CULPA: Universal Detection of Memory-Safety Bugs in Unsafe Rust Through the Lens of Safety Requirements

Hung-Mao Chen, Bo Lu, Xu He[†], Xiaokuan Zhang, Kun Sun
George Mason University
[†]*Visa Inc*

Abstract

Rust has emerged as a leading system programming language by providing strong compile-time guarantees for memory safety. However, these guarantees do not extend to unsafe Rust, where developers may bypass compiler checks and inadvertently introduce memory-safety vulnerabilities. Although prior static analyzers have made progress in detecting such bugs, existing approaches are often fragmented: they target specific coding patterns or bug classes without modeling the underlying causes of unsafety. In this paper, we present CULPA, a universal detector for memory-safety bugs in Rust programs. The key insight of CULPA is to detect the root cause of such bugs: the violation of safety requirements in unsafe Rust contexts. To do so, CULPA first transforms the safety requirements in the standard library documentation into machine-executable predicates. Then it constructs the distinct memory segments to comply with safety requirements. Finally, CULPA collects all safety-relevant safeguards to construct the Requirement Graph. Based on the requirement graph traversal, we can determine whether violations inside unsafe blocks can be triggered. CULPA covers existing bug classes addressed by four prior static analyzers and identifies additional memory-safety vulnerabilities beyond their scope. We evaluate CULPA on the top 1,000 Rust packages. CULPA uncovers 55 previously unknown (zero-day) memory-safety bugs, 29 of which have been confirmed by developers. Most of these vulnerabilities are missed by four state-of-the-art Rust static analyzers and one LLM-based tool. To date, we have received five RustSec IDs and one CVE ID.

1 Introduction

Memory safety vulnerabilities [16, 26, 29, 46, 53] in C and C++ have long plagued critical software systems, including the Linux kernel [24] and the Chrome browser [17], leading to severe security incidents and substantial maintenance costs [52]. Rust [47] has emerged as a promising solution to this problem, offering memory safety guarantees without sacrificing performance, thereby enabling the replacement of legacy C/C++ codebases in security-critical contexts [45]. The effectiveness of this transition is evidenced by empirical data: according to Android Developers, adopting Rust has resulted in a 1000-fold reduction in memory safety vulnerability density compared to equivalent C/C++ implementations [18].

Rust achieves its memory safety guarantees through a dual-language design comprising *safe Rust* and *unsafe Rust*. Safe Rust enforces strict ownership, borrowing, and lifetime rules at compile time, systematically preventing common memory errors such as use-after-free, double-free, and data races. However, certain low-level operations, including raw pointer manipulation, foreign function interface (FFI) calls, and direct hardware access, cannot be expressed within safe Rust's constraints without sacrificing performance or expressiveness. To support these essential use cases, Rust provides *unsafe Rust* [38], which allows developers to explicitly opt out of certain compile-time safety checks using the `unsafe` keyword. Within unsafe blocks, developers assume manual responsibility for upholding safety invariants that the compiler can no longer verify automatically. This design choice, while necessary for systems programming, introduces a critical vulnerability: violations of these manually enforced safety contracts constitute the primary source of memory safety bugs in Rust programs. Roughly 50% of Rust's Common Vulnerabilities and Exposures (CVEs) originate from violations of unsafe code safety requirements [33, 55].

Existing static analyzers [5, 6, 9, 23] for unsafe Rust detect memory safety violations through pre-defined bug patterns, such as Unsafe Dataflow Checker in Rudra [5] and type misalignment in TypePulse [6]. While effective within their targeted domains, these tools remain fundamentally limited: each focuses on a narrow class of violations tied to specific code idioms (e.g., unsafe trait implementations or type conversions), requiring distinct analysis strategies for different bug manifestations. Consequently, these analyzers detect only the *symptoms* of unsafe misuse, such as unresolved generic types or improper memory deallocation, rather than the underlying *root cause*, leaving a critical gap in Rust security analysis.

To identify the root causes of memory safety bugs in unsafe Rust, we systematically examined 64 memory-safety-related bugs from 39 RustSec advisories. Our analysis reveals that all 64 bugs stem from violations of safety requirements—the behavioral contracts explicitly documented for unsafe APIs in the Rust standard library (see Table 1). This finding establishes that safety requirement violations constitute the primary root cause of memory safety bugs in Rust programs. Motivated by this observation, we propose a fundamentally different approach: rather than detecting specific bug patterns or symptoms, we directly examine whether unsafe operations satisfy their documented safety contracts[1]. This contract-driven strategy enables a unified analysis framework capable of detecting diverse manifestations of unsafe misuse through a single detection pipeline. However, realizing such a static analyzer presents three fundamental challenges:

- **C1: Lack of Machine-Executable Safety Rules.** Safety requirements in Rust documentation are expressed in natural language and often encode multiple implicit properties, making them difficult to translate into static analysis predicates. Moreover, safety guarantees established by one unsafe operation may satisfy preconditions of subsequent operations, requiring cross-operation reasoning that cannot be captured by analyzing individual call sites in isolation.
- **C2: Invisible Boundaries within Memory Regions.** Safety requirements often apply to distinct segments within a single memory allocation, yet these boundaries are not directly observable in the program's control flow or type system. These boundaries evolve dynamically through raw pointer operations and safe API calls, but conventional data-flow analysis cannot track them because the metadata defining segment boundaries (e.g., `Vec::capacity`) is not explicitly passed as parameters to unsafe operations.
- **C3: False Alarms from Implicit Safeguards.** Data-flow reachability alone is insufficient to confirm violations, as implicit safeguards may prevent unsafe operations from manifesting as runtime errors. Type constructors, for instance, often establish initialization guarantees that satisfy preconditions at downstream unsafe sites, yet these constructors are typically omitted from standard call graphs. Furthermore, prior unsafe operations may invalidate these type-level guarantees.

To address these challenges, we design and implement CULPA, a static analysis framework for detecting safety requirement violations in Rust programs. CULPA consists of three core components: Specification Pool, Memory Segment Constructor, and Safeguard Analyzer. To resolve **C1**, we construct Specification Pool, which contains 228 compositional specifications that decompose compound safety requirements into fine-grained, machine-executable predicates. Each specification includes an emission mechanism whereby unsafe operations may produce guarantees that downstream operations can consume as evidence of satisfied preconditions, enabling cross-operation reasoning. To address **C2**, we design Memory Segment Constructor, a metadata-aware analyzer that represents memory boundaries using symbolic variables rather than concrete values. These symbolic variables are associated with inferred constraints (e.g., $len \leq capacity$ for `Vec`), which are checked whenever the variables are modified, enabling precise tracking of segment boundaries and constraints across program execution. Finally, to tackle **C3**, we design Safeguard Analyzer, which constructs a novel *Requirement Graph* to capture all implicit safeguards and type invariants associated with each memory region. The analyzer performs type dependency analysis to identify constructors that establish safeguards and tracks data-flow to detect prior unsafe operations that may invalidate these safeguards. A violation is reported only if no valid safeguard protects the memory region in the final Requirement Graph.

We implement CULPA with approximately 34,000 lines of Rust code and about 6,800 lines of YAML code (for the 228 specifications). We evaluate the coverage of CULPA on existing RustSec advisories and achieve 93.7%. Then, we evaluate the zero-day bug detection capability on top 1,000 ranked Rust packages from `crates.io`, it successfully detects 55 new bugs with 29 confirmation from developers, resulting in five RustSec IDs and one CVE ID. The performance of CULPA significantly outperform state-of-the-art static analysis tools [5, 6, 9, 23] and one additional LLM-based tool [22].

Contributions. This paper makes the following contributions:

- We systematically analyze 64 memory-safety bugs from RustSec advisories and demonstrate that safety contract violations constitute their primary root cause.
- We design and implement CULPA, a static analysis framework that detects memory-safety violations by checking adherence to documented safety requirements through compositional specifications, metadata-aware boundary tracking, and implicit safeguard reasoning.
- We apply CULPA to the top 1,000 most-downloaded Rust packages, discovering 55 previously unknown memory-safety bugs, 29 of which have been confirmed by developers, resulting in five RustSec advisories and one CVE.

2 Background

Safe vs. Unsafe Pointer in Rust. Rust programs manipulate memory-safety through two types of pointers. In safe Rust, the primary pointer forms are shared references (`&T`) and mutable references (`&mut T`), as well as smart pointers such as `Box<T>`, `Rc<T>`, and `Arc<T>`. These pointers are protected by the type system: lifetimes prevent dangling references, while the borrow checker enforces aliasing rules (e.g., at most one

[1] We use *safety contracts* and *safety requirements* interchangeably.

Table 1: We categorize 64 memory-safety bugs from 39 RustSec advisories in the past three years by types of safety requirement violations. **X (y)** means advisory X contains y bugs ($y > 1$).

Type	**2023**	**2024**	**2025**
InBound (27)	0055, 0059 0086 (2)	0377, 0379 0408	0003, 0005, 0032 0033 (2), 0050, 0053 0062, 0063, 0072 (10), 0106
Initialization (26)	0045, 0055 0075, 0078 0087	0001, 0359 0404, 0435	0049, 0105 0107, 0109 (2) 0108 (12)
Alignment (7)	0017, 0046 0047	0408, 0424 0426, 0431	—
NonNull (4)	—	0357, 0429 0442	0044

live `&mut T` to a location) and ensures that references cannot outlive the data they point to. In unsafe Rust, developers can use raw pointers (`*const T` and `*mut T`) and perform low-level operations such as pointer arithmetic, reinterpreting memory, and manually managing allocation and initialization. Raw pointers are not checked by the borrow checker and do not carry lifetime/aliasing guarantee, so they can violate assumptions of safe code (e.g., becoming dangling, misaligned, out of bounds, or pointing to uninitialized bytes).

Safety Requirements of Unsafe Rust. Although compiler checks can be bypassed in unsafe code, developers can still follow specific safety requirements to prevent undefined behaviors. For raw pointer dereference, the pointer is required to be properly aligned, non-null, initialized with valid value, and conform to aliasing rules (multiple mutable reference should not co-exist in the same lifetime) [50]. For calling unsafe functions, the Rust standard library usually maintains the safety requirements and semantics under the section of *Safety* for each API. However, the remaining unsafe behaviors do not admit compositional safety requirements. Mutating static variables depends on the program-wide synchronization context rather than the local conditions of individual accesses. Accessing the field of `union` type depends on the dynamic write history and semantic compatibility between the read and write invariants. Implementing unsafe traits encodes semantic invariants defined by the trait developer that must hold for all implementations across a single function boundary. These invariants lie outside Rust's enforceable semantics and cannot be reduced to concrete, checkable safety requirements. They represent explicit responsibility boundaries rather than enforceable safety contracts.

```
1 impl FromMdbValue for $t {
2     fn from_mdb_value(value: &MdbValue) -> $t {
3         unsafe { *transmute(value.get_ref()) }
4     }
5 }
6
7 #[inline]
8 pub fn new_from_sized<T>(data: &'a T) -> MdbValue<'a>
      {
9     unsafe { MdbValue::new(transmute(data),
          size_of::<T>())}
10 }
```

Listing 1: RustSec-2023-0047. `from_mdb_value` implemented on different types can violate multiple safety requirements of unsafe `transmute`.

