Verifying the Rust Standard Library

Byron Cook[1,3], Remi Delmas[1], Zyad Hassan[1], Bart Jacobs[4], Ranjit Jhala[5], Rahul Kumar[1], Felipe R. Monteiro[1], Thanh Nguyen[1], Rebecca Rumbul[2], Michael Tautschnig[1,6], Celina Val[1], and Carolyn Zech[7]

[1] Amazon Web Services
[2] Rust Foundation
[3] University College London
[4] KU Leuven
[5] University of California, San Diego
[6] Queen Mary University of London
[7] Massachusetts Institute of Technology

Abstract. Rust's type system prevents many classes of memory errors, yet its standard library relies heavily on `unsafe` code whose correctness is validated through testing, including dynamic checks under Miri, but lacks static verification. We present what is, to the best of our knowledge, the largest verification campaign reported for a software library: an open, crowdsourced effort that integrates complementary verification tools into the continuous integration of a verification repository forked from the Rust standard library. We analyze the campaign's effectiveness, discuss the practical value of machine-checked proofs for a subset of undefined behaviors (e.g., out-of-bounds access, null and dangling pointer dereferences, and use of uninitialized memory), and frame the remaining obstacles as open challenges for the formal-methods community.

1 Introduction

Rust's ownership-based type system statically prevents data races and many classes of memory errors, a guarantee that has driven adoption in operating system kernels [9], browser engines [1], cryptographic libraries [7], and safety-critical embedded systems [35]. Yet the guarantee has a gap. The Rust standard library is widely regarded as battle-tested software[8], but it contains approximately 7,500 `unsafe` functions and 3,000 additional safe functions that internally use `unsafe` code. Inside these `unsafe` regions, correctness relies on careful manual reasoning and testing rather than on machine-checked proof [3,15]. Testing, including dynamic analysis under Miri [30], has been effective at catching many issues, but it can only exercise a finite set of executions. Verification offers a qualitatively different assurance: a machine-checked proof that a given property holds for *all* inputs, providing a permanent, auditable guarantee.

Landmark projects and industrial efforts have demonstrated that verification can scale to compilers, microkernels, cryptographic stacks, and continuous-integration workflows [34,31,5,28,12,11]. However, a recent survey of 32 deployed

[8] `https://doc.rust-lang.org/std/`

verified systems [24] finds that even the largest efforts verified codebases of at most hundreds of thousands of lines, required dedicated teams of verification experts, and spanned multiple person-years of effort. The Rust standard library presents a qualitatively different challenge: approximately 34,000 functions across the `core`, `alloc`, and `std` crates, a codebase that ships a new release every six weeks, and a contributor base that spans industry, academia, and the open-source community. Verifying it demands not only advances in verification technology but also a new model for organizing proof work at scale.

This paper reports on the first large-scale effort to prove the absence of undefined behavior in the Rust standard library. The project is organized as an open challenge program that invites contributions from any individual or team. Since its launch in November 2024, the effort has integrated four verification tools (i.e., Kani [37], ESBMC [19], VeriFast [26], and Flux [33]) into continuous integration, with four additional tools under review. An independent study of the project's early phase is reported by Le Blanc and Lam [8]. To the best of our knowledge, and based on the most comprehensive survey of deployed verified systems to date [24], this is the largest verification campaign reported for a software library, both by number of functions mechanically proved free of classes of undefined behavior and by number of independent contributors.

We make four contributions. First, we describe the design of a crowdsourced verification campaign organized around challenges with financial rewards, and analyze how this structure enabled multi-tool integration and attracted contributions (Section 3). Second, we present Autoharness, a tool that automatically generates proof harnesses at the level of Rust's MIR intermediate representation; Autoharness produced 16,748 harnesses, including 4,645 for unsafe functions and 1,126 for safe abstractions over unsafe code, of which 11,970 were verified against Kani's supported classes of undefined behavior. Across both automatic and manual harnesses, 989 functions were verified against formal function contracts (Section 4). Third, we report on the integration of four verification tools into continuous integration, providing concrete evidence that multi-tool verification of a production Rust codebase is practical (Section 3.2). Fourth, we identify and analyze the main technical obstacles (e.g., generic function verification, intrinsic modeling, and concurrency) and frame them as open problems for the formal-methods community (Section 6). The project, its open challenges, and all verification artifacts are publicly available[9].

