-guided calling contexts for these functions on the fly during the main analysis (Section 3.3), enabling scalable and precise context sensitivity.

3.1 Core Rust Subset for Pointer Analysis

To support precise pointer analysis, we introduce a core language model derived from Rust MIR, shown in Figure 8. MIR is a structured mid-level IR that retains essential program semantics while abstracting away surface syntax, and is widely used in prior analysis and verification work [23, 14, 19, 18]. Our model isolates pointer-relevant constructs and omits extraneous features, providing a focused foundation for RCEUS's analysis.

In Rust MIR, a `Place` denotes a stack-allocated memory location, consisting of a base local variable v and an optional projection sequence j for field or index access. The projection j is either empty (ϵ) or extended by a projection element e representing a field or array index.

Statements capture the fundamental pointer-related memory operations, including assignments ($p_1 = p_2$), which copy or move values between places; references ($p_1 = \&p_2$), which create references or raw pointers to existing values; casts ($p_1 = p_2$ *as* τ), which reinterpret values under a new type τ; and dereference forms ($(*v).j = p$, $p = (*v).j$, $p = \&(*v).j$), which make explicit memory accesses by reading from, writing to, or taking the address of subfields through pointers. Function calls ($p = f(p_1, p_2, \ldots)$) denote direct or indirect invocations, where p_1 denotes the `self` parameter if present.

Local	$v \in \{v_0, v_1, \ldots\}$
Type	$\tau ::= \mathit{usize} \mid \mathit{i32} \mid \mathit{u64} \mid \cdots$
Function	$f \in \{f_0, f_1, \ldots\}$
ProjElement	$e ::= \mathsf{field}(\mathit{idx}) \mid \mathsf{index} \mid \cdots$
Projection	$j ::= \epsilon \mid e.j$
Place	$p ::= v.j$
Statement	$s ::= s_1; s_2 \mid p_1 = \&p_2 \mid p_1 = p_2 \mid p_1 = p_2 \ \mathit{as}\ \tau \mid$ $(*v).j = p \mid p = (*v).j \mid p = \&(*v).j \mid p = f(p_1, p_2, \ldots)$

Figure 8 Core Rust MIR syntax for the subset of language features relevant to pointer analysis.

3.2 Lightweight Pre-Analysis

RCEUS's pre-analysis consists of three steps: (1) RTA-based call graph construction, (2) interprocedural pointer-flow tracing, and (3) identification of precision-critical functions.

1) RTA-Based Call Graph Construction

We build an over-approximate call graph using Rapid Type Analysis (RTA) [1], adapted to Rust following [18]. RTA is a fast whole-program technique that resolves potential calls from the set of types instantiated in reachable code [1, 39, 46, 33]. The Rust-adapted variant further supports static dispatch, trait-object dispatch, trait upcasting, and function-pointer calls [18]. The resulting call graph $G_c = (F_c, E_c)$ forms the basis of our analysis, where F_c is the set of RTA-reachable functions and $E_c \subseteq F_c \times F_c \times \mathbb{L}$ contains edges $f \xrightarrow{\ell} g$ for calls at callsites $\ell \in \mathbb{L}$, with $f, g \in F_c$, and $\mathbb{L}$ denoting the set of all program locations.

2) Interprocedural Pointer-Flow Tracing

Given the RTA call graph $G_c = (F_c, E_c)$, we trace interprocedural pointer flows for each function $f \in F_c$ by constructing a *pointer-flow graph* $G_f = (V_f, E_f)$, where V_f contains the local variables of f and E_f represents flows induced by assignments, dereferences, address-of expressions, loads/stores, and calls in f. This tracing need not be sound: missing flows merely cause some functions to be treated context-insensitively by RCEUS, reducing precision but not the soundness of the main analysis. The goal is simply to capture the essential flows needed to identify precision-critical functions for context-sensitive treatment.

For $u, v \in V_f$, where f is an arbitrary function, we write $\pi : u \rightsquigarrow_{G_f} v$ for a program path in G_f, and $\mathit{labels}(\pi)$ for its set of *edge labels*. All intraprocedural edges are labeled ϵ, while interprocedural edges are labeled with their callsites. Finally, let param_f^i denote the i-th parameter of f, and ret_f its return variable.

Figure 9 summarizes the rules used to extract coarse-grained interprocedural pointer flows in Rust. The first six rules capture intraprocedural flows by modeling assignments derived from the MIR syntax in Figure 8, where each place p is instantiated as $v.j$ (a local variable v with an optional projection j). The final rule models interprocedural flow by propagating pointer information across call statements.

Let us first examine how RCEUS traces intraprocedural flows. The [P-ADDROF] rule models the creation of references or raw pointers ($v_1.j_1 = \&v_2.j_2$). References (e.g., `let x = &y`) are fundamental to safe memory access, whereas raw pointers arise through coercions (e.g., `&v as *const _`) or explicit syntax (e.g., `&raw const v`) and are common in unsafe code. [P-ASSIGN] captures assignments $v_1.j_1 = v_2.j_2$, covering ownership moves, reference copies,

$$\frac{v_1.j_1 = \&v_2.j_2}{v_2 \to v_1 \in E_f}\ [\text{P-AddrOf}] \qquad \frac{v_1.j_1 = v_2.j_2}{v_2 \to v_1 \in E_f}\ [\text{P-Assign}] \qquad \frac{v_1.j_1 = v_2.j_2 \text{ as } \tau}{v_2 \to v_1 \in E_f}\ [\text{P-Cast}]$$

$$\frac{v_1.j_1 = \&(*v_2).j_2}{v_2 \to v_1 \in E_f}\ [\text{P-Gep}] \qquad \frac{v_1.j_1 = (*v_2).j_2}{v_2 \to v_1 \in E_f}\ [\text{P-Load}] \qquad \frac{(*v_1).j_1 = v_2.j_2}{v_2 \to v_1 \in E_f}\ [\text{P-Store}]$$

$$\frac{\begin{array}{c}\ell:\ v_0 = g(v_1,\ldots,v_r) \quad f,g \in F_c \quad f \xrightarrow{\ell} g \in E_c \\ i \in [1,r],\ \pi : \mathit{param}_g^i \rightsquigarrow_{G_g} \mathit{ret}_g\end{array}}{v_i \xrightarrow{l} v_0 \in E_f}\ [\text{P-Call}]$$

Figure 9 Rules for interprocedural pointer-flow tracing (statements belong to function f).

and raw-pointer duplication. In MIR, array and struct copies are element-wise; therefore, we add a flow from v_2 to v_1 whenever the assigned value is a pointer or contains pointer fields. [P-Cast] handles pointer casts ($v_1.j_1 = v_2.j_2$ *as* τ), frequently used in libraries and low-level code manipulating raw pointers. [P-Gep] models field access through a dereferenced pointer ($v_1.j_1 = \&(*v_2).j_2$), common in struct and container access. [P-Load] captures projection-based loads ($v_1.j_1 = (*v_2).j_2$), typical when reading heap fields through references. Conversely, [P-Store] models field updates ($(*v_1).j_1 = v_2.j_2$), where a value is written into a field reachable from a pointer – a pattern frequent in low-level libraries.

Finally, [P-Call] lifts these flows across function boundaries. It models interprocedural propagation between call arguments and the value assigned at a callsite. For a callsite $\ell : v_0 = g(v_1,\ldots,v_r)$ in function f with a call-graph edge $f \xrightarrow{\ell} g \in E_c$, if, in g, the parameter param_g^i can reach its return value (i.e., $\mathit{param}_g^i \rightsquigarrow_{G_g} \mathit{ret}_g$), then we add an interprocedural edge $v_i \xrightarrow{\ell} v_0$ to G_f, indicating that argument v_i contributes to the callsite's assigned value.

3) Precision-Critical Function Identification

Given each function's pointer-flow graph $G_f = (V_f, E_f)$, Rceus computes the set *CritFuncs* of *precision-critical functions* – those whose return values depend on pointer flows originating from their parameters. This dependency is encoded directly in G_f: if any parameter can reach the return variable along a pointer-flow path, f is classified as precision-critical.

As formalized in Figure 10, the [C-Intra] and [C-Inter] rules classify functions based on whether the parameter-to-return dependence is purely intraprocedural or arises through interprocedural flows. A function f is added to *CritFuncs* whenever there exists a path $\pi : \mathit{param}_f^i \rightsquigarrow_{G_f} \mathit{ret}_f$. If $\mathit{labels}(\pi) = \{\epsilon\}$, we record $\epsilon \Rightarrow f$, indicating an intraprocedural dependence. If π traverses any callsite l (i.e., $l \in \mathit{labels}(\pi)$), we record $l \Rightarrow f$, indicating that the parameter flows to the return value via an interprocedural call.