3 Overview

3.1 Motivating Examples

In Table 1, we summarize 64 memory-safety bugs from 39 RustSec advisories and categorize all of them into the four kinds of violation on safety requirements: the pointer should access memory in the boundary, initialized, aligned, and non-null. To detect violations of these 64 memory-safety bugs, three challenges need be addressed, which we demonstrate using three existing RustSec bugs [11, 43, 44].

RustSec-2023-0047. In Listing 1, the function `from_mdb_value` is used to convert a reference of `MdbValue` into another type `$t` with unsafe `transmute` in line 3. Users can create an input for `from_mdb_value` with the `new_from_sized` function on line 8. However, memory-safety bugs can occur when we perform an arbitrary type conversion from generic type `T` to any type that implements `FromMdbValue`. Given that `FromMdbValue` is implemented on all primitive types (e.g., `u8`, `bool`), if we decide to use `u8` as `T` and type aligned to larger bytes (e.g., `u16`, `u32`), it will trigger misaligned pointer dereference at line 3. If the `u8` type has an arbitrary value and `transmute` to the `bool` type, it will trigger uninitialized memory access since Rust has strict bit patterns requirements on the value. The uninitialized memory access can also evolve to the out-of-bound memory access if we rely on the value of broken boolean type. Based on the safety documentation of `transmute`, these memory-safety bugs occur all because the safety requirement "the result must be valid at their given type" is violated [13].

RustSec-2025-0105. In Listing 2, the safe function `create_ring_buffer` first allocates a buffer using `Vec::with_capacity` on line 4. The function `Vec::with_capacity` is used to build a new, empty `Vec<T>` with at least the specified capacity to prevent frequent memory reallocation. However, the memory region within the size of the capacity is not initialized. Therefore, when the subsequent call to unsafe `Vec::set_len` tries to operate the length (line 5), which represents the size of initialized

```
1 pub fn create_ring_buffer<T: Copy>(size: usize) ->
      (Producer<T>, Consumer<T>) {
2     let buffer = Arc::new(DirectRingBuffer {
3         elements: UnsafeCell::new({
4             let mut vec =
                  Vec::<T>::with_capacity(size);
5             unsafe { vec.set_len(size) };
6             vec.into_boxed_slice()
7         }),
8         used: AtomicUsize::new(0),
9     });
10    // ..
11 }
```

Listing 2: RustSec-2025-0105. `create_ring_buffer` call unsafe `Vec::set_len` without checking the memory is initialized.

elements, it can create a `Box<[T]>` with uninitialized memory exposure (line 6). The uninitialized memory exposure triggered here is actually caused by the safety requirement of unsafe `set_len`, which requires the memory region between the old length and the new length to be initialized [41].

RustSec-2024-0431. In Listing 3, the safe function `as_slice` is used to interpret the memory object `MemoryRange` as a slice type. It is done using an unsafe function `slice::from_raw_parts` on line 16. Before calling `as_slice`, we must first use `new` on line 7 to construct `MemoryRange`. The constructor function `new` establishes the invariant that the memory layout is aligned to the page (typically 4096 bytes). Therefore, the type conversion behavior to arbitrary types `T` at line 17 will not cause misalignment pointer dereference as in RustSec-2023-0047. However, an illegal type can still be constructed when we use `bool` as type `T`. This bug also violates the safety requirement of `slice::from_raw_parts` where the type `T` should be properly initialized [37].

3.2 Challenges and Solutions

Based on motivating examples, we outline the three key challenges and their respective solutions.

C1: Lack of Machine-Executable Safety Rules. Rust's safety documentation describes unsafe operation contracts using high-level semantic requirements over memory, such as allocation validity, initialization coverage, and aliasing rules. However, these requirements are challenging to operationalize: (1) they are often implicit, where a single phrase like "valid at their given type" subsumes multiple distinct properties including allocation validity and full initialization (see RustSec-2023-0047), and (2) they span multiple operations, where guarantees established by one unsafe operation may satisfy requirements of a subsequent, seemingly unrelated operation. For example, `ptr::write` establishes initialization guarantees that satisfy the preconditions of a later `slice::from_raw_parts` call. Encoding these compound, cross-operation semantics as machine-executable rules is non-trivial.

```
1 pub struct MemoryRange {
2     pub(crate) addr: MemoryAddress,
3     pub(crate) size: MemorySize,
4 }
5
6 impl MemoryRange {
7     pub unsafe fn new(addr: usize, size: usize) ->
          Result<MemoryRange, Error> {
8         Ok(MemoryRange {
9             MemoryAddress::new(addr).ok_or(BadAddress)?,
10            MemorySize::new(size).ok_or(BadAddress)?,
11        })
12    }
13
14    pub fn as_slice<T>(&self) -> &[T] {
15        unsafe {
16            core::slice::from_raw_parts(
17                self.as_ptr() as *const T, self.len()
                      / core::mem::size_of::<T>())
18    // ..
19 }
```

Listing 3: RustSec-2024-0431. The type constructor function `new` provides the guarantee of alignment before calling unsafe function in `as_slice`. However, it can still break type validity with illegal bit pattern on same alignment.

Solution: We design a compositional specification that factors compound safety requirements into fine-grained, machine-executable predicates. Rather than directly encoding high-level requirements such as "valid at their given type", we decompose them into explicit, independently checkable and executable properties such as alignment, initialization, and allocation bounds. To capture multi-operation dependencies, we introduce an emission mechanism: each unsafe operation may emit guarantees that downstream unsafe operations can consume as evidence of satisfied preconditions. This design enables our static analyzer to verify requirement satisfaction across function boundaries by executing emission-requirement rules compositionally.

C2: Invisible Boundaries within Memory Regions. Safety requirements often apply to distinct segments within a single memory region, yet these boundaries are not directly observable in the program's control flow. For instance, a `Vec<T>` allocation comprises two memory segments delineated by its length and capacity: only memory within the length must be initialized, while capacity must remain greater than or equal to length before reallocation (see RustSec-2025-0105). These boundaries evolve dynamically through raw pointer operations and safe API calls, yet standard data-flow analysis cannot track them because the metadata defining these segments (e.g., capacity) typically resides outside the direct parameters of unsafe operations.

Solution: We develop a metadata-aware analyzer that represents these memory boundaries using symbolic variables rather than concrete integers. For example, at the construction site of a `Vec`, we assign symbolic identifiers `len(vec)` and `cap(vec)` to track length and capacity. Rather than performing symbolic arithmetic to track value changes,

we exploit invariants already maintained by safe APIs. When a safe function such as `Vec::reserve(n)` is invoked, we generate a fresh symbolic variable for `cap(vec)` and infer the constraint `cap(vec)` $\geq$ `len(vec)`. At a subsequent `Vec::set_len(m)` call site, we trace the provenance of parameter `m` to determine whether it maintains the required invariants related to these symbols. If the equality cannot be established, we conservatively mark the requirements as unverified, which needs to be further examined.

C3: False Alarms from Implicit Safeguards and Type Invariants. Even when a potential violation is reachable through data-flow analysis, it may constitute a false alarm if implicit safeguards prevent the violation from manifesting at runtime. Static analyzers face two fundamental challenges in accounting for such safeguards. First, safety protections may be established within type constructors (see RustSec-2024-0431), which are typically excluded from standard call graphs. Many type constructors establish initialization guarantees for `self` without invoking explicitly unsafe functions. Omitting these constructors causes the analysis to spuriously report initialization violations at downstream unsafe call sites. Second, safe type invariants may be invalidated and cannot be blindly trusted. Although Rust programs rely heavily on safe type construction to enforce invariants, these guarantees may be invalidated by prior unsafe operations. For example, consider a function `A` that accepts `&Vec<T>` and subsequently invokes `slice::from_raw_parts`. While `slice::from_raw_parts` requires the pointer to reference initialized memory, the initialization status of `&Vec<T>` cannot be assumed: a caller of `A` may have previously violated initialization requirements through unsafe operations such as `set_len`.

Solution: We construct a *Requirement Graph* that captures all safety-relevant safeguards associated with a specific memory region. To account for implicit safeguards embedded in type constructors, we perform type dependency analysis to identify functions within the same interface that return the `self` type. Furthermore, to handle the potential invalidation of safe type invariants, we track the data-flow of both memory allocations and prior unsafe operations, allowing us to prune safeguard that have been compromised. A violation is reported only if no valid safeguard protecting the memory region in the final Requirement Graph.

3.3 Scope

We aim to statically detect memory-safety bugs in Rust programs arising from safety requirement violations in unsafe APIs. Specifically, we identify whether `unsafe` code segments violate their documented safety requirements, then determine whether these violations are reachable from public APIs, thereby constituting security-relevant bugs.

We focus on memory-safety bugs including out-of-bound access, uninitialized memory access, misaligned memory access, null pointer dereference, and use-after-free. All of these

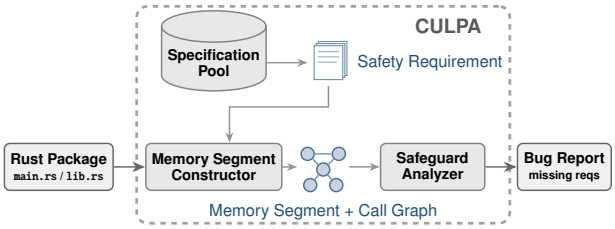

Fig. 1: Overview of CULPA's architecture.

bugs can be caused by violations of safety contracts specified in the Rust standard library documentation for raw pointer dereferences and unsafe APIs. To validate this scope, we analyzed 22 Rust packages related to 38 reported memory-safety bugs in 2025, extracting 643 unsafe operations: 85 raw pointer dereferences and 558 unsafe function calls. Of the 558 unsafe function calls, 468 directly invoke standard library functions, while the remaining 90 invoke third-party unsafe functions that transitively call standard library unsafe functions, thereby inheriting their safety contracts. This demonstrates that standard library safety contracts provide broad coverage for detecting memory-safety bugs across the Rust ecosystem. Issues related to Foreign Function Interface (FFI), logic errors, runtime panics are outside our scope. Although data races are categorized as undefined behaviors in Rust and some standard library safety requirements include concurrency-related preconditions, our analysis framework focuses on single-threaded control flow and does not model multi-thread interactions. Therefore, we exclude concurrency bugs from our scope.

4 Design

We design CULPA as a multi-layered analysis framework that systematically bridges the gap between high-level safety documentation and low-level code execution (see Fig. 1). While §3.2 described conceptual solutions to safety requirements violations, the architecture of CULPA operates these insights through a pipeline that transforms the Rust source code into a verifiable *Requirement Graph*. The design is centered on three major components: Specification Pool factors the compound safety requirements found in the official documentation into a set of machine-executable rules, providing the formal ground truth for what constitutes safe behavior in unsafe contexts. Memory Segment Constructor uses symbolic identifiers to handle the dynamic nature of distinct memory regions, which should meet the safety requirements. As the final stage of the pipeline, Safeguard Analyzer constructs a requirement graph that maps safety-relevant safeguards to their corresponding memory operations. By traversing this graph, it determines whether a memory-safety violation can be triggered by unsafe operations.