2 How is Rust unsafe?

Rust's ownership-based type system statically prevents data races and many classes of memory errors [28], but the standard library supplements these guarantees with `unsafe` code that accesses five additional operations the compiler does not check for memory safety: dereferencing raw pointers, calling unsafe functions, accessing mutable statics, implementing unsafe traits, and accessing

[9] `https://github.com/model-checking/verify-rust-std`

fields of unions [29]. Importantly, `unsafe` Rust is still Rust, not C: the code is written in the same language and compiled by the same compiler, but the programmer takes responsibility for a small set of properties the compiler cannot verify. The borrow checker, type checking, and lifetime analysis still apply inside `unsafe` regions, but the soundness of these additional operations rests on the developer's reasoning, typically recorded only in natural-language comments [3,16]. Over the last three years, developers have reported over 74 soundness bugs[10]; 18 CVEs have been filed historically.[11]

Throughout this paper, *absence of undefined behavior* refers to the property that a function, when executed on any valid input, does not trigger any of the behaviors that the Rust Reference classifies as undefined [36]. Absence of undefined behavior is a sufficient condition for memory safety, but not for full *correctness*, which additionally requires functional correctness, liveness, and security properties. The current verification tooling does not check for all classes of undefined behavior: Kani does not detect violations of the pointer aliasing rules (as formalized by Stacked Borrows [27] and Tree Borrows [38]), data races, invalid uses of inline assembly, or all forms of provenance-related undefined behavior. Section 4 details the precise scope of each tool, and Section 7 discusses the implications of this incomplete coverage.

3 Scaling the verification effort

The scale of the Rust standard library makes verification difficult to centralize within a single team or institution. To address this, we designed a community-oriented verification program, run by the Rust Foundation[12], that treats the verification of the standard library as an open contest.

3.1 Crowd-sourcing the verification effort

Verification tasks are organized into *challenges*. Each challenge specifies a concrete verification target, a list of assumptions, explicit success criteria, and a financial reward disbursed upon completion. The status of all challenges is maintained in the project website[13]. Participants fork the repository, implement a candidate solution, and submit it as a pull request reviewed by a technical review committee. Accepted solutions are merged into a dedicated fork[14] of the Rust repository that serves as the verification target. The repository has received more than 450 pull requests from at least 21 unique external contributors affiliated with four distinct institutions.

Contributors specify contracts and loop invariants using the contract systems provided by the participating tools. For Kani-based Autoharness, these contracts

[10] `https://github.com/rust-lang/rust/labels/I-unsound`
[11] `https://rustsec.org/packages/std.html`
[12] `https://rustfoundation.org/`
[13] `https://model-checking.github.io/verify-rust-std/`
[14] `https://github.com/model-checking/verify-rust-std`

express safety-related preconditions, postconditions, type invariants, and loop conditions targeting absence of undefined behavior. The contest welcomes any verification tool for Rust programs, and some tools go beyond safety properties: for example, the VeriFast proofs in Section 4.3 establish functional correctness of linked-list operations.

3.2 Tool Integration and Continuous Verification

The contest is designed to be tool-agnostic. Any verification tool is eligible, provided it can operate on the Rust standard library, can be integrated into continuous integration (CI), and provides clear soundness guarantees, typically supported by peer-reviewed publications. The project currently supports Kani [15], ESBMC [19], Flux [33], and VeriFast [26]. Four additional tools are under review: Verus [32], Creusot [18], KRust [39], and RAPx [16]. This diversity is essential because no single tool can discharge all verification conditions; architecture-specific intrinsics, pointer-heavy code, concurrency primitives, and loops with complex invariants require complementary reasoning techniques. Each tool targets a different set of properties: Kani and ESBMC perform bounded model checking for memory safety violations (e.g., out-of-bounds access, null and dangling pointer dereferences, use of uninitialized memory, and arithmetic overflow), and support unbounded analysis when loops and recursive functions are annotated with loop contracts and function contracts, respectively; Flux checks refinement types that encode numeric bounds and safety preconditions; and VeriFast uses separation logic to verify absence of all undefined behavior, including pointer aliasing violations, for the functions it covers.