A function may therefore receive multiple such annotations, reflecting distinct flow paths through which its parameters may influence its return value.

In Figure 3, all functions except `main()` are precision-critical. For example, `into_vec()` is classified as such because its parameter flows to its return value interprocedurally via $\ell_{14} \Rightarrow$ `into_vec()` and $\ell_{15} \Rightarrow$ `into_vec()`. In Figure 4, all except `foo()`, `bar()`, `baz()`, and `is_ascii()` are precision-critical. In particular, `String::deref()` is precision-critical due to the interprocedural parameter-to-return flow across $\ell_{18} \Rightarrow$`String::deref()`.

Time Complexity. The pre-analysis runs in time linear in the number of MIR statements. Pointer-flow tracing constructs a pointer-flow graph for each function and computes parameter-to-return reachability in $O(V_f + E_f)$ time, where both the number of nodes and edges are linear

$$\frac{\pi : param_f^i \rightsquigarrow_{G_f} ret_f \quad labels(\pi) = \{\epsilon\}}{\epsilon \Rightarrow f \in CritFuncs} \quad \text{[C-Intra]}$$

$$\frac{\pi : param_f^i \rightsquigarrow_{G_f} ret_f \quad \ell \in labels(\pi)}{\ell \Rightarrow f \in CritFuncs} \quad \text{[C-Inter]}$$

Figure 10 Rules for identifying precision-critical functions.

$$\frac{\ell : v = \text{alloc}(\tau) \quad c \in ctx(f)}{\langle c, o_\ell : \tau \rangle \in pt(c, v)} \quad \text{[AllocHeap]} \qquad \frac{\ell : p_1 = \&p_2 \quad c \in ctx(f)}{\langle c, p_2 \rangle \in pt(c, p_1)} \quad \text{[Addrof]}$$

$$\frac{\ell : p_1 = p_2 \quad c \in ctx(f) \quad j = ptr\_proj(p_2)}{pt(c, p_2.j) \subseteq pt(c, p_1.j)} \quad \text{[Assign]}$$

$$\frac{\ell : p = \&(*v).j \quad c \in ctx(f) \quad \langle c', o \rangle \in pt(c, v)}{\langle c', o.j \rangle \in pt(c, p)} \quad \text{[Gep]}$$

$$\frac{\ell : p = (*v).j \quad c \in ctx(f) \quad \langle c', o \rangle \in pt(c, v) \quad j' \in ptr\_proj(p)}{pt(c', o.j.j') \subseteq pt(c, p.j')} \quad \text{[Load]}$$

$$\frac{\ell : (*v).j = p \quad c \in ctx(f) \quad \langle c', o \rangle \in pt(c, v) \quad j' \in ptr\_proj(p)}{pt(c, p.j') \subseteq pt(c', o.j.j')} \quad \text{[Store]}$$

$$\frac{\ell : p_1 = p_2 \text{ as } \tau \quad c \in ctx(f) \quad \langle c', o \rangle \in pt(c, p_2) \quad \tau' = deref\_ty(\tau)}{o : \tau' = transmute(c', o, \tau') \quad \langle c', o : \tau' \rangle \in pt(c, p_1)} \quad \text{[Cast]}$$

$$\frac{\begin{array}{c}\ell : p_0 = f'(p_1, \ldots, p_r) \quad c \in ctx(f) \quad g \in dispatch(c, f', p_1) \\ c' = \textit{rceus\_ctx}(f, c, \ell, g)\end{array}}{\begin{array}{c}\forall i \in [1, r],\ \forall j_i \in ptr\_proj(p_i) : pt(c, p_i.j_i) \subseteq pt(c', param_g^i.j_i) \\ \forall j_0 \in ptr\_proj(p_0) : pt(c', ret_g.j_0) \subseteq pt(c, p_0.j_0)\end{array}} \quad \text{[Call]}$$

where $param_g^i$ denotes the i-th parameter of g, and ret_g its return variable

Figure 11 Rules for Rust callsite-sensitive pointer analysis (each rule applies to statements in function f), with *rceus_ctx* preserving the appropriate flow-entry callsite for context construction.

in the function size. Recursive and mutually recursive functions are handled via an incremental fixed-point computation over reachability. Precision-critical function identification incurs constant time per function in practice, as it directly queries the reachability information computed during tracing.

3.3 Pointer-Flow-Guided Context-Sensitive Analysis

Rceus performs context-sensitive pointer analysis guided by interprocedural pointer flows. Only precision-critical functions are analyzed context-sensitively, and their calling contexts are built from the flow-entry callsites introduced earlier in Section 2.3 – one per calling context – achieving scalable precision without relying on large-k sensitivity. We next present the analysis formulation and the construction of these flow-guided contexts.

1) Analysis Formulation

In context-sensitive pointer analysis, the MIR assignment forms in Figure 8 are abstracted through the eight pointer-flow rules shown in Figure 11. Each allocation site $\ell \in \mathbb{L}$ introduces a fresh abstract object $o_\ell \in \mathbb{O}$, with type τ written as $o_\ell : \tau$ when needed, where $\mathbb{O}$ denotes the set of pointed-to memory locations. Calling contexts c range over either the empty context [ ] or a singleton $[\ell]$ containing exactly one callsite.

The following auxiliary functions are used in these rules:

- $pt(c, p)$ – the context-sensitive points-to set of pointer p under context c.
- $ctx(f)$ – the set of contexts under which function f is analyzed.
- $ptr\_proj(p)$ – the set of pointer-typed projections of place p (e.g., struct fields, tuple components, slice elements), including ϵ for no projection.
- $deref\_ty(\tau)$ – the dereferenced type of a pointer type τ.
- $transmute(c, o, \tau)$ – the τ-specific variant of object o under context c.
- $dispatch(c, f', p)$ – the set of call targets resolved for a call to f' under context c.
- $rceus\_ctx(f, c, \ell, g)$ – the context for callee g at callsite ℓ in f under context c.

The eight inference rules in Figure 11 for analyzing a function f follow the standard formulation [19, 18]; RCEUS is distinguished by how it computes calling contexts. [ALLOCHEAP] introduces a fresh object o_ℓ for each allocation. [ASSIGN] propagates points-to sets from p_2 to p_1, and [ADDROF] records that p_1 points to p_2. [GEP] computes field addresses, [LOAD] reads field contents, and [STORE] writes them. [CAST] performs type reinterpretation by creating type-specific variants of objects. Finally, [CALL] resolves callees at callsite ℓ under context $c \in ctx(f)$ via *dispatch*, constructs for each callee g a new context $c' = rceus\_ctx(f, c, \ell, g)$ that preserves only the appropriate flow-entry callsite, transfers arguments to formals, and propagates return values back to the caller, thereby enabling the analysis of g under context c'. Below we define $rceus\_ctx$, which determines the callee's context.

2) Context Construction

The auxiliary function $rceus\_ctx(f, c, \ell, g)$, central to RCEUS, computes the context used to analyze a callee g invoked at callsite ℓ in caller f under caller context c:

$$c' = rceus\_ctx(f, c, \ell, g) = \begin{cases} [\,] & \text{if } \_ \Rightarrow g \notin \mathit{CritFuncs} \\ [\ell] & \text{else if } \ell \Rightarrow f \notin \mathit{CritFuncs} \\ c & \text{otherwise} \end{cases} \tag{1}$$

where the three cases are evaluated sequentially, following an if–elseif–else structure:

- **Case 1.** $c' = [\,]$: g is not precision-critical (i.e., $\_ \Rightarrow g \notin \mathit{CritFuncs}$) and is therefore analyzed context-insensitively (i.e., under the empty context $[\,]$).
- **Case 2.** $c' = [\ell]$: g is precision-critical (i.e., $\_ \Rightarrow g \in \mathit{CritFuncs}$), but the callsite ℓ does not lie on any parameter-to-return flow in f (i.e., $\ell \Rightarrow f \notin \mathit{CritFuncs}$). Thus, any pointer reaching g must originate within f rather than from its callers, making ℓ the appropriate flow-entry callsite for g. Thus, RCEUS assigns context $[\ell]$ to g.
- **Case 3.** $c' = c$: Both f and g are precision-critical (i.e., $\_ \Rightarrow g \in \mathit{CritFuncs}$ and $\ell \Rightarrow f \in \mathit{CritFuncs}$), and ℓ lies on an interprocedural parameter-to-return flow that passes through g. The corresponding flow-entry callsite is already encoded in f's context c, and this flow continues through g. Hence, g inherits the caller's context c.