Table 2: This table summarizes safety requirements in five categories. Violations of the safety requirements will lead to memory-safety bugs. (None of the existing bugs in Table 1 violates **ExclusiveAlias**)

Requirement	Description	Count
InBound	Memory access should be in the boundary	27
NonNull	Pointer should be not null	82
Alignment	Pointer should be aligned	82
Initialization	Memory should be initialized	61
ExclusiveAlias	Pointer should follow alias discipline	22

4.1 Specification Pool

Specification Pool converts the implicit safety contracts of Rust's standard library into explicit machine-checkable predicates. To bridge the gap between human-readable documentation and automated analysis, the design of the specification pool must: (1) define a structured schema that can accurately represent the diversity of Rust's safety requirements, and (2) have a clearly defined scope of knowledge to be effective in real-world scenarios. We will elaborate on how we prioritize memory-safety-relevant modules in the standard library and how the system remains extensible to the broader third-party ecosystem. Together, these elements transform the theoretical need for rules into a robust, scalable configuration layer for memory-safety detection.

4.1.1 Specification Schema

Specifications are defined as a concise YAML schema, as shown in Listing 4. During analyzing the Rust package, CULPA will look up the specification and inject the declared safety contracts into the analyzer. The specification schema consists of three sections: `match`, `requirements`, and `emissions`. The `match` section identifies the target function using a path suffix pattern at MIR level so that it can match the same functions in different modules. `requirements` specify the conditions that must hold before invocation, and `emissions` describe the facts that become true after successful verification by CULPA. We will elaborate more details on `requirements` and `emissions`, which act as the core configuration to reason the safety contracts.

Requirements. The role of `requirements` is to ensure that the safety checks have been made on the unsafe code by callers. Listing 4 shows the specification of `Vec::set_len` where the unsafe function is used to manually change the length of the vector. Rust uses the concept of length to represent the memory region of the initialized memory and capacity to show the memory allocated for the vector. The caller of `Vec::set_len` must ensure that: first, the new length is less than or equal to the capacity; second, the memory range between the old length and the new length is properly initialized. [41]. In the specification, two rules will correspondingly execute `GuardLeq` and `InitCoveringRange` (see lines 5 and 7), both of which have the analysis pipeline in CULPA. The overview of requirement categories are elaborated in Table 2. Note that none of the 64 memory-safety bugs violate the alias discipline (e.g., The memory referenced by the returned slice must not be mutated for the duration of lifetime). To cover the requirements more comprehensively, we add the alias discipline as `ExclusiveAlias`. These safety requirements correspond to five common memory-safety bugs: uninitialized memory exposure (launched by `InitCoveringRange` above), null pointer or misaligned pointer dereference, out-of-bound read/write (launched by `GuardLeq` above), and use-after-free (launched by `ExclusiveAlias`). If multiple unsafe operations occur along a call chain, CULPA verifies the requirements at each point. When it encounters an unsafe operation whose requirements are not fulfilled, CULPA immediately stops analyzing the remaining code and reports the error. Any requirements that have already been confirmed will support the verification of subsequent unsafe operations, which should be done by emissions.

```
1 # Example: Vec::set_len
2 match:
3   path_suffix: "Vec::set_len"
4 requirements:
5   - kind: GuardLeq(VecLen(Receiver),
6           VecCap(Receiver))
7   - kind: InitCoveringRange(start=0,
8           len=VecLen(Receiver))
9 emissions:
10  target: Receiver
11  state:
12    - !set_len { value: !Arg 0 }
```

Listing 4: This example shows the specification schema of unsafe function `Vec::set_len`.

Emissions. The role of `emissions` is to propagate the state of the program state including verified requirements and safety semantics. It is included in the specifications of both safe and unsafe functions. In Listing 4, after the requirements of `Vec::set_len` are resolved, it will set the length of the `Vec` type to the function argument (line 12), which is necessary to verify the subsequent requirements on the unsafe operations of `Vec` type. In addition to the pointer metadata such as length and capacity, it also emits the semantics of type (e.g., `as_slice`) and mutability (e.g., `as_mut`). If the emission is used by unsafe function, it can help verify the following unsafe operations after its requirements are resolved. For example, it will also emit the relationship that the length of current `Vec` type is less than the capacity, and guarantee that the memory within size of length is initialized.

4.1.2 Specification Scope

The scope of our specification focuses on the standard library, which provides rich semantic abstractions. Based on

our observation mentioned in §3.3 where the unsafe functions in third-party libraries transitively call standard library unsafe functions, we will extend the specifications to third-party libraries to examine their calls to standard library unsafe functions.

Standard Library. CULPA currently provides behavioral specifications for 228 functions with following criteria: Among 330 public unsafe functions from Rust's standard library (excluding SIMD that is still experimental), we first exclude 203 functions that are not directly related to pointer or memory manipulation (e.g., `sync`, `thread`), leaving 127 unsafe functions. CULPA ensures that 100% (127/127) of documented unsafe pointer operations in the Rust standard library are enforced by the selective safety requirements (`NonNull`, `Alignment`, `Initialization`, `InBound`, `ExclusiveAlias`). Second, we add 101 additional safe functions that impact the preconditions of unsafe operations, including allocation origins (e.g., `Vec::new`, `Box::new`), capacity/length state changes (e.g., `Vec::with_capacity`, `Vec::set_len`), and pointer production (e.g., `Vec::as_ptr`, `Box::into_raw`), which must be tracked to verify that downstream unsafe operations satisfy their safety requirements.

In addition to the specifications of standard library functions, CULPA also maintains the specification of the raw pointer dereference, which is also `unsafe`. All of the requirements in Table 2 exclusive of *Bounds Checking* are applied to the raw pointer dereference. The reason is that out-of-bound access to the raw pointer will always be pre-analyzed when we validate the safety requirements of upstream unsafe functions such as `ptr::offset`. All other safety requirements should be satisfied in the raw pointer dereference to avoid undefined behaviors.

Third-party libraries. In addition to the specifications in the standard library, we also extend the specifications for the *custom* unsafe functions used in third-party libraries *given its source code*. Given a call-chain of `fn A`, `unsafe fn B`, `unsafe fn std::C`, we find that most of these custom unsafe functions (`B`) wrap the usages of unsafe functions (`C`) in the standard library or in raw pointer dereference inside. If we trace from each unsafe operation (`B` and `C`) to the caller context `A`, which means that the call-chain from `B` to `C` will cause duplicate analysis. To avoid duplicate analysis, we maintain a set of *Custom Unsafe wrappings* by recursively analyzing unsafe behaviors in their function bodies. The unsafe functions that belong to Custom Unsafe Wrappers inherit the safety requirements of their internal unsafe operations, which are already defined by the fixed specification. They are only used within the target crates for an ad-hoc analysis. CULPA demonstrates this capability to detect RustSec-2023-0086. Given the source code of the custom unsafe functions is not present, we will not maintain the specifications. We will discuss more details in §6.

Algorithm 1: Memory Segment Construction

Input: Function F, Specification Pool $\mathcal{S}$
Output: Memory segment annotations $\mathcal{L}$, sink sites Σ

```
// Phase 1: Identity Seeding
foreach pointer-type parameter p_i in F do
    create memory segment ℓ = (p_i, entire object);
    annotate p_i with ℓ in L;

foreach instruction s in F in control-flow order do
    // Phase 1: Identity Seeding
    if s is a heap allocation returning v then
        create ℓ = (alloc_site(s), entire object);
        annotate v with ℓ in L;
    // Phase 2: Propagation & Partition
    else if s is assignment v1 ← v2 then
        v1 shares the same memory segment as v2;
    else if s is container range access on c returning r then
        foreach (obj, _) annotated on c do
            annotate r with (obj, [len, cap)) in L;
    else if s is struct field projection b.f then
        foreach (obj, _) annotated on b do
            annotate b.f with (obj, field(f)) in L;
    else if s is unsafe call to f with pointer-type arguments
      a_1, ..., a_n then
        record sink site σ: map each a_j to its memory
          segments in L;
        Σ ← Σ ∪ {σ};
return L, Σ;
```

4.2 Memory Segment Constructor

Since safety requirements are applied to distinct segments within a single memory region, we define these segments as *memory segment*. Memory Segment Constructor extracts these segments from each function. However, extracting memory segments is non-trivial: both source code and traditional compiler IRs (e.g., LLVM-IR, MIR) represent memory through typed variables or raw addresses without exposing the specific sub-regions that safety requirements target. Given the Mid-level Intermediate Representation (MIR) [39], Memory Segment Constructor builds segments from pointer metadata and loads emissions from the Specification Pool to track metadata state changes. As shown in algorithm 1, the whole process can be separated into two phases: *Identity Seeding*, which creates initial memory segments for pointer-type parameters and heap allocations, and *Propagation & Partition*, which tracks how these segments evolve through assignments, field projections, and container operations along the control flow, ultimately recording all unsafe call sites where safety requirements must be checked.

Definition. Formally, memory segment is represented as a pair $(Object, Partition)$, where $Object$ represents the code place where the memory is allocated, and $Partition$ can refer to an

Table 3: Memory segment construction rules for common Rust memory patterns. Each memory segment includes a pair (Object, Partition) that enables range-aware and field-sensitive pointer analysis.

Construction Site	Object	Partition
Phase 1: Identity Seeding		
Function parameter e.g., `fn(a: *const i32)`	`ptr`	entire object
Heap allocation e.g., `Box::new(v)`	`Box`	entire object
Phase 2: Propagation & Partition		
Container type e.g., `Vec::with_capacity(n)`	`Vec`	(0, len) (len, cap)
Struct field e.g., `point.field`	`point`	`field`

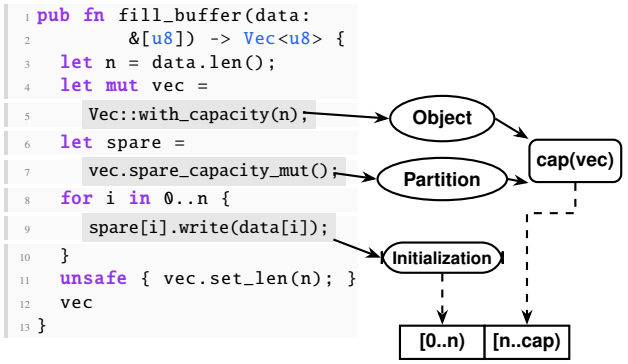

Fig. 2: Workflow of memory segment construction for a `Vec` object. The highlighted code lines construct Object/Partition to initialize the `cap(vec)` symbolic value and provide initialization guarantees with `write` operations.

entire allocated memory region or a structural region such as a field or specific range of memory, e.g., `[start, len)`. This allows CULPA to reason about entire objects and sub-regions systematically.

Phase 1: Identity Seeding. In this phase, we create an initial memory segment for each pointer-type function parameter and each heap allocation return site (e.g., `Box::new()`, `Vec::new()`) (see two rows of phase 1 in Table 3). The memory segment is labeled with the entire object since no sub-region information is available at construction time.