All verification is performed automatically through CI. The verification repository is a maintained fork of upstream `rust-lang/rust`, periodically synchronized so that proofs remain current. Each pull request triggers the full suite of verification tools on all active proofs; any violation is detected immediately (cf. Section 4.4). This continuous verification model, inspired by prior industrial efforts [13,12,11], ensures that proofs remain valid across revisions and guards against regressions.

4 Verification progress

We automated the collection of both code and proof coverage metrics to track the state of the verification effort over time. Each CI run produces a JSON report that records, for every function in the standard library, whether a proof harness exists, whether the harness succeeded, and which tool was responsible for the proof. As of `nightly-2025-10-08` snapshot, the `core`, `alloc`, and `std` crates together contain 33,955 functions. The metrics below cover all three crates.

Over the first fifteen months of the project, contributors manually wrote 725 proof harnesses using Kani (694 with function contracts), plus more than

[15] `https://github.com/model-checking/kani`
[16] `https://github.com/safer-rust/RAPx`

50 proofs constructed with VeriFast. Because the project began with manual proof engineering, these early harnesses span a wide range of complexity, from straightforward type conversions to intricate pointer manipulations, loop-heavy algorithms, and linked data structures requiring carefully crafted preconditions and loop invariants. Figure 1 shows the growth of manually written function contracts (i.e., precondition/postcondition annotations) and their verified subset over time, broken down by function category. After an initial ramp-up driven by the first wave of challenge solutions, the rate of new contracts plateaued around October 2025. At that point, manual harnesses covered only a small fraction of the standard library's functions, leaving the vast majority unverified. This plateau illustrates a fundamental scalability limitation: manual proof engineering alone cannot reach the tens of thousands of functions in the standard library.

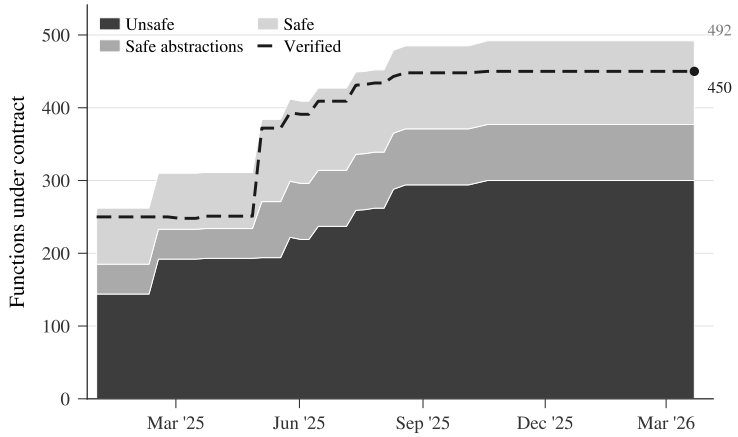

Fig. 1. Growth of manually written function contracts over time (`core` + `std`), broken down by function category: *unsafe* functions (marked `unsafe fn`), *safe abstractions* (safe functions that internally use `unsafe` blocks), and *safe* (all other functions). The dashed line shows the subset of contracts whose proofs pass verification (always $\leq$ the total). The plateau after October 2025 motivated the development of automatic harness generation.

Even with limited coverage, the manual verification effort demonstrated how effective function contracts could be for verifying the standard library, and this had a significant institutional impact. The Rust language team accepted contracts as an experimental language feature[17], and the Rust Project adopted a formal goal to instrument the standard library with safety contracts[18].

[17] `https://doc.rust-lang.org/beta/unstable-book/language-features/contracts.html`

[18] `https://rust-lang.github.io/rust-project-goals/2024h2/std-verification.html`

To overcome the scalability barrier, we developed *Autoharness*, a tool that automatically generates proof harnesses at the level of Rust's MIR intermediate representation. A *proof harness* is analogous to a unit test, but instead of calling a function with concrete inputs, it calls the function with *symbolic* (nondeterministic) values that represent all possible inputs simultaneously. A model checker then exhaustively explores all reachable execution paths, turning the harness into a formal verification problem rather than a test.

4.1 Autoharness: Design and Workflow

Autoharness[19] is implemented as a compiler pass inside Kani. Given a crate, it enumerates every function definition, determines which functions are eligible for automatic harness generation, and for each eligible function synthesizes a proof harness that calls the function on fully nondeterministic inputs. The generated harness is then verified by Kani's back-end (e.g., CBMC), which checks for the absence of its supported classes of undefined behavior (UB) along all reachable execution paths.