Consider the program in Figure 3. All functions except `main()` are precision-critical. Let us see how their contexts, as shown in Figure 5, are derived. For **Case 2**, the calls to `into_vec()` at lines ℓ_3 and ℓ_8 occur under the empty context $[\,]$, so RCEUS assigns $[\ell_3]$ and $[\ell_8]$, respectively – each marking a distinct flow-entry callsite. For **Case 3**, when `into_vec()` is reached under $[\ell_3]$ or $[\ell_8]$, its precision-critical callees (e.g., `Vec::from_raw_parts()` at ℓ_{15}) inherit the same context. Thus, the call at ℓ_{15} is analyzed under $[\ell_3]$ when the flow originates from ℓ_3, and under $[\ell_8]$ when it originates from ℓ_8. This propagation continues through deeper precision-critical calls at $\ell_{26}, \ell_{29}, \ell_{32}$, and ℓ_{35}. RCEUS therefore preserves the flow-entry callsite throughout the precision-critical call chain, ensuring that all heap allocations remain associated with their correct origins (either ℓ_3 or ℓ_8).

```
1 fn main() {
2     let v1 = vec![1, 2, 3];
3     let v2 = vec![4, 5, 6];
4     let z = zip(&v1, &v2);
5 }
```

```
6 fn zip(v1: &Vec, v2:  &Vec)->Zip<IntoIter, IntoIter>{
7     Zip::new(v1.into_iter(), v2.into_iter())
8 }
9 fn Zip::new(a: IntoIter, b: IntoIter)->Zip<IntoIter, IntoIter>{
10     Zip {a: a, b: b, ...}
11 }
```

Figure 12 A program where both `a.into_iter()` and `b.into_iter()` are analyzed under $[\ell_4]$, conflating their points-to sets; avoiding this requires $k = 5$ under standard k-limiting.

Consider now the program in Figure 4. All functions except `foo()`, `bar()`, `baz()`, and `is_ascii()` are precision-critical. Let us see how their contexts, as shown in Figure 6, are derived. For **Case 1**, `is_ascii()` (called at ℓ_4) is not precision-critical and thus analyzed context-insensitively. For **Case 2**, the calls to `String::deref()` at ℓ_3, `Vec::deref()` at ℓ_8, `slice::get()` at ℓ_{10}, and `array::index()` at ℓ_{15} each occur under the empty context and therefore generate the contexts $[\ell_3]$, $[\ell_8]$, $[\ell_{10}]$, and $[\ell_{15}]$ for analyzing their respective callees. These four flow-entry callsites ensure complete separation of the corresponding pointer flows.

Unlike standard k-limiting [19, 9, 45, 35], which requires a sufficiently large k to keep these call chains distinct ($k \geqslant 6$ for Figure 3 and $k \geqslant 4$ for Figure 4), RCEUS separates the same call paths using only a single flow-entry callsite per context.

3) Discussion

RCEUS is sound: regardless of the pre-analysis, the rules in Figure 11 ensure that every function reachable from `main()` is analyzed under at least one context.

Although RCEUS is presented as an enhancement to standard k-limited context-sensitive pointer analysis, it can also be used as a drop-in improvement to existing k-limited techniques, substantially boosting both scalability and precision (Section 4). This enables analyses to retain tunable precision without paying the high cost of large-k sensitivity.

RCEUS is designed for achieving scalable precision in Rust by applying context sensitivity only where it matters – namely, to precision-critical functions – and by representing each calling context with exactly one flow-entry callsite. As a result, RCEUS does not aim to be strictly more precise than uniform k-limiting, nor does it preserve its precision in every case; rather, it achieves a different, Rust-oriented balance that mitigates points-to blow-up while retaining sufficient precision for real-world analyses.

A small example in Figure 12 illustrates this tradeoff. Since RCEUS identifies ℓ_4 as the sole flow-entry callsite for both `zip()` and the two `into_iter()` calls, all iterator constructions are analyzed under the same context $[\ell_4]$, conflating the points-to sets for `a` and `b`. Avoiding this imprecision would require $k = 5$ under standard k-limiting.

Crucially, this imprecision is narrowly scoped: it affects only the two iterator fields of the same `Zip` object, does not propagate beyond this context, and does not hinder RCEUS's ability to eliminate the far more severe points-to blow-up caused by Rust's deeply layered abstractions. While this imprecision could be reduced by enriching the context with parameter indices, doing so yields limited additional precision in practice while increasing analysis cost. We therefore adopt the current design as a better balance between precision and efficiency, and view parameter-aware contexts as a direction for future work.

4 Evaluation

We evaluate RCEUS on real-world Rust programs to demonstrate that it advances the state of the art in Rust pointer analysis by substantially improving scalability, efficiency, and precision. Our study is organized around four research questions:

- **RQ1 (Pre-Analysis Cost).** How efficient is RCEUS's pre-analysis phase, and what overhead does it introduce?
- **RQ2 (Scalability, Efficiency, and Precision).** Does RCEUS improve precision compared to existing techniques while also enhancing scalability and efficiency?
- **RQ3 (Extensibility).** Can RCEUS integrate with existing context-sensitivity optimizations to further improve analysis performance?
- **RQ4 (Impact of Flow-Entry Preservation).** How does preserving flow-entry callsites contribute to the performance benefits observed in RCEUS?

Implementation. We implement RCEUS atop RUPTA [19], an open-source framework for k-limited callsite-sensitive pointer analysis on MIR. RCEUS can be applied to a GitHub-based Rust project using a single command (`cargo pta`), provided that the project is a binary crate with a `main()` entry point and compiles under the supported Rust compiler version. The latest RUPTA release targets `rustc` 1.78.0-nightly. Our prototype adds about 1000 lines of Rust code, demonstrating that RCEUS's core idea is compact and integrates cleanly into existing analysis infrastructure.

Baselines. We compare RCEUS against three state-of-the-art techniques. Two are Rust-specific: RUPTA [19], the standard k-limited (*kcs*) analysis, and SF-*kcs* [18], which augments *kcs* with stack filtering to eliminate spurious stack objects. The third baseline, SEL-*kcs*, is a selective context-sensitivity technique originally developed for Java [25, 20] and adapted with Rust extensions; we include it for completeness but do not treat it as a contribution. In SEL-*kcs*, a function f is analyzed context-sensitively under k-limiting if it is precision-critical (i.e., $\_ \Rightarrow f \in \mathit{CritFuncs}$), and context-insensitively otherwise (Figures 9 and 10).

All baselines are evaluated with $k \in \{1, 2\}$, the standard setting in prior work [19, 9, 45, 35]. We also evaluate RCEUS integrated with each baseline under the same $k \in \{1, 2\}$.

Table 1 Real-world Rust projects used in our evaluation.

Project ID	Project	#LOC	#Stmts	Description
EP1	exa	25K	104K	File listing (`ls`) utility.
EP2	zoxide	40K	220K	Smarter `cd` command.
EP3	dust	54K	283K	Disk space analyzer.
EP4	lsd	62K	349K	Modern `ls` replacement.
EP5	bandwhich	62K	314K	Network utilization CLI.
EP6	rustscan	65K	383K	Fast port scanner.
EP7	navi	76K	434K	Interactive cheat sheet tool.
EP8	resvg	84K	379K	SVG rendering library.
EP9	fselect	91K	436K	File search with SQL-like queries.
EP10	mdbook	106K	666K	Documentation generator.
EP11	gitui	109K	569K	Git terminal UI.
EP12	grin	116K	789K	Cryptocurrency node.
EP13	atuin	150K	849K	Shell history manager.
EP14	meilisearch	196K	1919K	Search engine.
EP15	qdrant	207K	2044K	Vector database.
EP16	wasmtime	669K	2078K	WebAssembly runtime.