Phase 2: Propagation & Partition. In this phase, we traverse instructions in control-flow order to refine and propagate these segments. When an assignment $v_1 \leftarrow v_2$ is encountered, v_1 shares the same memory segment as v_2. Container range accesses (e.g., `Vec::with_capacity`) create range-based partitions such as $(obj, [len, cap))$. To handle the challenge of invisible memory boundaries, the Specification Pool is loaded during this phase and `emissions` will set pointer metadata as symbolic values rather than concrete integers, producing dynamic partitions such as $(0, len(vec))$ or $(len(vec), cap(vec))$. Struct field projections (e.g., `point.field`) create a partitioned segment $(obj, field(f))$, enabling field-sensitive analysis. Both of them can be found in phase 2 of Table 3. Finally, when an unsafe function is called with pointer-type arguments, we consider it as a *sink site* σ that maps each argument to its memory segments, collecting all such sites into the output set Σ. The construction workflow following algorithm 1 is based on a single function where each function could establish multiple memory segments. If any memory segments are pointer aliases across function boundaries, we will perform inter-procedural analysis to connect them in later component. The following example illustrates how we construct and refine memory segments in practice.

Example. In Fig. 2, the function `fill_buffer` constructs a new `Vec` with elements from `data`. Following the phase 1 in algorithm 1, the parameter `data` receives an initial memory segment. When `Vec::with_capacity(n)` on line 5 performs heap allocation, we create a new segment $\ell = (alloc\_site, \text{entire object})$ and annotates `vec` with ℓ; the `emissions` from specification sets the capacity to the symbolic value $cap(vec)$.

In the phase 2, the function call to `spare_capacity_mut` on line 7 creates the partition $(len(vec), cap(vec))$ and binds it to the new memory segment `spare`. On line 9, iterative `write` operations to `spare` provide initialization guarantees for this partition. When unsafe function `Vec::set_len` is called on line 11, we record the sink site σ, mapping `vec` to its memory segments. To resolve the requirement that the new length must not exceed the capacity, we analyze that `n` shares the same memory segment as $cap(vec)$ established by `Vec::with_capacity`, thereby satisfying the requirement.

The memory segment annotations L and sink sites Σ produced by algorithm 1 serve as direct inputs to the next component, Safeguard Analyzer: L provides the mapping from unsafe operation arguments to their memory segments, while Σ identifies the sites where safety requirements must be verified. Although the memory segment design maintains pointer aliases, it differs from traditional abstract memory locations (e.g., Andersen [54]) in two ways: (1) traditional algorithms trace only whole allocated objects, not sub-regions, and (2) they do not track dynamic metadata such as length or capacity. These capabilities make memory segments a distinctive contribution of CULPA.

4.3 Safeguard Analyzer

To detect the violations of safety requirements, we should not only identify the illegal operations but also the safety protection against the memory segment, which is the critical target constructed in prior phase (see §4.2). We define the concept of

safety protection as *Safeguard*. One common example of safeguard is bound check. If bound check is enforced to memory segment such as index, we can effectively confirm that the requirement of `InBound` is satisfied. Therefore, we will identify the safeguard before analyzing the violations of requirements. We categorize safeguard into the explicit and implicit ones: For the explicit safeguard, we summarize the code patterns that can satisfy any one of the safety requirements mentioned in Table 2. In contrast, implicit safeguard that is provided by type system (e.g., `Vec`) usually can satisfy multiple safety requirements at the same time. After identifying all safeguards, we construct a *Requirement Graph* for each unsafe operation by connecting relevant safeguards to the memory segments in single function. The final bug report will be generated when no applicable safeguards are found against the corresponding memory segment after traversing the requirement graph.

4.3.1 Explicit Safeguard

We summarize five types of explicit safeguards (Table 2) that correspond directly to the five safety requirements for memory-safety. Each explicit safeguard can be identified by pattern recognition such as code structures (e.g., if-condition checks) or concrete values (e.g., index, `Null`, content). Others, such as alignment constraints or type details of generic types, cannot be resolved via static analysis alone and should infer the potential type sets with trait-bound analyzer. There are also safeguard relying on temporal reasoning to determine whether memory has been freed at particular program points (e.g., ensuring memory is not accessed through another pointer after deallocation). After we identify explicit safeguards, we will also associate the safeguard to the relevant memory segment. We will describe how these five categories of safeguards are identified as the following.

InBound Safeguard. This type of safeguard is usually associated with memory segment of integer type as a function parameter. It can be established from the pattern of comparing the parameter to the constant value or the capacity of `Vec` type. We will trace the data-flow origin of memory segment. If it is created locally in the function, the `InBound` safeguard will be automatically identified for memory segment since its value cannot be controlled externally. If it is derived from the parameter, the memory segment will be temporarily marked as missing `InBound` safeguard.

NonNull Safeguard. This type of safeguard is associated to memory segment of pointer type. It can be established through an explicit null check (e.g., `!ptr.is_null()`) on the raw pointer level. In addition, we also monitors uses of `Null` (e.g. `ptr::null`) that can directly nullify the established safeguard.

Alignment Safeguard. This type of safeguard is associated with memory segment of pointer type. We will first collect the concrete values of aligned bytes based on the type information. The aligned bytes consist of a pair of values where we define them as *aligned-target* and *aligned-view*. Aligned-target is calculated based on the type used to interpret the pointer memory at the unsafe site. In other words, aligned-target represents the expected byte value of alignment to access the memory. Aligned-view is also calculated by type information; however, it is the actual value of alignment that can be changed through the type conversion or manual alignment. After tracking changes of alignment, both aligned-target and aligned-view will be sent to later component to decide whether the check can be established with the caller context. Since the calculation relies on the type information, We will maintain an *AlignedDomain* when encountering the generic type. When trait-bound is identified, we will collect all types that implement traits then insert their aligned-byte into the domain. If trait bound does not exist, AlignedDomain is default to include all possible values (1, 2, 4, 8, 16). When checking the requirements of larger aligned-bytes such as 8 bytes, it will consider `Alignment` safeguard as missing since `{1, 2, 4}` violates the requirement.

Initialization Safeguard. This type of safeguard is used to satisfy the requirement such that the memory region `[start, len)` must contain initialized values before access. Therefore, this safeguard is associated with memory segment of pointer types including the type with contiguous memory (e.g., `Vec`, slice). The initialization status in Rust depends not only on the memory write but also on the bit-pattern stored in the memory. Therefore, safeguard can be established in two steps: (1) explicit write event and (2) legal bit patterns. In the first step, we identifies the explicit write events (e.g., `ptr::write`, `vec![0u8; n]`) to the local variables or function parameters. Since safeguard can be revoked by reallocation or deallocation, which invalidates the previous initialization status, we will also record such kinds of behaviors.

In the second step to check the legal patterns (e.g., Boolean type can only be 0 or 1), we construct a pair of values, *validity-target* and *validity-view*, for memory segment. Validity-view is computed based on the type information that can be dynamically changed throughout the program. It can be established through two kinds of safe functions: the function that can directly construct legal value (e.g., `String::as_bytes`) and the one coming with error handling on illegal value (e.g., `str::from_utf8`). With both functions, memory segment of the parameter can also be marked as `UTF8Valid`. When analyzing the generic type, Safeguard Analyzer employs the *ContentDomain*, which models a lattice over validity states. After the same step of resolving trait-bound and collecting possible types, each type will be mapped to the validity requirement (e.g., `BoolValid`, `CharValid`, or `AllValid`). For `ContentDomain` without trait-bound, it uses the most strict one as the target of validity (`BoolValid`) to resolve the requirement (e.g., whether value is 0 or 1). However, the check can be revoked through the content mutation with invalid or unknown values.

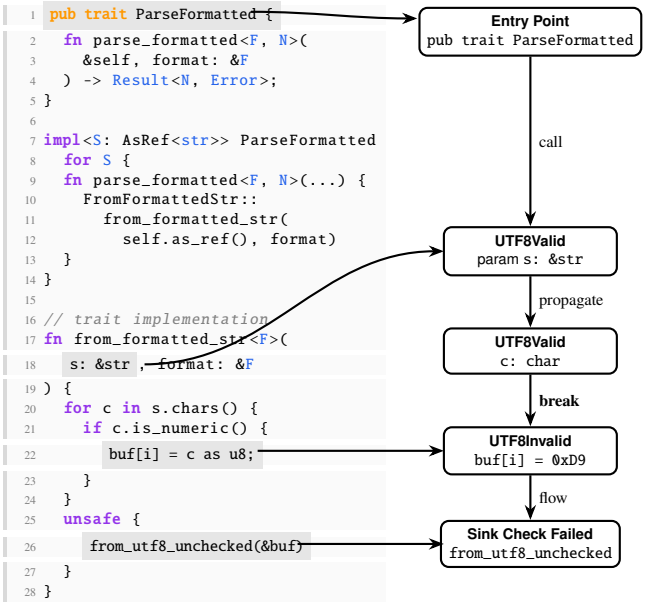

Fig. 3: The method `from_formatted_str` under the trait `ParseFormatted` demonstrates how implicit safeguard is established and revoked by casting operations. The rectangles on the right-hand side represent the identification process of entry point, implicit safeguards, and requirement violations.

ExclusiveAlias Safeguard. This type of safeguard follows the exclusive alias discipline, e.g., the memory can not be accessed with other pointer. In other words, missing `ExclusiveAlias` safeguard at specific unsafe code site can cause to temporal bugs such as use-after-free. `ExclusiveAlias` safeguard can be associated to pointer type and updated with both the *alias state* and the *alias escapability* whiling traversing the MIR. For alias state, we insert the pointer into the set of shared pointers alias whenever it encounters the creation of an immutable pointer. If a mutable pointer is created, it will be inserted into the set of unique pointer instead. When the lifetime of a reference ends (e.g., dropped), the pointer is correspondingly removed from its set. When both sets of shared pointer alias and unique pointer are not empty, it means that the alias state reaches a conflict and an illegal pointer alias exists in the same lifetime. However, not all pointer aliases will be kept in the set. With alias escapability, we will further investigate whether the pointer is derived from a parameter (including `self` in methods), a return value, or a global value. The alias escapability demonstrates whether the co-existing pointer alias is exposed to the external caller or accessed by the internal caller functions. We will keep only the pointer with the escapability in the set as valid safeguard.

4.3.2 Implicit Safeguard

In addition to explicit safeguards, we also identify the implicit safeguards provided by the safe types. The implicit safeguard can also satisfy the safety requirements of our five categories. The key difference to distinguish it from explicit safeguards is that each safe type usually provides multiple guarantee (e.g., reference can meet all of our safety requirements) rather than specifications and active behaviors. It is also more straightforward to associate implicit safeguard to the memory segment where the segment is always the safe types themselves. The safe types include references, `NonNull`, `Box`, `Vec`, `String`, slice, etc. Some safety types are also excluded in specific category (e.g., `MaybeUninit<T>` is used to keep uninitialized). However, it is critical to check the data-flow of the safe type construction since the unsafe operations may break the guarantees and disguise as a safe type. During analysis on the control-flow graph, we will mark the local variables or function parameter as the origin of safe type. When any pointer alias of memory segment is a raw pointer and participates in the unsafe operation, we will revoke the implicit safeguard provided by the safe type. If the origin is the safe type parameter, we can assign safeguard.