Eligibility filtering. A function is eligible if it satisfies three conditions: (i) it is *monomorphic*, i.e., it has no unresolved generic type parameters; (ii) every argument type either implements or can automatically derive the `kani::Arbitrary` trait, which generates a fully symbolic value covering all bit-valid representations of that type; and (iii) it is not a Kani-internal implementation function. Functions that fail any condition are recorded with a skip reason (e.g., generic, missing `Arbitrary`, no body, or internal) and excluded from harness generation. User-supplied include and exclude patterns can further restrict the set.

Nondeterministic inputs and type invariants. A key subtlety is the distinction between *validity invariants* and *safety invariants* [36]. Validity invariants must hold at all times and are exploited by the compiler for optimizations (e.g., a `bool` is always 0 or 1); Kani's `Arbitrary` implementations respect these by construction. Safety invariants, however, are semantic properties that safe code may assume but that the type system does not enforce (e.g., `Duration`'s nanosecond field must be less than 10^9). A derived `Arbitrary` implementation generates all bit-valid values of a type but may violate its safety invariants, potentially causing false positives when the function under test assumes those invariants hold. To address this, we introduced an `Invariant` trait into the standard library that allows types to express their safety invariants programmatically via an `is_safe()` predicate. When a function carries a contract, Autoharness generates a contract-verification harness that assumes the preconditions, which can include `Invariant` checks, restricting inputs to states that satisfy the type's safety invariants. For functions without contracts, the proof is valid over all bit-valid inputs; any failure indicates genuine UB, while the absence of failure

[19] `https://model-checking.github.io/kani/reference/experimental/autoharness.html`

means the function is safe for all such inputs. For unsafe functions specifically, this distinction is important. An unsafe function without a contract is verified against all bit-valid inputs with no assumed preconditions. If the proof succeeds, the function cannot trigger any of Kani's supported classes of UB regardless of how it is called, a strong guarantee. If it fails, the failure may reflect genuine UB or a missing precondition that callers are expected to establish; such functions require function/loop contracts to verify meaningfully.

Harness synthesis. For each eligible function f with argument types $\tau_1, \ldots, \tau_n$, Autoharness constructs a new MIR function body that:

1. allocates a local variable x_i for each argument and initializes it by calling `kani::any::<`τ_i`>()`, which produces a fully nondeterministic value of type τ_i;
2. calls $f(x_1, \ldots, x_n)$.

If f carries a function contract (i.e., preconditions and postconditions), the harness is generated as a contract-verification harness: Kani assumes the preconditions, executes f, and checks the postconditions. Similarly, if f contains loop contracts, Autoharness detects and verifies them. If f has no contract, the harness simply calls f on unconstrained inputs (as shown in Listing 1.1), and Kani checks for its supported classes of undefined behavior[20] during execution. Crucially, all harnesses are generated internally at the MIR level without modifying the crate's source code, which makes the approach suitable for continuous integration.

Listing 1.1. Generated harness for a safe wrapper (simplified). Kani verifies this harness by exploring all possible values of `self_val` and `rhs`, checking that no execution path triggers UB. Because `wrapping_shl` is defined to wrap on overflow, the proof succeeds for all inputs.

```
// Target function (in core::num)
pub const fn wrapping_shl(self, rhs: u32) -> u32 {
    unsafe { ... } // intrinsic call
}

// Harness generated by autoharness (conceptual)
#[kani::proof]
fn check_wrapping_shl() {
    let self_val: u32 = kani::any();
    let rhs: u32 = kani::any();
    let _ = self_val.wrapping_shl(rhs);
}
```

For argument types that do not have a source-level `Arbitrary` implementation, Autoharness includes a companion pass (`AutomaticArbitraryPass`) that

[20] `https://model-checking.github.io/kani/undefined-behaviour.html`

synthesizes `Arbitrary` derivations for structs and enums at the MIR level, constructing each field nondeterministically. For enums, the pass generates a nondeterministic discriminant constrained to the valid variant indices and initializes the fields of the selected variant; variants with non-public fields or unsupported types cause the derivation to fail, and the function is recorded as skipped. The `Invariant` trait approach has a known limitation: types whose safety invariants are not yet annotated with `is_safe()` are tested over all bit-valid values, which may include states that violate the type's intended invariants. This can produce false positives (e.g., harness failures on inputs that no well-behaved caller would construct); in the current campaign, such failures are classified as "needs contract" rather than as bugs.