Benchmarks. We evaluate RCEUS on 16 open-source Rust projects, listed in Table 1 (denoted EP1–EP16). The benchmark suite is selected based on three criteria: comparability with prior work, technical feasibility, and diversity. It includes all 13 projects used in prior work [19, 18], along with three large systems from different domains – `meilisearch` (196K LOC), `qdrant` (207K LOC), and `wasmtime` (669K) – to strengthen practical relevance. All projects are binary crates that compile under `rustc` 1.78.0-nightly. Column 3 reports the LOC reachable from `main()` (via *SF-1cs*), which is smaller than the full project size as unused code is excluded. Column 4 reports the total number of MIR statements after monomorphization.

Metrics. We measure efficiency using total analysis time, reported as the arithmetic mean of three runs. For precision, we adopt two standard metrics from prior work [19, 18]. The first is the average points-to set size per pointer (`#avg-pts`), computed after collapsing contexts in context-sensitive results. The second is the number of dynamically resolved call edges (`#dce`), capturing calls through trait objects and function pointers. In Rust, `#avg-pts` is generally the more informative metric, since dynamic dispatch is far less common than in object-oriented languages such as Java or C++; indeed, Rust documentation explicitly recommends static dispatch [42, 44].

We report peak memory consumption as the arithmetic mean of three runs, obtained by periodically sampling each analysis process's resident set size (RSS).

Experimental Settings. All experiments were conducted on an Intel Xeon 3.50 GHz machine with 512 GB RAM, using a two-hour timeout per analysis. Average speedups (reported as geometric means) over less scalable baselines are computed only on benchmarks that the baseline completes within the timeout, ensuring fairness since a more scalable analysis can always handle a superset of those programs.

Main Results. Table 2 summarizes the overall outcomes of our evaluation – including pre-analysis cost as well as scalability, efficiency, and precision relative to the three baselines – and provides the basis for answering RQ1–RQ4 in the following subsections.

4.1 RQ1: Pre-Analysis Cost

In Table 2, Column 2 reports the pre-analysis times for all 16 benchmarks. Both SF-*kcs* and Rceus construct the same RTA-based call graph before running their respective pre-analyses; thus, the reported time includes call-graph construction plus the method-specific pre-analysis ("SF-*kcs* adt" and "Rceus adt"). For SF-*kcs*, "adt" refers to function reachability analysis, while for Rceus it refers to the combined cost of pointer-flow tracing and precision-critical function identification.

Like SF-*kcs*, Rceus's pre-analysis is lightweight, contributing only a small fraction of the total analysis time, particularly on the largest benchmarks. Accordingly, in Table 2 (under "Time (s)"), we report only main-analysis times, as including pre-analysis would not affect any relative speedup comparisons.

4.2 RQ2: Scalability, Efficiency, and Precision

Across all 16 benchmarks, Rceus consistently outperforms the three baselines – *kcs*, SF-*kcs*, and Sel-*kcs*– in scalability, efficiency, and precision. The only exception, discussed below, is `meilisearch`, where Rceus yields a much smaller `#avg-pts` but a larger number of `#dce` than *SF-1cs* and *SEL-1cs* (*1cs* is unscalable) at $k = 1$.

Sel-*kcs* improves the efficiency of *kcs* but does not improve precision, while SF-*kcs* improves precision but is less efficient than Sel-*kcs* on most benchmarks.

By analyzing only precision-critical functions and using flow-entry callsites as their contexts, Rceus achieves substantially better scalability and efficiency while simultaneously delivering higher precision.

1) Rceus vs. *k*-Limiting

For **scalability**, *kcs* fails to complete within the memory budget on the largest benchmarks – including `meilisearch` (*1cs* and *2cs*), `qdrant` (*2cs*), and `wasmtime` (*2cs*) – whereas Rceus scales to all 16 programs while also using less memory throughout.

Table 2 Comparison of RCEUS against *kcs*, SF-*kcs*, SEL-*kcs* ($k \in \{1, 2\}$) across time, memory, and precision metrics. "PA" denotes pre-analysis; "OOM" denotes out-of-memory.

Project ID	PA (s)	Metrics	Baselines						RCEUS
			1cs	SF -1cs	SEL -1cs	2cs	SF -2cs	SEL -2cs	
EP1	RTA: 0.40 SF-*kcs* adt: 0.02 RCEUS adt: 0.08	Time (s)	2.4	1.3	0.9	7.1	2.6	1.2	**0.8**
		Mem (GB)	0.4	0.3	0.3	1.4	0.5	0.4	**0.2**
		#dce	185	185	185	185	185	185	**184**
		#avg-pts	5.84	3.46	5.84	5.03	2.88	5.03	**2.37**
EP2	RTA: 0.73 SF-*kcs* adt: 0.06 RCEUS adt: 0.24	Time (s)	7.8	4.9	3.6	35.7	21.6	5.6	**2.7**
		Mem (GB)	1.6	1.3	1.1	5.6	4.5	1.8	**0.6**
		#dce	426	426	426	422	422	422	**420**
		#avg-pts	13.57	7.53	13.57	11.15	6.94	11.15	**3.69**
EP3	RTA: 0.97 SF-*kcs* adt: 0.09 RCEUS adt: 0.26	Time (s)	11.1	5.1	4.2	39.1	12.1	6.6	**3.2**
		Mem (GB)	2.3	1.1	1.2	8.8	3.4	2.0	**0.8**
		#dce	434	434	434	434	434	434	**434**
		#avg-pts	11.56	4.42	11.56	9.54	3.42	9.54	**2.44**
EP4	RTA: 1.21 SF-*kcs* adt: 0.12 RCEUS adt: 0.41	Time (s)	12.8	7.5	5.9	37.7	20.2	9.3	**4.9**
		Mem (GB)	3.6	1.5	2.1	11.7	8.5	3.5	**1.4**
		#dce	320	320	320	319	319	320	**319**
		#avg-pts	10.43	4.90	10.44	6.78	3.69	6.79	**3.13**
EP5	RTA: 1.18 SF-*kcs* adt: 0.12 RCEUS adt: 0.35	Time (s)	14.7	8.2	6.5	43.9	18.2	7.6	**3.6**
		Mem (GB)	3.2	2.2	1.9	8.7	6.8	2.3	**0.9**
		#dce	531	531	531	531	531	531	**506**
		#avg-pts	19.45	10.35	19.47	11.55	4.67	11.56	**2.72**
EP6	RTA: 1.38 SF-*kcs* adt: 0.16 RCEUS adt: 0.40	Time (s)	17.4	11.2	7.3	101.8	52.0	11.2	**5.4**
		Mem (GB)	4.0	3.2	2.1	18.3	9.8	3.5	**1.1**
		#dce	888	888	888	884	884	884	**876**
		#avg-pts	14.04	6.46	14.15	11.15	4.86	11.15	**2.87**
EP7	RTA: 1.62 SF-*kcs* adt: 0.19 RCEUS adt: 0.43	Time (s)	22.9	14.0	9.6	137.6	65.0	14.2	**6.4**
		Mem (GB)	5.3	4.7	2.9	22.5	18.1	5.0	**1.7**
		#dce	529	529	529	529	529	529	**527**
		#avg-pts	16.56	8.61	16.56	12.70	6.69	12.72	**3.35**
EP8	RTA: 0.92 SF-*kcs* adt: 0.10 RCEUS adt: 0.34	Time (s)	23.6	14.4	12.5	79.3	32.6	17.7	**4.5**
		Mem (GB)	7.0	6.5	4.1	24.7	20.2	6.8	**1.0**
		#dce	474	474	474	474	474	474	**474**
		#avg-pts	37.06	13.33	37.07	29.00	11.04	29.36	**2.19**
EP9	RTA: 1.62 SF-*kcs* adt: 0.15 RCEUS adt: 0.48	Time (s)	29.5	12.2	13.6	185.6	40.4	18.0	**6.3**
		Mem (GB)	8.2	3.2	4.6	25.8	10.7	5.1	**1.7**
		#dce	749	749	750	748	748	749	**678**
		#avg-pts	24.69	7.12	24.69	20.04	5.05	20.05	**3.06**
EP10	RTA: 4.08 SF-*kcs* adt: 0.38 RCEUS adt: 1.15	Time (s)	59.3	30.3	23.5	877.5	282.4	213.3	**15.8**
		Mem (GB)	12.0	10.1	6.5	129.8	93.0	38.2	**5.5**
		#dce	921	921	921	919	919	919	**899**
		#avg-pts	22.01	8.18	21.98	17.32	6.52	18.24	**3.36**
EP11	RTA: 2.11 SF-*kcs* adt: 0.27 RCEUS adt: 0.75	Time (s)	57.6	23.8	19.7	260.6	91.9	25.7	**8.4**
		Mem (GB)	18.8	9.9	10.0	51.9	36.2	11.8	**2.0**
		#dce	778	778	778	778	778	778	**774**
		#avg-pts	21.68	8.23	21.68	15.75	6.40	15.76	**3.01**
EP12	RTA: 3.17 SF-*kcs* adt: 0.53 RCEUS adt: 1.10	Time (s)	133.9	56.3	53.3	809.3	220.6	92.2	**16.8**
		Mem (GB)	41.8	23.6	16.3	102.7	73.5	25.7	**5.3**
		#dce	2006	2006	2006	2004	2004	2004	**1990**
		#avg-pts	52.09	16.31	52.11	46.53	14.44	46.53	**2.95**
EP13	RTA: 3.82 SF-*kcs* adt: 0.72 RCEUS adt: 1.25	Time (s)	156.7	57.5	66.2	713.3	202.8	78.2	**15.3**
		Mem (GB)	37.1	29.7	21.1	115.6	80.3	29.7	**4.7**
		#dce	1283	1283	1315	1211	1211	1244	**1135**
		#avg-pts	58.56	15.90	58.56	34.93	10.08	35.30	**3.66**
EP14	RTA: 7.79 SF-*kcs* adt: 1.69 RCEUS adt: 2.78	Time (s)	OOM	607.0	1178.7	OOM	OOM	OOM	**113.1**
		Mem (GB)		447.6	371.2				**83.3**
		#dce		2386	2411				**2832**
		#avg-pts		27.53	152.78				**4.32**
EP15	RTA: 9.18 SF-*kcs* adt: 3.34 RCEUS adt: 2.83	Time (s)	1358.4	216.7	424.5	OOM	OOM	421.2	**167.6**
		Mem (GB)	294.3	103.7	176.6			129.8	**61.0**
		#dce	1639	1639	1663			1650	**1618**
		#avg-pts	84.23	22.79	84.28			38.52	**4.86**
EP16	RTA: 16.01 SF-*kcs* adt: 2.24 RCEUS adt: 3.41	Time (s)	1809.9	607.6	599.4	OOM	OOM	OOM	**121.3**
		Mem (GB)	301.0	259.7	196.3				**101.7**
		#dce	2733	2733	2733				**2563**
		#avg-pts	120.29	75.13	120.36				**7.20**