Example. In Fig. 3, we aim to resolve the safety requirements of legal bit patterns at the unsafe call site `from_utf8_unchecked` (line 26). The trait `ParseFormatted` as an externally reachable entry point since it is public, developers can implement the trait to access the internal method. In function `from_formatted_str`, the parameter `&str` provides an implicit safeguard `UTF8Valid` (line 18), preserved through `.chars()` (line 20). However, the `char as u8` cast (line 22) breaks the UTF-8 guarantee where multi-byte unicode characters are truncated to invalid single bytes, which revoke the previous safeguard to `UTF8Invalid`. Therefore, in line 26 of the sink `from_utf8_unchecked`, the safety requirement of initialization is violated.

4.3.3 Requirement Graph

Even though we can identify all explicit and implicit safeguards, it is still non-trial to identify whether requirements are violated or satisfied since safeguards may exist or even be invalidated across multiple function call chain. Therefore, we define a new data structure, Requirement Graph, which is used to collect all potential safeguards and track its changes throughout the control-flow and data-flow. Following algorithm 2, we first construct one requirement graph per memory segment at unsafe operation site in each single function. It consists of two steps: intra-procedural analysis from the single function that directly includes the unsafe operation and inter-procedural analysis that links all relevant information in the caller function to the current function. Finally, we traverse the constructed requirement graph from each entry point to the memory segment of unsafe operation to determine whether

Algorithm 2: Requirement Graph ($\mathcal{R}_u$) Construction & Traversal

Input: Call graph G, memory segment annotations $\mathcal{L}$, safeguards $\mathcal{SG}$, Specification Pool $\mathcal{S}$
Output: Set of bug reports $\mathcal{B}$

```
B ← ∅;
// Each unsafe operation site derives its own
// requirement graph on one memory segment.
foreach unsafe operation site u in G do
    look up requirements of u from S;
    resolve spec target (partition) to memory segment ℓ via L;
    initialize R_u with memory segment node ℓ;
    // Construction: Intra-procedural analysis
    F ← function containing u;
    foreach control-flow path from entry of F to u do
        collect safeguard nodes from SG(ℓ) that dominate u on this path;
        attach collected nodes to ℓ in R_u;
    // Construction: Inter-procedural analysis
    if F is not externally reachable then
        foreach caller C of F via BFS on G do
            map ℓ to corresponding parameter in C;
            extend R_u with safeguard nodes from C;
            extend R_u with type constructor safeguards via type dependency analysis;
            if C is externally reachable then break;
    // Traversal: Violation detection from entry point
    foreach path from entry point to ℓ in R_u do
        prune invalidated safeguard nodes;
        if path lacks any required safeguard then
            B ← B ∪ {(u, ℓ, path)} ;
return B;
```

safeguards present or invalid, hence verify the violations of safety requirements.

Construction: Intra-procedural Analysis. The analysis begins at each unsafe operation site within a single function. For each site, we look up its safety requirements from the Specification Pool and resolve which memory segment the requirements apply to, using the annotations produced by Memory Segment Constructor. We then build a requirement graph rooted at resolved memory segment: along every control-flow path from the function entry to the unsafe site, we identify which safeguards dominate the site and attach them as nodes in the graph. An edge from a safeguard node to the memory-segment node represents that the safeguard is intended to block the corresponding illegal operation to provide the guarantee. Note that safeguards node are collected from $\mathcal{SG}(\ell)$ in algorithm 2, which is the association between safeguards and memory segments in the identification phases (§4.3.1 and §4.3.2).

Construction: Inter-procedural Analysis. We also analyze whether the callers of current function, who directly performs the unsafe operation, provide the safeguards. If the current function is not externally reachable, we trace upward through the call graph via breadth-first search, extending the requirement graph at each caller. For each caller function, the extension involves three operations: (i) we map the memory segment in the callee function to the corresponding parameter in the caller so that safeguards in different functions can be linked to the same memory region; (ii) we attach the caller's own safeguards to the current graph; and (iii) we incorporate type constructor safeguards that are typically absent from a standard call graph. Type constructors (e.g., `self::new`) establish invariants such as proper initialization before any method is invoked, but they are always not direct callers of the unsafe site. We identify them via type dependency analysis, which associates a type with the function in the same interface that returns the `self` type. This upward traversal continues until either all requirements are satisfied or an externally reachable entry point is reached.

Traversal: Violation Detection. Once the requirement graph is fully constructed from the unsafe site up to a public entry point, we perform a final traversal to determine whether any violation path exists. The key insight of this step is that not all safeguards in the graph remain valid: prior unsafe operations along the same path may have compromised the type invariants that a safeguard relies on. We therefore prune safeguard nodes whose guarantees have been invalidated, based on their data-flow provenance. After pruning, if any path from the entry point to the memory segment still lacks a required safeguard, that path constitutes a *bypass*: a concrete route through which an attacker can trigger undefined behavior via a safe public API. Since multiple distinct call-chains may reach the same memory segment, we will report a bug if a violation appears on any single path.

Based on requirement graph, CULPA will automatically generate a bug report containing detailed diagnostic data. It presents the findings using `rustc`'s error format and outputs the full function call chains that an attacker could leverage to trigger undefined behavior. This information helps users confirm and understand the reported bug.

5 Evaluation

Implementation. CULPA comprises approximately 34,000 lines of Rust code and integrates with `rustc`'s nightly toolchain (2025-02-20) to extract MIR for analysis. Specification Pool consists of 228 YAML files totaling 6,800 lines, initially generated using a large language model over one week and subsequently refined manually over one month. Specifications are loaded lazily at runtime via `serde_yaml` deserialization. The database indexes specifications by unique path suffixes, enabling $O(1)$ lookup and automatic matching of

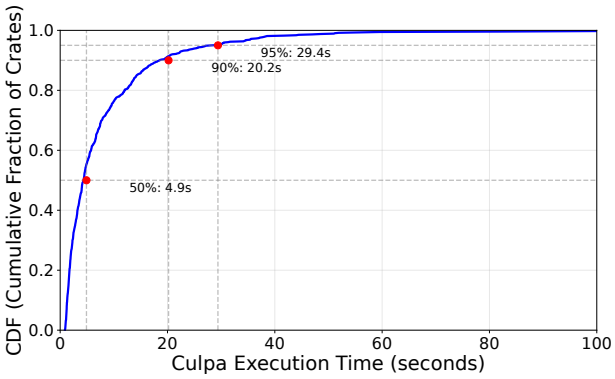

Fig. 4: Runtime performance of CULPA on the top 1,000 packages.

duplicate specifications across modules (e.g., `std::vec::Vec` and `alloc::vec::Vec`). Memory Segment Constructor maintains aliasing relationships using a union-find data structure with path compression, providing amortized constant-time lookups for aliased memory locations across function boundaries. Safeguard Analyzer performs inter-procedural analysis via breadth-first traversal from each unsafe operation site to its transitive callers, using a visited set to prevent infinite recursion in cyclic call graphs.

Dataset Collection. We evaluate CULPA on the top 1,000 Rust packages from `crates.io`, ranked by download count and updated within three months prior to December 1, 2025. This selection prioritizes widely-used packages, as download counts reflect transitive dependencies through `cargo add`, `cargo build`, and `cargo install` operations, thereby capturing packages with significant ecosystem impact.

Experiment Setup. We conduct all experiments on a server equipped with a 48-core Intel Xeon E5-2630 CPU and 256 GB of memory, running Ubuntu 22.04 with the `rustc` nightly toolchain (2025-02-20). We impose a 10-minute timeout per package and analyze all 1,000 packages using 16 parallel processes.

5.1 Bug Detection Results

We analyze 1,000 packages using 16 parallel processes, completing the scan in 18 minutes. Of these, 270 (27%) packages fail before entering the analysis pipeline due to compilation errors: 113 encounter internal `rustc` compiler errors (e.g., panic); 92 in crates that only define macros rather than using them, so the code will not be present in MIR to be analyzed, 31 have unresolved dependency conflicts; 27 lack analyzable targets (`lib` or `bin`); and 7 fail for other reasons. To minimize the failed packages, we have tried switching the version of `rustc` toolchain (nightly-2025-05-10 based on error messages) and re-analyze the failed packages to see whether the original `rustc` compiler errors and dependency conflicts can be resolved. The results show that both issues remain and they cannot be addressed by simply changing the toolchain version. We introduce the most common example of dependency conflicts in Appendix A. For the remaining 730 (73%) packages that CULPA successfully analyzes, the mean analysis time is 9.0 seconds per package. As shown in Fig. 4, 50% of packages are completed in 4.9 seconds, 90% in 20.2 seconds, 95% in 29.4 seconds and 99% in 60 seconds.

Table 4: Zero-days bugs detected by CULPA, sorted by the download rate by December 1, 2025. Note that the download rate in this table is calculated within recent 3 months rather than total numbers. Bug types include **UAF**: use-after-free, **Uninit**: uninitialized memory access, **NPD**: null pointer dereference, **MAG**: misaligned access, **OOB**: out-of-bound access. The bug report status includes **R**: reported; **C**: confirmed; **P**: Patched.

Package	Version	Download	Bug	Status
allocator-api2	0.3.1	44.2M	UAF (2)	R
data-encoding	2.9.0	34.9M	Uninit (5)	P
rkyv	0.8.12	16.0M	NPD (4)	P
git2	0.20.3	10.6M	NPD (1)	P
aligned-vec	0.6.4	9.6M	MAG (7)	R
ascii	1.1.0	7.9M	Uninit (3) OOB (1)	P
num-format	0.4.4	7.4M	Uninit (2) OOB (1)	R
arrow-arith	57.1.0	6.6M	OOB (1)	P
arraydeque	0.5.1	6.4M	Uninit (2)	R
gix-date	0.11.0	5.4M	Uninit (1)	P
quick-protobuf	0.8.1	3.9M	MAG (1) OOB (1)	R
inlinable_string	0.1.15	3.0M	Uninit (3)	C
capnp	0.23.0	934K	OOB (11)	P (4) R (7)
json	0.12.4	920K	Uninit (2)	R
parquet	57.1.0	883K	MAG (3)	C
allo-isolate	0.1.27	647K	NPD (1)	R
extfmt	0.1.1	642K	Uninit (1)	P
rust-crypto	0.2.36	184K	OOB (2)	C

CULPA reports 107 potential zero-day bugs across the evaluated packages. Upon manual inspection, 55 are validated as true positives, 23 are determined to be false positives, and 29 correspond to out-of-scope FFI cases. This yields an overall precision of 70.5% (55 / (55 + 23)). Table 4 lists packages containing confirmed bugs or those assigned RustSec advisories and CVE identifiers. Notably, 35 of the 55 confirmed bugs occur in packages with over 3.0M downloads from `crates.io`, including widely-used libraries such as `git2` [8] (maintained by the Rust-Language team) and `arrow-arith` [48] (maintained by the Apache Software Foundation). Table 5 includes

Table 5: Bug detection results of CULPA on the top 1,000 crates, grouped by the safety-requirement violation and its corresponding memory-safety bug type.