Property discharged. Kani instruments the code with assertions that check for UBs including out-of-bounds memory access, null and dangling pointer dereference, use of uninitialized memory, and arithmetic overflow in unsafe contexts. For functions without loop contracts, Kani unrolls loops up to a configurable bound and checks all reachable paths; an *unwinding assertion* verifies that the bound is sufficient, i.e., that no loop attempts to execute beyond the unwinding limit, and reports a verification failure otherwise. This yields a bounded proof of absence of the supported classes of UB. When a function carries loop contracts, Kani instead verifies that the invariant is inductive and uses it to abstract the loop, replacing exhaustive unrolling with a single-iteration check from an arbitrary state satisfying the invariant. A successful verification with loop contracts therefore constitutes an inductive proof for the annotated loops, removing the dependency on the unwinding bound. Kani does not yet support loop variant (`decreases`) clauses, so termination is not verified; this is planned for future work.

4.2 Manual vs. Automatic Harness Generation

Autoharness produced 16,748 proof harnesses (Figure 2), including 4,645 for unsafe functions and 1,126 for safe abstractions, an order-of-magnitude increase over the 725 manual harnesses accumulated over fifteen months. Not all produced harnesses pass verification: of the 16,748, 11,970 were successfully verified against Kani's supported classes of UB[21]; the remaining 4,778 failed due to missing models, timeouts, or unsupported features.

Interpreting the verified count. Not all 11,970 verified functions carry the same evidential weight. Table 1 breaks down the Autoharness results by function category and presence of contracts. Including the 694 contract-verified manual harnesses, a total of 989 functions (295 automatic + 694 manual) were verified

[21] Including out-of-bounds access, null and dangling pointer dereference, use of uninitialized memory, and arithmetic overflow in unsafe contexts; see `https://model-checking.github.io/kani/undefined-behaviour.html` for the full list. Classes not yet covered include violations of the aliasing model and some forms of provenance-related UB.

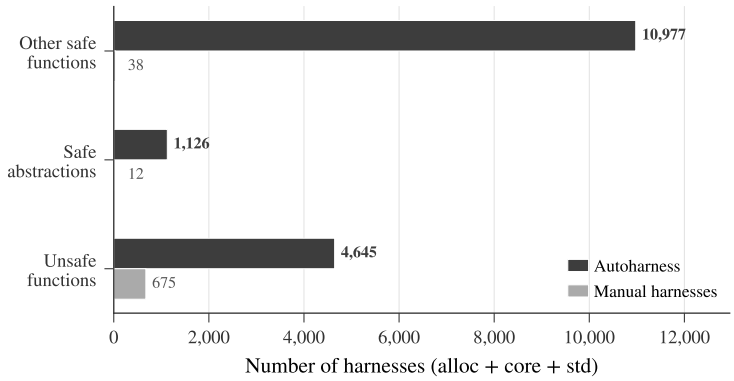

Fig. 2. Number of proof harnesses *produced* (not necessarily verified) by Autoharness vs. manual effort, across function categories (`alloc` + `core` + `std`, March 2026). Of the 16,748 automatically produced harnesses, 11,970 were successfully verified (Table 1).

Table 1. Autoharness verification results by function category and contract status (`alloc` + `core` + `std`).

Category	With contracts	Without	Total verified
Unsafe functions	184	622	806
Safe abstractions	44	926	970
Safe functions	67	10,127	10,194
All	295	11,675	11,970

against formal function contracts, the strongest guarantee the campaign provides. The 10,194 verified safe functions (85% of the Autoharness total) provide unambiguous value: Kani symbolically executes through all callees, including any unsafe code reached transitively, confirming that no path triggers UB from Kani's checked classes on all bit-valid inputs. The 970 verified safe abstractions similarly exercise the unsafe code they encapsulate. For the 806 verified unsafe functions, the interpretation depends on whether a contract is present. Of these, 184 carry function contracts whose preconditions are assumed during verification, yielding genuine safety proofs. The remaining 622 pass on fully nondeterministic, bit-valid inputs with no assumed preconditions. A passing proof for such a function means either that it is trivially safe for Kani's checked UB classes (and its `unsafe` marker relates to properties Kani does not yet check, such as aliasing), or that Kani's UB coverage is incomplete for the function's actual behavior. Distinguishing these two cases requires human review or complementary analysis techniques. These 622 functions are best understood as candidates surfaced by automated triage, and they motivate both expanding Kani's UB coverage and writing formal contracts for unsafe functions.