For **precision**, RCEUS reduces `#avg-pts` by **86.6%** over *1cs* and **80.0%** over *2cs* on average across all benchmarks. The largest improvements occur on `grin` (94.3%, 52.09 → 2.95) and `qdrant` (94.2%, 84.23 → 4.86) vs. *1cs*, and on `grin` (93.7%, 46.53 → 2.95) and `resvg` (92.4%, 29.00 → 2.19) vs. *2cs*. These precision gains also eliminate many spurious call edges (`#dce`), most prominently in `wasmtime` (↓ 170 under *1cs*), `atuin` (↓ 148 under *1cs*; ↓ 76 under *2cs*), and `fselect` (↓ 71 under *1cs*; ↓ 70 under *2cs*).

Beyond precision, RCEUS also delivers substantial **efficiency** gains, averaging **4.9×** faster than *1cs* and **20.4×** faster than *2cs*. The largest speedups are 14.9× on `wasmtime` and 10.2× on `atuin` (vs. *1cs*), and 55.5× on `mdbook` and 48.0× on `grin` (vs. *2cs*).

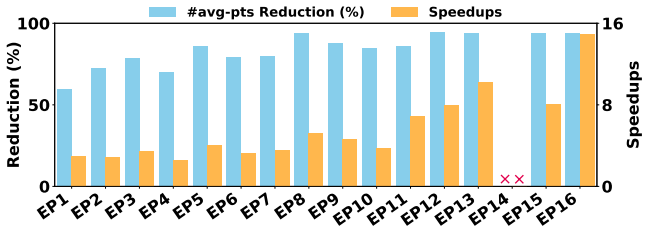

Figure 13 RCEUS vs. *1cs*: speedups and `#avg-pts` reductions (× indicates *1cs* is unsalable).

As program size increases, RCEUS yields progressively larger speedups while maintaining consistently high – and often increasing – reductions in `#avg-pts`. Detailed results appear in Table 2, with overall trends relative to *1cs* shown in Figure 13.

Finally, RCEUS lowers **memory usage** by **74.8%** relative to *1cs* and **93.1%** relative to *2cs*, on average, owing to the substantially reduced points-to sets it computes.

2) RCEUS vs. SF-*kcs*

SF-*kcs* [18] augments *kcs* with stack-object lifetime filtering, improving precision (`#avg-pts` reductions of 59.0% at $k=1$ and 59.3% at $k=2$) and efficiency (2.2× and 2.7× speedups). However, because raw pointers mediating heap objects do not carry lifetimes, SF-*kcs* cannot refine them. In addition, SF-*kcs* applies uniform k-limiting to all functions, causing unnecessary context construction and analysis. As a result, it exhausts memory on the largest benchmarks (`meilisearch`, `qdrant`, `wasmtime`) under *2cs*, although lifetime filtering allows it to scale to `meilisearch` at *1cs*. In contrast, RCEUS scales to all 16 benchmarks.

Beyond scalability, RCEUS outperforms SF-*kcs* in both precision and efficiency. At $k = 1$, RCEUS reduces `#avg-pts` by **68.9%**, runs **2.4**× faster, and uses **61.9%** less memory. At $k = 2$, it reduces `#avg-pts` by **50.8%**, runs **7.6**× faster, and requires **88.9%** less memory.

Overall, RCEUS delivers superior precision and efficiency to SF-*kcs* across all 16 benchmarks, achieving consistently smaller `#avg-pts` and faster analysis, with only a single deviation in `#dce` on `meilisearch`. This deviation, similar to Figure 12 (Section 3.3), arises when multiple trait objects invoke a method returning a reference to the receiver, and the returned references are used in subsequent calls. Because RCEUS preserves only the flow-entry callsite, these flows may be analyzed under the same context. While this avoids context blow-up, it can increase the number of dynamically resolved call edges. Even so, RCEUS still achieves substantially smaller `#avg-pts`, and adding one most-recent callsite (Section 4.3) eliminates this isolated conflation while preserving scalability.

3) RCEUS vs. Sel-*kcs*

SEL-*kcs* [25, 20] applies k-limiting only to precision-critical functions in *CritFuncs*, improving efficiency at the cost of modest precision loss. It scales on 14 of the 16 benchmarks (failing on `meilisearch` and `wasmtime` under *2cs*) and speeds up *kcs* by 2.5× at $k = 1$ and 6.9× at $k = 2$, though sometimes increasing `#avg-pts` or the number of resolved call edges (`#dce`). These results align with prior findings for Java, where the technique originated [25, 20].

In `lsd`, `fselect`, `atuin`, and `qdrant`, SEL-*kcs* produces slightly higher `#dce` than *kcs*. This is because SEL-*kcs*'s selection of precision-critical functions targets layered-abstraction flows rather than allocation-site precision; consequently, some allocation-related functions are classified as non-precision-critical, leading to minor heap-object conflation.

Table 3 Impact of integrating Rceus with stack filtering (SF), *kcs*, SF-*kcs*, and Sel-*kcs* (for $k \in \{1, 2\}$) across time, memory, and precision metrics. "OOM" marks out-of-memory runs.