Violation	Bug Type	TP	FP	Precision
InBound	Out-of-Bounds	17	4	80.9%
NonNull	Null-Pointer Dereference	6	3	66.7%
Alignment	Misaligned Access	11	2	84.6%
Initialization	Uninitalized Access	19	14	57.6%
ExclusiveAlias	Use-After-Free	2	0	100%
Overall		**55**	**23**	**70.5%**

Table 6: CULPA achieves superior performance on the dataset of 119 memory-safety bugs (55 zero-day bugs + 64 existing RustSec bugs), compared to the state-of-the-art bug detectors.

	Zero-day (55)	RustSec (64)	Total (119)
MirChecker	5	4	9
Rudra	9	3	12
TypePulse	8	22	30
SafeDrop	4	1	5
Safe4U	18	35	53
CULPA	**55**	**60**	**115**

all bugs reported by CULPA. Each violation of safety requirement correspond to one bug type. The results show that CULPA can detect five types of memory-safety bugs: out-of-bound, null pointer dereference, misaligned memory access, uninitialized memory access, and use-after-free. To facilitate bug resolution, we provide reports and Proof-of-Concept (PoC) exploits for reported issues based on CULPA's diagnostic output. At the time of writing, we have documented all 55 confirmed bugs, of which 29 have been acknowledged or patched by maintainers. We have submitted these findings to the RustSec advisory database and CVE program, receiving five RustSec IDs and one CVE ID to date, with additional submissions under review. The bug disclosure timeline is long and usually takes several months due to RustSec's official requirements, including (i) prior consent from the package maintainer, (ii) review by RustSec advisors, (iii) advisory filing, which is typically deferred after a patch is released.

Coverage on Existing Bugs. We evaluate CULPA on the RustSec dataset of existing unsafe-related bugs as of December 1, 2025 (see Table 1), achieving a coverage rate of 93.7% (60 of 64 bugs). Among the 4 bugs that are not reported by CULPA, 2 bugs (RustSec-2023-0046, RustSec-2023-0087) are reported in the crates that define the functions. 1 bug (RustSec-2024-0424) is related to the unsafe function that is in experimental SIMD module. 1 bug (RustSec-2023-0075) is only triggered in specific platform. To support detecting these 4 bugs, CULPA should track reverse dependencies to find downstream crates that use these malicious macros, expand new specification *after* the Rust official stabilizes the SIMD functions, and integrate the analysis of `rustflags` into the detection logic, which all require non-trivial efforts. However, current coverage rate of CULPA already demonstrates that we can analyze most of the real-world bugs.

Comparison with Existing Tools. We compare CULPA against four state-of-the-art static analyzers (MirChecker [23], Rudra [5], TypePulse [6], SafeDrop [9]) and one LLM-based tool (Safe4U [22] with GPT-4o, which is its best model) for unsafe Rust bugs on a combined dataset of 119 bugs: 64 existing memory-safety vulnerabilities from RustSec advisories and 55 confirmed zero-day bugs discovered by CULPA. As shown in Table 6, CULPA detects 115 bugs (96.6%), substantially outperforming the detection capability of all five tools. Among them, Safe4U achieves the closest performance with 53 detections; however, it still misses 62 bugs that CULPA identifies. Among 53 results, there are 4 misaligned access, 6 null-pointer dereference, 26 out-of-bound access, and 17 uninitialized access bugs, but do not include any use-after-free bugs. Moreover, Safe4U may have potential data contamination issues as the GPT-4o may have already seen the publicly disclosed RustSec bugs. CULPA covers all bugs that can be detected by existing tools. This performance gap reflects the CULPA's effectiveness of unsafe code analysis with the most comprehensive coverage of bug types. We will explain the technical differences in depth between their analysis approaches in Appendix B.

5.2 False Positive Analysis

CULPA reports 23 false positives among 78 total warnings (29.5%), as summarized in Table 5. We systematically categorize these false positives by root cause to identify the causes of false positives. The distribution is as follows: 11 cases (47.8%) arise from missing domain-specific validity invariants, 5 cases (21.7%) from untracked correlated field constraints, 5 cases (21.7%) from insufficient arithmetic guard propagation, and 2 cases (8.7%) from external format guarantees. Notably, `Initialization` violations exhibit a higher false positive rate due to CULPA's limited ability to verify content-dependent validity properties beyond structural initialization. We will discuss the false positive case as following.

Case Study. Listing 5 illustrates the primary challenge in eliminating false positives: domain-specific validity invariants, which account for 47.8% of all false positives. These invariants arise from arithmetic properties, encoding specifications, or implicit validation embedded in parsing logic. In this example, CULPA reports an `Initialization` violation at line 9, where `str::from_utf8_unchecked` is invoked. However, the code is safe: the arithmetic at line 5 ensures UTF-8 validity because `value % 10` produces values in $[0,9]$,

```rust
fn write_integer(mut value: u32, buffer: &mut [u8; 10]) -> &str {
    let mut index = buffer.len();
    loop {
        index -= 1;
        buffer[index] = b'0' + (value % 10) as u8;
        value /= 10;
        if value == 0 { break; }
    }
    unsafe {
        str::from_utf8_unchecked(&buffer[index..])
    }
}
```

Listing 5: The function `write_integer` guarantees the UTF8-validity by modular arithmetic before calling unsafe `str::from_utf8_unchecked` to create string type.

and `b'0' + [0,9]` yields bytes in `[0x30, 0x39]`, which correspond to ASCII digits that constitute valid single-byte UTF-8 sequences. Verifying this property requires *semantic domain knowledge* spanning three distinct domains: (1) modular arithmetic (`value % 10` $\in [0,9]$ for unsigned *value*), (2) byte-level addition (`b'0' + n` where $n \in [0,9]$ yields `[0x30, 0x39]`), and (3) the UTF-8 encoding specification (bytes in `[0x00, 0x7F]` are valid single-byte sequences). Though we can track that `value % 10` $\in [0,9]$, connecting this fact to UTF-8 validity requires embedding encoding-specific axioms.

One practical way to further reduce false positive is to mark buffer that is called with function `str::from_utf8_unchecked` and filled with `byte_literal + (value % k)` to be safe. However, such rule sacrifices soundness even with generalization: given the same pattern `b'A' + (value% 100)` produces bytes in `[0x41, 0xA4]`, values above `0x7F` are invalid UTF-8. The true bug can be silently suppressed. Therefore, it is impractical for general-purpose static analyzers to cover the infinite combinations of encoding arithmetic without either encoding unbounded domain axioms or sacrificing soundness.

5.3 Case Study

In this section, we present a zero-day vulnerability discovered by CULPA in the `arrow-arith` package [49], which is part of the Apache Arrow [1]. This package serves as the fundamental memory format for modern data engineering, supporting high-performance analytical engines such as DataFusion [2] and various cloud-native data platforms. Its Rust implementation [48] has accumulated over 3.3k stars with GitHub repositories and lists 557 dependents on `crates.io`. In Listing 6, CULPA detects a potential violation of `InBound` requirements on `slice::get_unchecked_mut` at line 17. When CULPA analyzes whether `idx` can be larger than the boundary of `buffer`, it will trace back to line 13 to identify the range of possible values in `nulls.valid_indices`, which is decided by the

```rust
pub fn try_binary<A, B, F, O>( a: A, b: B, op: F)
    -> Result<PrimitiveArray<O>, ArrowError>
where
    A: ArrayAccessor, B: ArrayAccessor, O:
        ArrowPrimitiveType,
    F: FnMut(A::Item, B::Item) -> Result<O::Native,
        ArrowError> {
    // ...
    let nulls =
        NullBuffer::union(a.logical_nulls().as_ref(),
        b.logical_nulls().as_ref());
    // Buffer allocation based on Array::len()
    let len = a.len();
    let mut buffer =
        BufferBuilder::<O::Native>::new(len);
    // ...
    // Loop iteration based on logical_nulls().len()
    for idx in nulls.valid_indices() {
        unsafe {
            // OOB write when nulls.len() > len
            *buffer.as_slice_mut()
            .get_unchecked_mut(idx) = op(...)?;
        }
    }
}
// Attacker's PoC
impl Array for Evil { fn len(&self) -> usize { 1 } }
```

Listing 6: Unchecked OOB write: The `try_binary` function assumes that the buffer length is equal to logcial_nulls, permitting unchecked OOB memory writes via `get_unchecked_mut`.

length of `logical_nulls` on line 9. While the value of `idx` is determined by the length of `logical_nulls`, the actual boundary of `buffer` is obtained from the length of type that implements `Array` trait (see line 4). This shows that the requirements of `get_unchecked_mut` might be violated by the inconsistency between the lengths of `buffer` and `logical_nulls`. After CULPA identifies this inconsistency, it continues to analyze whether `len` or `logical_nulls` is a method under public trait, which means that external attackers can overwrite the implementations at will. We draft a PoC (line 22) to overwrite the implementation of `len`, which will shrink allocation sizes while `logical_nulls` retains the original size. Our PoC triggers out-of-bound write at line 17 and causes unexpected runtime panic to crash the program.

In the context of the downstream package, DataFusion assumes that the type metadata provided by `arrow-arith` accurately reflects the underlying memory layout of each data chunk. The bug in `arrow-arith` violates this assumption: when an attacker-controlled `Array` implementation returns an inconsistent length, the engine performs out-of-bounds writes that silently corrupt memory without triggering immediate crashes, propagating invalid data across distributed computations. After we reported this bug to the Apache community, it was confirmed and subsequently patched. The discussion highlighted a common dilemma in Rust's unsafe encapsulation. Initially, developers suggested sealing the trait (i.e., making it `private`) to prevent external error-prone implementations. However, downstream developers revealed that they rely on implementing the `Array` trait for custom data types.

Consequently, the maintainers implemented runtime assertions to ensure length consistency before entering the `unsafe` block.

6 Limitations

Manual Specification Construction and Maintenance. The manual effort of creating 228 specifications includes roughly 20 hours of initial generation and 100 hours of refinement against existing bug dataset. For a new function, it takes around 30 minutes to create its specification. The effort of maintaining specifications is minimal since the function signatures of public APIs in the standard library will not be changed within any major version (Rust-RFC 1105 [51]). Although LLMs can assist in initial specification generation, iterative manual refinement is necessary to ensure both completeness and soundness, particularly when the official documentation is ambiguous or underspecified. Future work could explore fine-tuning domain-specific models to automate this process, though validation remains challenging without ground truth data.

Standard Library Focus. We deliberately scope our analysis to raw pointer dereferences and unsafe APIs in the standard library, which form the foundation of nearly all unsafe operations in Rust programs (see §3). This design choice is justified by two factors: first, the standard library provides broad coverage, as third-party unsafe functions typically delegate to standard library primitives. Second, the standard library is stable across Rust versions, making specification construction a one-time investment. Given the source code of third-party libraries are not included, extending specifications would require the cross-package analysis and continuous specification maintenance due to frequent API changes, significantly increasing maintenance overhead without proportional benefit.