Project ID	Metrics	Rceus	Rceus + Baselines						
			Rceus + SF	Rceus + 1cs	Rceus + SF-1cs	Rceus + SEL-1cs	Rceus + 2cs	Rceus + SF-2cs	Rceus + SEL-2cs
EP1	Time (s)	0.8	0.9	1.2	1.3	0.9	2.2	2.3	0.9
	Mem (GB)	0.2	0.2	0.3	0.3	0.2	0.6	0.6	0.2
	#dce	184	184	184	184	184	184	184	184
	#avg-pts	2.37	2.36	2.32	2.31	2.32	2.30	2.30	2.30
EP2	Time (s)	2.7	2.9	4.3	4.7	3.1	9.5	9.6	3.4
	Mem (GB)	0.6	0.7	1.0	1.0	0.7	2.5	2.5	0.7
	#dce	420	420	420	420	420	420	420	420
	#avg-pts	3.69	3.68	3.67	3.66	3.67	3.65	3.64	3.65
EP3	Time (s)	3.2	3.3	5.5	5.7	3.4	11.7	12.4	3.9
	Mem (GB)	0.8	0.9	1.4	1.5	0.9	3.9	4.0	0.9
	#dce	434	434	434	434	434	434	434	434
	#avg-pts	2.44	2.43	2.43	2.42	2.43	2.42	2.41	2.42
EP4	Time (s)	4.9	4.8	8.8	9.1	5.7	24.4	25.7	6.2
	Mem (GB)	1.4	1.2	2.4	2.5	1.4	7.5	7.5	1.5
	#dce	319	319	319	319	319	318	318	319
	#avg-pts	3.13	2.70	2.69	2.67	2.69	2.67	2.66	2.68
EP5	Time (s)	3.6	4.0	6.4	6.4	3.9	12.8	14.1	4.4
	Mem (GB)	0.9	1.0	1.6	1.8	1.0	3.9	4.0	1.0
	#dce	506	506	506	506	506	506	506	506
	#avg-pts	2.72	2.69	2.68	2.66	2.71	2.66	2.64	2.69
EP6	Time (s)	5.4	5.6	10.8	10.5	5.9	41.6	43.0	6.4
	Mem (GB)	1.1	1.3	2.5	2.7	1.2	9.3	9.3	1.2
	#dce	876	876	876	876	876	876	876	876
	#avg-pts	2.87	2.84	2.84	2.81	2.85	2.81	2.78	2.82
EP7	Time (s)	6.4	6.2	11.2	11.7	7.0	31.5	32.8	7.9
	Mem (GB)	1.7	1.9	3.3	3.5	1.8	10.6	10.9	2.0
	#dce	527	527	527	527	527	527	527	527
	#avg-pts	3.35	3.31	3.31	3.29	3.32	3.28	3.27	3.29
EP8	Time (s)	4.5	4.8	7.5	8.2	5.2	17.0	17.6	6.0
	Mem (GB)	1.0	1.1	1.9	2.0	1.1	6.2	6.3	1.2
	#dce	474	474	474	474	474	474	474	474
	#avg-pts	2.19	2.18	2.18	2.17	2.18	2.16	2.16	2.16
EP9	Time (s)	6.3	6.2	10.7	10.7	6.7	33.0	33.4	7.5
	Mem (GB)	1.7	1.7	3.5	3.3	2.0	11.5	10.3	2.2
	#dce	678	678	678	678	678	678	678	678
	#avg-pts	3.06	2.88	3.03	2.85	3.05	2.96	2.82	2.98
EP10	Time (s)	15.8	15.9	44.2	43.8	37.7	123.0	121.2	40.9
	Mem (GB)	5.5	5.2	14.7	13.6	7.8	50.2	49.9	9.2
	#dce	899	899	899	899	899	898	898	899
	#avg-pts	3.36	3.14	3.27	3.08	3.33	3.22	3.04	3.28
EP11	Time (s)	8.4	8.5	16.1	17.2	9.2	52.5	51.4	10.2
	Mem (GB)	2.0	2.4	5.0	5.5	2.2	22.8	19.0	2.4
	#dce	774	774	774	774	774	774	774	774
	#avg-pts	3.01	2.99	3.02	3.01	3.03	3.01	2.99	3.01
EP12	Time (s)	16.8	18.1	43.9	46.0	20.1	172.7	178.6	22.0
	Mem (GB)	5.3	5.8	17.0	17.3	6.8	53.1	53.1	6.9
	#dce	1990	1990	1990	1990	1990	1990	1990	1990
	#avg-pts	2.95	2.92	2.91	2.89	2.92	2.89	2.87	2.91
EP13	Time (s)	15.3	15.3	32.3	34.1	16.5	111.0	111.5	18.9
	Mem (GB)	4.7	4.9	11.8	11.8	5.2	40.7	38.8	5.9
	#dce	1135	1135	1107	1107	1135	1106	1106	1135
	#avg-pts	3.66	3.57	3.63	3.54	3.66	3.61	3.52	3.63
EP14	Time (s)	113.1	92.3	259.8	264.2	117.3	OOM	OOM	136.6
	Mem (GB)	83.3	48.5	139.1	140.6	64.2			76.4
	#dce	2832	2832	2216	2216	2241			2241
	#avg-pts	4.32	4.02	3.65	3.60	3.70			3.68
EP15	Time (s)	167.6	169.0	237.1	247.0	190.7	OOM	OOM	289.2
	Mem (GB)	61.0	61.8	136.9	138.7	73.8			138.2
	#dce	1618	1618	1592	1592	1618			1618
	#avg-pts	4.86	4.83	4.83	4.80	4.84			4.81
EP16	Time (s)	121.3	120.1	501.4	501.1	298.0	OOM	OOM	336.9
	Mem (GB)	101.7	104.8	382.6	381.7	277.4			309.3
	#dce	2563	2563	2557	2557	2557			2557
	#avg-pts	7.20	6.61	6.97	6.39	7.07			7.05

Compared with Sel-*kcs*, Rceus provides substantially higher precision and efficiency. At $k = 1$, it reduces `#avg-pts` by **87.9%**, runs **2.2×** faster, and uses **57.4%** less memory. At $k = 2$, it reduces `#avg-pts` by **80.7%**, runs **2.9×** faster, and consumes **71.3%** less memory.

4.3 RQ3: Extensibility

Table 3 shows that Rceus composes cleanly with existing context-sensitive techniques – including standard k-limiting [19, 9, 45, 35], selective k-limiting [22, 20, 25, 11, 36], and stack filtering [18] – thereby supporting a range of precision–performance tradeoffs and further demonstrating the benefit of preserving flow-entry callsites.

We evaluate four integrated configurations: Rceus + SF (stack filtering only) and Rceus + *kcs*, Rceus + SF-*kcs*, Rceus + Sel-*kcs* for $k \in \{1, 2\}$, where each baseline augments its context abstraction with the one selected by Rceus. Each integrated variant

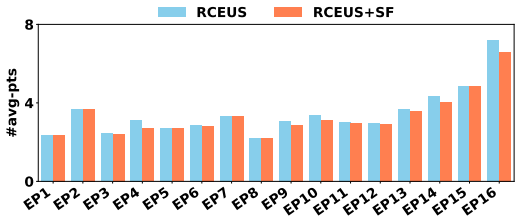

Figure 14 #avg-pts of RCEUS and RCEUS +SF across all benchmarks.

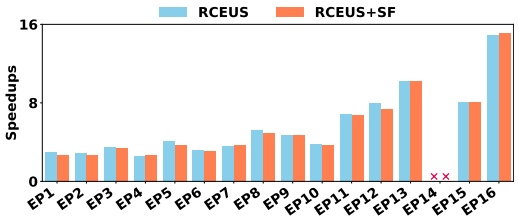

Figure 15 Speedups of RCEUS and RCEUS+ SF (normalized to *1cs*; unscalable marked ×).

achieves smaller `avg-pts` than RCEUS alone, either due to stack filtering (RCEUS +SF) or additional context sensitivity (RCEUS + *kcs*, RCEUS + SF-*kcs*, RCEUS + SEL-*kcs*), at higher analysis cost. Conversely, for every k-limiting baseline $\mathcal{A} \in \{kcs, \text{SF-}kcs, \text{SEL-}kcs\}$, RCEUS $+\mathcal{A}$ consistently yields lower `avg-pts` and, in almost all cases, faster analysis than running $\mathcal{A}$ alone, since RCEUS reduces the amount of propagation work each baseline must perform.

Overall, RCEUS is the fastest across nearly all benchmarks and baselines (with only six exceptions relative to RCEUS + SF), while incurring only slightly larger `avg-pts`. These results confirm that selectively preserving flow-entry callsites for precision-critical functions provides an effective and extensible mechanism for handling Rust's layered abstractions.

Let us now examine our results in more detail below.

- RCEUS **vs.** RCEUS**+SF.** Adding SF to RCEUS yields modest extra benefits: a further 3.3% drop in `#avg-pts` (Figure 14), with nearly identical analysis time across all benchmarks (Figure 15). This indicates that RCEUS already captures most effects of stack filtering: analyzing precision-critical functions under their flow-entry callsites removes the bulk of spurious heap and stack objects introduced by *kcs*. Still, SF provides a small, consistent precision lift at no additional cost, offsetting RCEUS's minor loss from analyzing non-*CritFuncs* functions context-insensitively.
- RCEUS **vs.** RCEUS $+\,\mathcal{A}$ **(**$\mathcal{A} \in \{$***kcs*, SF-*kcs*, Sel-*kcs***$\}$**).** RCEUS also functions as a drop-in enhancement for existing k-limited techniques, improving their scalability, efficiency, and precision. With assistance from RCEUS, both *1cs* and *SEL-2cs* scale to all the 16 benchmarks, whereas they fail to do so on their own (Table 2).
  Across nearly all benchmarks, adding RCEUS accelerates each baseline. At $k = 1$, integrating RCEUS yields average speedups of 2.5×, 1.2×, and 1.8× for *kcs*, SF-*kcs*, and SEL-*kcs*, respectively. At $k = 2$, the corresponding speedups are 4.0×, 1.4×, and 2.2×.
  Precision improves only modestly because RCEUS already resolves the dominant sources of imprecision in Rust's pointer analysis. At $k = 1$, *kcs*, SF-*kcs*, and SEL-*kcs* further reduce `#avg-pts` by 3.0%, 4.9%, and 2.5%; at $k = 2$, by 3.0%, 4.3%, and 3.2%.

4.4 RQ4: Impact of RCEUS's Flow-Entry Preservation

RCEUS scales reliably across programs of all sizes, typically outperforming the baselines in scalability, efficiency, and precision. As shown in Table 2, *kcs* is slowest and least precise, failing on the largest benchmarks (`meilisearch`, `qdrant`, `wasmtime`) within the memory budget. SEL-*kcs* speeds up *kcs* but is never more precise (provably as precise or less), while SF-*kcs* improves precision at the cost of slower performance.

For the largest benchmark (`wasmtime`, 669K LOC), *kcs*, SF-*kcs*, and SEL-*kcs* all fail at $k = 2$ due to OOM. Relative to *1cs*, RCEUS reduces `#avg-pts` by 94.0% (120.29 → 7.20), cuts memory by 66.2% (301.0GB → 101.7GB), and achieves a 14.9× speedup (1809.9s → 121.3s). By comparison, both SEL-*kcs* and SF-*kcs* achieve a 3.0× speedup: SEL-*kcs* slightly increases `#avg-pts`, whereas SF-*kcs* reduces it by 37.5%.

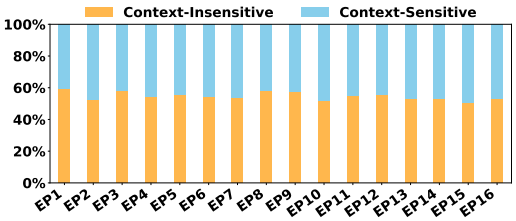

Figure 16 Distribution of context-sensitive and context-insensitive functions (RCEUS).

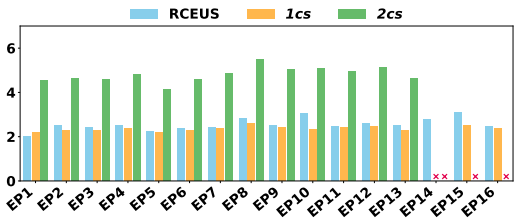

Figure 17 Average calling contexts per function for *1cs*, *2cs*, and RCEUS (×: unscalable).

In Section 4.2, we demonstrated – albeit indirectly – that preserving flow-entry callsites for precision-critical functions is highly effective for handling Rust's layered abstractions. We now examine this effect more directly and present an additional real-world case study.

Additional Scalability Analysis. Figure 16 shows the distribution of functions that RCEUS classifies as context-sensitive or context-insensitive. On average, only 45.3% of functions (those in *CritFuncs*) require context sensitivity, while the remaining 54.7% are analyzed context-insensitively. RCEUS therefore applies context sensitivity only to this precision-critical subset, preserving precision without sacrificing scalability.

Importantly, *CritFuncs* is distributed pervasively across all benchmarks, extending well beyond Rust standard-library functions. Among functions in *CritFuncs*, an average of 48.33% are standard-library functions, 23.45% are user-defined, and 33.41% are from third-party libraries. Manual inspection of the largest benchmark (`wasmtime`) further confirms that user-defined and third-party code frequently exhibits similar layered abstraction patterns, including compositional wrappers over standard containers, custom implementations of encapsulated unsafety, and trait-driven adapters layered on top of existing abstractions. These pervasive and heterogeneous patterns make manual modeling impractical, highlighting the importance of RCEUS's flow-guided identification of precision-critical functions for achieving both scalability and precision in real-world Rust programs.

Figure 17 compares per-function context counts for RCEUS, *1cs*, and *2cs*. RCEUS averages 2.53 contexts per function, close to *1cs* (2.35) and far below *2cs* (4.79). Despite analyzing all *CritFuncs* functions in separate contexts, RCEUS keeps context growth low, matching *1cs* while avoiding the explosion of *2cs*. Although RCEUS uses slightly more contexts per function than *1cs* on average, this selective precision prunes imprecise points-to information, reducing propagation cost and memory consumption during the main analysis.

Real-World Case Study. We examine a real-world program to illustrate how RCEUS prevents context explosion while preserving precision.

Rust's major application domains – from command-line utilities to network services – depend heavily on formatting and parsing built atop deeply layered trait implementations. Frameworks such as `Serde` use chains of trait-driven adapters (`Deserialize`, `Visitor`, `MapAccess`) that delegate across multiple layers and often recurse over input formats, creating substantial call-graph depth even for simple tasks. For example, deserialization in `zoxide` – one of our benchmarks – traverses a 31-function call chain that repeatedly manipulates the same heap-allocated data. This layered design makes large k unscalable, and parsing-heavy programs readily trigger combinatorial context blow-up.

Figure 18 shows a simplified example of regex parsing in Rust. The `parse()` function repeatedly invokes `check_values()`, which creates an iterator over the input strings and, on error, returns a reference derived from that input – a common pattern in real-world parsing

```rust
fn main() {
    let mut parser = Parser::new();
    parse(&parser);
}
fn parse(p: &Parser) -> Result<(), Error> {
    for flag in ["--a", "--b", "--c"] {
        match flag {
            "--a" => check_values(&p.vals)?,
            "--b" => check_values(&p.vals)?,
            "--c" => check_values(&p.vals)?,
            _ => (),
        };
    }
    Ok(())
}
```

```rust
fn check_values(vals: &Vec<String>) -> Result<(), Error>
    {
    for s in vals.iter() {   // it = vals.iter()
                             // s = it.next()
        if s.starts_with("!") {
            return Err(s);   // returns reference
    } } Ok(())
}
```

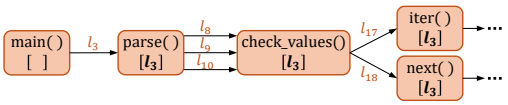

Figure 18 A Rust parsing example illustrating how RCEUS unifies repeated `check_values()` calls under one flow-entry context while preserving precise iterator flows.

code. In full programs, `parse()` may be invoked on different `Parser` objects from different flow-entry callsites; a context-insensitive analysis conflates them. Uniform k-limiting would require at least $k > 4$ to separate these pointer flows, but such repeated, recursive operations appear frequently in parsing code, making large k prone to combinatorial context blow-up.

RCEUS avoids this blow-up by preserving only flow-entry callsites. In this example, `parse()`, `check_values()`, `iter()`, and `next()` are precision-critical: when an error occurs, `check_values()` returns an argument-derived reference that `parse()` propagates. RCEUS identifies l_3 as their flow-entry callsite and propagates it to the three calls to `check_values()` at lines 8, 9, and 10, unifying these invocations under the same context. This prevents redundant exploration of the deep call chain (`iter()`, `next()`) while preserving precision.

5 Related Work

RCEUS builds on two bodies of work: Rust-oriented static analyses targeting MIR, and general techniques for context-sensitive pointer analysis developed largely for Java and C/C++. We summarize both to clarify how RCEUS differs from and advances the state of the art.

5.1 Static Analyses for Rust

Rust combines high-level memory safety with low-level control, but still permits unsafe code, including external libraries. Protecting safe Rust from such untrusted components – through memory isolation [24, 3] or address sanitization [30] – relies on precise points-to information. Some approaches use SVF [37], a C/C++-oriented framework on LLVM IR that requires manual compiler extensions to propagate MIR-specific constructs (such as raw pointers) into LLVM IR. Our evaluation confirms that k-limiting – also employed in SVF – is imprecise and inefficient for small k, and unscalable for larger k.