Safeguard Coverage. CULPA may fail to identify certain complex explicit safeguard patterns, particularly those involving non-linear integer arithmetic or loop invariants with saturating operations. Precise modeling of such idioms would require symbolic execution and extensive manual annotations, neither of which scale to large codebases. Consequently, CULPA conservatively reports potential violations in these cases, contributing to missing identification of safeguard. Future work could explore hybrid approaches combining static analysis with targeted dynamic validation on paths identified by CULPA.

7 Related Work

Unsafe Rust and Safety Requirements. Researchers have explored how unsafe Rust can compromise the integrity of Rust programs [14,20,27,32,33,36,55,58]. Xu et al. analyzed hundreds of memory-safety issues, determining that safety assurances can be violated by `unsafe` code [55], while other works study how to protect the Rust program [20,36]. In addition, some scholars have investigated both memory-safety and concurrency issues, assessing both effects of eliminating unsafe code [33]. Our observation also suggests a significant correlation between memory-safety issues and the violation of safety requirements. Several works have proposed structured classifications of safety requirements for unsafe functions. Cui et al. [10] defined 19 Safety Properties by auditing unsafe APIs in the standard library, categorizing them into 12 preconditions and 7 post-conditions that related to all categories of undefined behaviors. Rao et al. [35] characterized the structural patterns and isolation types of unsafe code encapsulation in real-world systems. They also formalized informal safety descriptions in Rustdoc into structured safety requirements using a 21 novel domain-specific language tags to facilitate modular auditing [34].

Table 7: Comparison on safety property and specification with following features of CULPA and prior works: classification taxonomy, format representation. Based on representation, we can infer that whether these rules are machine-executable themselves and the undefined behaviors they can cover. Only CULPA produces executable predicates and verifies the usage of them statically.

	Cui et al. [10]	Rao et al. [34]	**CULPA**
Taxonomy	19 SPs	21 tags	5 categories
Format	Natural Language	DSL	Executable Predicates
Machine-Executable	✗	✗	✓ static analysis
Coverage	all UB	all UB	memory-safety

Although these works identify and organize the safety conditions that callers should satisfy, they serve different purposes than CULPA's specifications. As summarized in Table 7, prior works produce property taxonomies expressed as natural-language labels or structured tags, whereas CULPA decomposes five memory-safety requirement that are machine-executable predicates, each is integrated into static analysis pipeline. We also provide a practical example to demonstrate the difference between prior works and CULPA's specification in Appendix C.

Bug Detection in Rust. Previous works employ static analysis [5, 6, 9, 23, 30] or dynamic analysis [7, 28, 31, 56, 57] to identify memory-safety bugs. Static approaches like MirChecker [23], detect memory corruption using numerical abstract domains and SMT-based reasoning; SafeDrop [9], targets use-after-free vulnerabilities from pointer aliasing and deallocation; Rudra [5] identifies bugs in panic-induced inconsistencies and incorrect trait implementations; and TypePulse [6] detects unsafe type conversions violating compiler-enforced validity invariants. Dynamic approaches include

RuMono [57] and FourFuzz [31] apply fuzzing with `unsafe` code prioritization, and RustSan [7] and ERASan [28] adapt AddressSanitizer to reduce instrumentation overhead by exploiting Rust's type-safety guarantees. LLM-based approach like Safe4U [22] inherits the specification properties from Cui et al. by decomposing natural-language contracts into classified sub-contracts and using LLMs to analyze all categories of undefined behaviors. As shown in Table 7, even though Safe4U leverages these properties as LLM prompts to perform bug detection, the specification remains not machine-executable. Compared to these approaches, CULPA converts ambiguous, vague safety documents into machine-executable specifications with abstract domain, parameter, and emission for deterministic analysis.

Formal Verification in Rust. Formal verification provides mathematically rigorous correctness guarantees for `unsafe` Rust code. Foundational work such as RustBelt [19] and RustHornBelt [25] establishes semantic models within interactive proof assistants to verify type-safety invariants. Automated deductive verifiers such as Prusti [3], Creusot [12], and Verus [21] enable functional correctness proofs using separation logic or linear ghost types, while RefinedRust [15] and Gillian-Rust [4] employ refinement types and symbolic execution for precise reasoning. The verify-rust-std project [40] further instruments the standard library with contract attributes (`#[requires]`, `#[ensures]`) and employs model checkers to verify that the standard library's implementation satisfies its contracts. These efforts share two fundamental differences from CULPA. First, they all verify the *implementation* of the standard library, whereas CULPA verifies the *usage* of the standard library: whether downstream code in third-party crates satisfies those preconditions in the first place. The two directions are complementary but address distinct analysis problems. Second, their specifications are embedded in the verified source code (as proof annotations, ghost code, or harnesses) and are not designed to be extracted and applied to external codebase. CULPA's specification pool fills this gap by encoding caller-facing requirements as portable, executable predicates that can be checked against arbitrary Rust packages at ecosystem scale.

8 Conclusion

In this paper, we present CULPA, a static analysis framework that detects memory-safety bugs in Rust by verifying adherence to documented safety requirements of unsafe operations. Unlike prior tools that target specific bug patterns, CULPA addresses the root cause of unsafe misuse through compositional contract specifications, metadata-aware memory region tracking, and implicit safeguard reasoning. Our evaluation on the top 1,000 Rust packages demonstrates that CULPA discovers 55 previously unknown memory-safety bugs, with 29 confirmed by developers.

Acknowledgment

We thank our shepherd and the reviewers for their insightful feedback. This work is partially supported by the US Office of Naval Research grant N00014-23-1-2122, the Commonwealth Cyber Initiative: AI for Cybersecurity (N-3Q25-002), and a gift from VISA Inc.

9 Ethical Considerations

Stakeholders. Our study involves three primary stakeholder groups. First, developers of Rust crates may face scrutiny from identified memory-safety vulnerabilities, though our analysis focuses on systemic unsafe code patterns rather than individual fault attribution. Second, users of affected crates rely on the security guarantees provided by the Rust ecosystem to make informed decisions about dependency selection. Third, the Rust community and language maintainers may benefit from insights into prevalent unsafe code misuse patterns, which we view as an opportunity to strengthen safety documentation and tooling.

Data Collection. We analyzed publicly available Rust packages from `crates.io`, including source code, documentation, and metadata. Our data collection process was designed to minimize disruption to the platform by implementing rate limiting and caching mechanisms. We did not access any private repositories or proprietary code during our study.

Static Analysis Scope. We performed static analysis on the collected packages to identify potential memory-safety violations. All analysis was conducted on code we downloaded to our own infrastructure. We did not execute untrusted code or perform dynamic analysis that could affect third-party systems. The analysis focused on identifying violations of documented safety requirements in unsafe code blocks.

Responsible Disclosure. We reported all confirmed vulnerabilities to affected crate maintainers through official channels (e.g., RustSec, CVE, github issues). The developers are given adequate time to address identified issues and coordinate CVE/RustSec advisory assignments before publication. The RustSec IDs are: RUSTSEC-2025-0137, RUSTSEC-2025-0140, RUSTSEC-2025-0143, RUSTSEC-2026-0001, RUSTSEC-2026-0008; The CVE ID is CVE-2026-0810.

10 Open Science

The data artifacts of this paper will be made publicly available, including source code, detected bugs, and assigned RustSec/CVE IDs. We disclose only those issues that have been acknowledged and resolved by the developers. Issues that remain unresolved at the time of writing are not included in detail. All data is available at: https://zenodo.org/records/20359382.

References

[1] Apache, "Official Rust implementation of Apache Arrow," 2026, version 57.2.0, Accessed: 2026-01-25. [Online]. Available: https://github.com/apache/arrow-rs

[2] Apache Software Foundation, "Apache DataFusion," https://github.com/apache/datafusion, 2026, extensible query engine written in Rust. Accessed: 2026-02-05.

[3] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods Conference*. Springer, 2022, pp. 88–108.

[4] S.-É. Ayoun, X. Denis, P. Maksimović, and P. Gardner, "A hybrid approach to semi-automated rust verification," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 970–992, 2025.

[5] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 84–99. [Online]. Available: https://doi.org/10.1145/3477132.3483570

[6] H.-M. Chen, X. He, S. Wang, X. Zhang, and K. Sun, "Typepulse: detecting type confusion bugs in rust programs," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[7] K. Cho, J. Kim, K. D. Duy, H. Lim, and H. Lee, "RustSan: Retrofitting AddressSanitizer for efficient sanitization of rust," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3729–3746. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon

[8] A. Crichton *et al.*, "git2 - Rust," https://docs.rs/git2/latest/git2/, 2024, accessed: 2026-02-03.

[9] M. Cui, C. Chen, H. Xu, and Y. Zhou, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–21, 2023.

[10] M. Cui, S. Sun, H. Xu, and Y. Zhou, "Is unsafe an achilles' heel? a comprehensive study of safety requirements in unsafe rust programming," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[11] R. A. Database, "RUSTSEC-2025-0105: direct_ring_buffer: Uninitialized memory exposure in create_ring_buffer," https://rustsec.org/advisories/RUSTSEC-2025-0105.html, 2025, accessed: 2026-02-02.

[12] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *International Conference on Formal Engineering Methods*. Springer, 2022, pp. 90–105.

[13] T. R. P. Developers, "Function std::mem::transmute," https://doc.rust-lang.org/std/mem/fn.transmute.html, accessed: 2026-02-02.

[14] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 246–257, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220302286

[15] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer, "Refinedrust: A type system for high-assurance verification of rust programs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1115–1139, 2024.

[16] A. Gaynor, "What science can tell us about c and c++'s security," https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, empirical evidence on memory safety vulnerabilities in large codebases.

[17] Google, "Google Chrome Browser," https://www.google.com/chrome/, 2026, accessed: 2026-02-04.

[18] Google Security Blog, "Rust in android: Move fast, fix things," https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html, 2025, accessed: 2026-01-25.

[19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.

[20] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: Automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 132–148.

[21] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[22] H. Li, B. Wang, X. Hu, and X. Xia, "Safe4u: Identifying unsound safe encapsulations of unsafe calls in rust using llms," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728890

[23] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2183–2196. [Online]. Available: https://doi.org/10.1145/3460120.3484541

[24] Linux Kernel Organization, "The Linux Kernel Archives," https://www.kernel.org/, 2026, accessed: 2026-02-02.

[25] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer, "Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 841–856.

[26] MemorySafety, "What is memory safety and why does it matter?" https://www.memorysafety.org/docs/memory-safety/.

[27] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks." in *NDSS*, 2022.

[28] J. Min, D. Yu, S. Jeong, D. Song, and Y. Jeon, "Erasan: Efficient rust address sanitizer," *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4053–4068, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:272434127

[29] National Security Agency, "Software memory safety," https://media.defense.gov/2023/Apr/27/2003210083/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY_V1.1.PDF, cybersecurity Information Sheet, Version 1.1.

[30] V. Nitin, A. Mulhern, S. Arora, and B. Ray, "Yuga: Automatically detecting lifetime annotation bugs in the rust language," *IEEE Transactions on Software Engineering*, vol. 50, pp. 2602–2613, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:263908830

[31] D. Paaßen, J.-R. Giesen, and L. Davi, "Targeted fuzzing for unsafe rust code: Leveraging selective instrumentation," in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 488–499. [Online]. Available: https://doi.org/10.1145/3756681.3756956

[32] M. Papaevripides and E. Athanasopoulos, "Exploiting mixed binaries," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 2, pp. 1–29, 2021.

[33] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Replication package for article: Understanding memory and thread safety practices and issues in real-world rust programs," *Artifact Digital Object Group*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:215761919

[34] Z. Rao, H. Tian, X. Wang, and H. Xu, "Annotating and auditing the safety properties of unsafe rust," *arXiv preprint arXiv:2504.21312*, 2025.

[35] Z. Rao, Y. Yang, and H. Xu, "Characterizing unsafe code encapsulation in real-world rust systems," *arXiv preprint arXiv:2406.07936*, 2024.

[36] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 824–836.

[37] Rust, "core::slice::from_raw_parts," https://doc.rust-lang.org/core/slice/fn.from_raw_parts.html, Rust Language Team, 2025, accessed: 2025-12-13.

[38] Rust, "Unsafe rust," https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html, Rust Language Team, 2025, accessed: 2025-12-13.

[39] ——, "The mid-level ir," https://rustc-dev-guide.rust-lang.org/mir/index.html, 2026, accessed: 2026-02-02.

[40] Rust official, "Verify rust standard library effort," https://model-checking.github.io/verify-rust-std/, 2026, accessed: 2026-05-17.

[41] Rust standard libraries, "set-len in std::vec," https://doc.rust-lang.org/std/vec/struct.Vec.html#method.set_len, Rust Language Team, 2025, accessed: 2025-12-23.

[42] Rust Team, "rust analyzer," https://rust-analyzer.github.io/, 2026, accessed: 2026-05-21.

[43] RUSTSEC, "Rustsec-2024-0431 - xous: Unsound usages of core::slice::from_raw_parts," https://rustsec.org/advisories/RUSTSEC-2024-0431.html, RUSTSEC, 2024, accessed: 2026-02-02.

[44] RustSec Advisory Database, "RUSTSEC-2023-0047: lmdb-rs: impl FromMdbValue for bool is unsound," https://rustsec.org/advisories/RUSTSEC-2023-0047.html, 2023, accessed: 2026-02-02.

[45] Steven J. Vaughan-Nichols, "Rust goes mainstream in the linux kernel," https://thenewstack.io/rust-goes-mainstream-in-the-linux-kernel/, 2025.

[46] L. Szekeres, M. Payer, T. Wei, and D. X. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 48–62. [Online]. Available: https://ieeexplore.ieee.org/document/6547101/

[47] R. Team, "Rust programming language," 2026, accessed: 2026-02-01. [Online]. Available: https://rust-lang.org/

[48] The Apache Software Foundation, "Apache Arrow: The universal columnar format and multi-language toolbox for fast data interchange and in-memory analytics," 2026, accessed: 2026-01-25. [Online]. Available: https://arrow.apache.org/

[49] The crates.io team, "arrow: Rust implementation of Apache Arrow — reverse dependencies," crates.io: Rust Package Registry, 2026, accessed: 2026-01-25. [Online]. Available: https://crates.io/crates/arrow/reverse_dependencies

[50] The Rust Project Developers, "std module ptr," https://doc.rust-lang.org/std/ptr/index.html, Rust Language Team, 2025, accessed: 2025-12-13.

[51] The Rust RFC Book, "1105-api-evolution," https://rust-lang.github.io/rfcs/1105-api-evolution.html, 2026, accessed: 2026-05-20.

[52] Top Memory Corruption Bugs, "Understanding memory safety vulnerabilities: Top memory corruption bugs and how to address them," https://runsafesecurity.com/blog/memory-safety-vulnerabilities/.

[53] Trust-in-soft, "memory-safety-issues-still-plague-new-c-cpp-code," https://www.trust-in-soft.com/resources/blogs/memory-safety-issues-still-plague-new-c-cpp-code, 2024.

[54] Wikipedia, "Pointer analysis," https://en.wikipedia.org/wiki/Pointer_analysis, Wikipedia, 2025, accessed: 2025-12-23.

[55] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, pp. 3:1–3:25, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:222080332

[56] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1521–1533, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268297575

[57] Y. Zhang, J. xia Wu, and H. Xu, "Rumono: Fuzz driver synthesis for rust generic apis," *ACM Transactions on Software Engineering and Methodology*, vol. 34, pp. 1 – 28, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:266348280

[58] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu, "Towards understanding the runtime performance of rust," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6.

```
1 [dependencies.derive_more]
2 version = "0.99"
3 default-features = false
4
5 [features]
6 std = [ ..., "derive_more/std", ... ]
```

Listing 7: This example shows the dependency conflicts in `Cargo.toml`: The version requirement of dependency `derive_more` that is 0.99.x with the feature of `std` enabled never exists.

A Toolchain Upgrade for Failed Packages

The dependency conflicts remain due to the contradiction within the package manifest itself, which cannot be resolved by any toolchain version. As shown in Listing 7, the failed package `alloy-consensus` restricts its dependency `derive_more` to versions 0.99 at line 2 and enable its `std` feature. However, this `std` feature is introduced in `derive_more 2.0` as part of a major API revision whereas the version of 0.99.x do not have such flag. When `Cargo`'s dependency resolver tries to find a single version of `derive_more` that satisfies both constraints: fall within `^0.99` *and* expose a `std` feature, no such version exists. The conflict occurs entirely within the package's own manifest rather than in the compiler, changing `rustc` to any older or newer nightly has no effect on the outcome.

B Comparison with Existing Tools

While all five existing tools target unsafe code, their detection capabilities differ substantially due to fundamental differences in analysis methodology. MirChecker targets runtime panics and memory-safety violations (use-after-free, double-free, out-of-bounds), but detects no `InBound` violations in our dataset. This gap arises because MirChecker focuses on array indexing (`array[i]`), which triggers runtime panics, rather than unchecked pointer arithmetic (`ptr.add(offset)`), which silently violates bounds. Rudra employs data-flow pattern matching to identify specific unsafe idioms (`WriteFlow`,

`CopyFlow`, `SliceFromRaw`). However, it misses violations lacking recognizable patterns. For example, missing bounds checks before `ptr.add(offset)` produce no characteristic data-flow signature. TypePulse analyzes type-level properties (bit patterns, memory layout) but restricts its scope to type conversions, missing memory-safety violations arising from pointer manipulation. SafeDrop focuses exclusively on ownership violations and does not verify pointer arithmetic, alignment, or initialization requirements. The fundamental limitation of four static analyzers above is their reliance on pattern-based heuristics: they detect known bug patterns rather than systematically verifying safety requirements.

Even though Safe4U achieves higher performance than traditional static analysis tools by LLM, it still failed to detect more than half of the bugs in the dataset due to following reasons: (i) limitation in inter-procedural analysis (as claimed in the paper [22], "Safe4U only involves items that are directly referenced"), which means the unsafe function wrapped deep inside the private function cannot be analyzed; (ii) the usage of rust-analyzer [42] failed to resolve callees, e.g., failures of returning target for intrinsics or generic trait methods dispatch; (iii) limitation in candidate extraction, e.g., grep-based candidate extractor selects the first-ranked match. When multiple functions share the same name in different `impl` blocks, the bug can be hidden in the later ones.

C Mappings of Safety Requirements

Table 8 maps CULPA's five requirement categories to the safety properties defined by Cui et al. [10] and the safety tags derived by Rao et al. [34] Although the number of CULPA's categories is much less than other two works, we still cover their categories with different granularity in our scope of memory-safety bugs. For instance, CULPA's `Initialization` category can cover four properties of Cui et al. (`Initialized`, `Encoding`, `ConsistentLayout`, `Untyped`) and Rao et al. across five tags (`Init`, `ValidString`, `ValidCStr`, `Typed`, `Unwrap`). Overall, 12/19 of Cui et al.'s properties and 15/21 of Rao et al.'s tags fall within CULPA's memory-safety scope; the remainder address concurrency, platform, or non-UB concerns that lie outside CULPA's scope (§3.3).

Example. To illustrate how these differences manifest in practice, we compare how each work represents the safety requirements of `slice::from_raw_parts` [37], which constructs a slice reference from a raw pointer and a length.

Cui et al. label this API with seven properties: *Allocated* (pointer must not be null or dangling), *Dereferencable* (memory range within a single allocation), *Aligned* (pointer aligned for T), *ConsistentLayout* (type T compatible with stored memory), *Initialized* (elements properly initialized), *RelativeBound* (`len * size_of::<T>()` $\leq$ `isize::MAX`), and *Aliasing&Mutating* (no concurrent mutation during the slice's lifetime). These labels correctly identify *which* properties the caller must satisfy; however, they are natural-language descriptions that serve as a documentation taxonomy, not as inputs to a static analyzer.

Table 8: Mapping between safety property classifications and CULPA's requirement categories. 12/19 of Cui et al.'s properties and 15/21 of Rao et al.'s tags fall within CULPA's memory-safety scope; the remaining properties are outside of CULPA's scope and not listed here.

CULPA Specs	**Cui et al. SPs**	**Rao et al. Tags**
NonNull	Allocated	!Null
Alignment	Aligned	Align Size !Padding
Initialization	Initialized Encoding ConsistentLayout Untyped	Init ValidString ValidCStr Typed Unwrap
InBound	ConstNumericBound RelativeNumericBound Dereferencable	Allocated InBound ValidNum !Overlap
ExclusiveAlias	DualOwner Aliasing&Mutating Freed	Owning Alias

Rao et al. annotate the same API with six parameterized tags: `ValidPtr(data,T,len)`, `Init(data,T,len)`, `Alive(data,'a)`, `Alias(data)`, `Align(data,T)`, and `ValidNum(len*sizeof(T),[0,isize::MAX])`. The parameterized DSL is more structured than natural-language labels and supports lightweight consistency checking via their *safety-tool* linter. However, the authors explicitly leave the safety contract verification "a separate problem orthogonal to their focus."

CULPA assigns four executable predicates in the specifications of this API: `Initialization(range:[0,len),T)`, `NonNull(arg0)`, `Aligned(arg0,T)`, `ExclusiveAlias(arg0)`. Each predicate has a corresponding analysis pipeline: `NonNull` checks null-pointer evidence through data-flow tracking; `Aligned` resolves generic-type alignment via trait-bound analysis; `Initialization` verifies both initialization coverage and content validity against type-specific requirements; `ExclusiveAlias` tracks alias escapability across pointer lifetimes. Crucially, when all four predicates are verified, CULPA *emits* an initialization guarantee on the returned slice, which is a compositional evidence that downstream operations (e.g., `ptr::read`) can consume to satisfy their own requirements without re-verification. This cross-operation emission mechanism is the key capability that distinguishes executable predicates from property labels or structured tags.