A growing body of verification [14, 28, 27, 6], security [4], and bug-detection work [23, 2, 5, 17] now targets MIR, which faithfully captures Rust-specific semantics. These analyses depend on precise points-to information and accurate call graphs, underscoring the need for MIR-level pointer analyses such as RCEUS. For Rust pointer analysis, still in its early stages, prior work includes RUPTA (*kcs*) [19] and its stack-filtering extension SF-*kcs* [18], which serve as baselines demonstrating RCEUS's advances over the state of the art.

5.2 Context-Sensitive Pointer Analysis Across Languages

Extensive research has explored different flavors of context sensitivity – callsite [31], object [29], and type sensitivity [35] for Java and C/C++ languages. All approaches rely on k-limiting: increasing k sharply raises cost but yields only modest precision improvements, so practical analyses typically restrict k to 1 or 2 [9, 45, 35]. Numerous techniques have sought to improve the efficiency-precision tradeoff of context-sensitive pointer analysis.

Selective Context Sensitivity. To improve the efficiency of k-limiting (especially in Java), selective context sensitivity applies context sensitivity only to methods that meaningfully affect precision, analyzing the rest context-insensitively [20, 22, 25, 13, 40]. The core challenge lies in identifying these precision-critical methods.

Early approaches rely on heuristics computed via pre-analysis. Smaragdakis et al. [36] proposed heuristics based on six manually designed metrics, using context-insensitive points-to information to estimate method criticality. Jeong et al. [13] introduced a machine-learning approach that assigns a context length to each method based on 25 atomic features, improving scalability but incurring substantial training cost. Scalar [21] estimates the amount of context-sensitive points-to information to choose an appropriate context-sensitivity variant.

To improve the precision of selective strategies, Zipper [20, 22] identifies three precision-loss patterns of value flows, while Eagle [25] applies partial context sensitivity to selected variables or allocation sites by reasoning about connected value-flow edges. Building on these ideas, we develop a Rust-specific pre-analysis that uniformly handles both stack and heap objects, avoiding the ad hoc code patterns required by earlier techniques.

Context Element Selection. Other approaches refine k-limiting by choosing which context elements to retain. Bean [41] removes redundant elements without reducing the number of distinct contexts, yielding higher precision than traditional k-limiting with only small overhead. Jeon et al. [10, 12] extend this line of work using machine-learned heuristics to select context elements in Java, enabling deeper and more precise contexts without increasing k.

In contrast to approaches that rely on language-specific heuristics or manually crafted rules, Rceus exploits Rust's underlying, language-wide design patterns. This generality allows Rceus to handle real-world Rust programs with deep layered abstractions effectively, while improving precision and scaling robustly to large codebases.

Other Approaches. Additional techniques address the efficiency – precision tradeoff from orthogonal angles. Wimmer et al. [48] improve scalability of type-based analysis by using saturation: once points-to sets exceed a threshold, they are replaced by an over-approximation, preventing further propagation. SkipFlow [16] enhances precision by pruning unreachable control-flow branches using interprocedural data-flow tracking. Ma et al. [26] improve precision by summarizing Java functions under three manually identified patterns, relying on Java-specific heuristics and API specifications. In contrast, Rceus's pre-analysis approximates *CritFuncs* and preserves soundness through its context-sensitive formulation.

6 Conclusion

Rust's ownership model and layered abstractions fundamentally reshape the demands placed on scalable and precise pointer analysis. We introduced Rceus, a Rust-oriented framework that constructs contexts by tracing interprocedural pointer flows and selectively preserving only the flow-entry callsites needed to distinguish parameter-derived flows in precision-critical functions. This flow-entry-preserving strategy eliminates the heap and stack conflation inherent in k-limiting while avoiding its exponential context blowup.

Across 16 real-world Rust applications – including `Wasmtime` – RCEUS delivers substantial improvements in scalability, efficiency, and precision over state-of-the-art techniques, sharply reducing points-to set sizes, memory usage, and analysis time. By controlling context growth and mitigating points-to explosion, RCEUS provides a practical, extensible foundation for future Rust analyses. Its precision and robustness enable more effective compiler optimizations, bug detection, and security analyses within Rust's expanding ecosystem.

References

1 David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/236337.236371.

2 Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 84–99, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483570.

3 Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bang.

4 Hung-Mao Chen, Z. Morley Mao, Yuan-Hong Wu, and Kuan-Yu Chen. TYPEPULSE: Detecting type confusion bugs in Rust programs. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/chen-hung-mao.

5 Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–21, 2023. doi:10.1145/3542948.

6 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. RefinedRust: A type system for high-assurance verification of Rust programs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656422.

7 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, USA, 2011. IEEE Computer Society. doi:10.1109/CGO.2011.5764696.

8 Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. A container-usage-pattern-based context debloating approach for object-sensitive pointer analysis. *Proc. ACM Program. Lang.*, 7(OOPSLA2), 2023. doi:10.1145/3622832.

9 Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222, pages 30:1–30:29, Dagstuhl, Germany, 2022. doi:10.4230/LIPIcs.ECOOP.2022.30.

10 Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276510.

11 Minseok Jeon, Myungho Lee, and Hakjoo Oh. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428247.

12 Minseok Jeon and Hakjoo Oh. Return of cfa: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498720.

13 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017. doi:10.1145/3133924.

14 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154.

15 Martin Kayondo, Inyoung Bang, Yeongjun Kwak, HyunGon Moon, and Yunheung Paek. MetaSafe: Compiling for protecting smart pointer metadata to ensure safe Rust integrity. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3711–3728, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/kayondo.

16 David Kozak, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. Skipflow: Improving the precision of points-to analysis using primitive values and predicate edges. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, pages 347–361, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3696443.3708932.

17 Wei Li, Wenyao Chen, and Jingling Xue. From raw pointers to memory safety: A modular demand-driven typestate analysis for rust. *Proc. ACM Program. Lang.*, 10(OOPSLA1), 2026. doi:10.1145/3798266.

18 Wei Li, Dongjie He, Wenguang Chen, and Jingling Xue. Stack Filtering: Elevating precision and efficiency in Rust pointer analysis. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, pages 331–346, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3696443.3708921.

19 Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. A context-sensitive pointer analysis framework for Rust and its application to call graph construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, pages 60–72, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3640537.3641574.

20 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276511.

21 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 129–140, 2018. doi:10.1145/3236024.3236041.

22 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Trans. Program. Lang. Syst.*, 42(2), May 2020. doi:10.1145/3381915.

23 Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 2183–2196, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484541.

24 Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe Rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 234–245, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380325.

25 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360574.

26 Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):539–564, 2023. doi:10.1145/3591242.

27 Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, PLDI 2022, pages 841–856, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523704.

28 Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for Rust programs. *ACM Trans. Program. Lang. Syst.*, 43(4):1–54, 2021. doi:10.1145/3462205.

29 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, 2002. doi:10.1145/566172.566174.

30 Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. ERASan: Efficient Rust address sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4053–4068, 2024. doi:10.1109/SP54263.2024.00258.

31 M Pnueli and Micha Sharir. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981.

32 Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020. doi:10.1145/3385412.3386036.

33 Olivier Sallenave and Roland Ducournau. Lightweight generics in embedded systems through static analysis. *ACM SIGPLAN Notices*, 47(5):11–20, 2012. doi:10.1145/2248418.2248421.

34 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 693–706, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192418.

35 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926385.1926390.

36 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594320.

37 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, pages 265–266, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892235.

38 Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 254–264, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2338965.2336784.

39 Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000. doi:10.1145/353171.353189.

40 Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485524.

41 Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510. Springer, 2016. doi:10.1007/978-3-662-53413-7_24.

42 The Rust Project Developers. The Rust programming language, trait objects, 2024. URL: https://doc.rust-lang.org/1.30.0/book/first-edition/trait-objects.html.

43 The Rust Project Developers. The Rust programming language, 2025. URL: https://www.rust-lang.org/.

44 The Rust Project Developers. The Rust reference, dispatch, 2025. URL: https://doc.rust-lang.org/beta/reference/glossary.html#dispatch.

45 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. *SIGPLAN Not.*, 52(6):263–277, June 2017. doi:10.1145/3140587.3062359.

46 Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, 2000. doi:10.1145/353171.353190.

47 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/996841.996859.

48 Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. Scaling type-based points-to analysis with saturation. *Proc. ACM Program. Lang.*, 8(PLDI):990–1013, June 2024. doi:10.1145/3656417.