To complement Autoharness, we experimented with LLM-based contract synthesis, translating natural-language `SAFETY` comments into function contracts. This approach is preliminary: the generated contracts require manual review before merging, and a systematic evaluation of precision and recall is left to future work.

Despite its effectiveness, Kani cannot yet handle all functions. Of the 17,207 functions skipped during automatic harness generation, the dominant reason is generic type parameters (9,635 functions, 56%): Rust monomorphizes generics at compile time, and Autoharness currently cannot synthesize inputs for uninstantiated type parameters. The second largest category is missing `Arbitrary` trait implementations in Kani for the argument types of the target functions (3,826 functions, 22%), which prevents the generation of nondeterministic inputs for types such as raw pointers, references, and complex standard library types. Internal Kani implementation functions account for a further 3,246 (19%). These limitations represent concrete targets for future tool development: extending Kani's `Arbitrary` support to cover pointer and reference types, and supporting generic function verification, would together address 78% of the currently skipped functions.

The rapid growth in harness count created a new engineering challenge: compilation time. Kani compiles Rust code into a GOTO-style intermediate representation before symbolically executing it, and the sheer volume of 16,748 harnesses made this pipeline a bottleneck. To address this, we developed dedicated compiler benchmarking tools to expose performance bottlenecks and detect regressions in pull requests. Guided by these benchmarks, we introduced parallelization, caching, heuristically ordered code generation, and function stubbing, which together produced a 3.97× speedup in compilation for the standard library. This improvement made it practical to run thousands of proofs in CI within acceptable time budgets.

4.3 Case Study: Verifying `LinkedList` with VeriFast

The results presented so far are dominated by model checking via Autoharness. The challenge system, however, also produced proofs whose guarantees are qualitatively different: they reason about unbounded inputs and establish that verified functions cannot cause UB for any well-typed caller. We describe one such proof to make this contrast concrete.

The VeriFast proof of `LinkedList` targets one of the most pointer-intensive modules in the standard library: every public operation manipulates raw `NonNull` pointers, manually manages heap allocations, and must preserve a cycle-free invariant across insertions, removals, splits, and cursor traversals. The proof directly verifies 19 functions (e.g., `push_front`, `split_off`, `remove_current`) and implies the soundness of 5 additional non-`unsafe` functions (e.g., `contains`, `remove`, `drop`) that call only the verified functions. For each function, the proof establishes *soundness*: the function will not exhibit UB when called by a well-typed caller.

Listing 1.2. Separation-logic predicate `Nodes` and contract for `push_front_node`. The predicate recursively asserts exclusive ownership of each heap-allocated node. The contract transfers ownership of the new node into the list.

```
/*@ pred Nodes<T>(alloc_id: alloc_id_t, n: Option<NonNull<Node<T>>>,
                  prev: Option<NonNull<Node<T>>>,
                  last: Option<NonNull<Node<T>>>,
                  next: Option<NonNull<Node<T>>>;
                  nodes: list<NonNull<Node<T>>>) =
    if n == next {
        nodes == [] &*& last == prev
    } else {
        n == Option::Some(?n_) &*&
        alloc_block_in(alloc_id, n_.as_ptr() as *u8,
        Layout::new::<Node<T>>()) &*&
        struct_Node_padding(n_.as_ptr()) &*&
        (*n_.as_ptr()).prev |-> prev &*&
        (*n_.as_ptr()).next |-> ?next0 &*&
        pointer_within_limits(&(*n_.as_ptr()).element) == true &*&
        Nodes(alloc_id, next0, n, last, next, ?nodes0) &*&
        nodes == cons(n_, nodes0)
    }; @*/

unsafe fn push_front_node(&mut self, node: NonNull<Node<T>>)
    /*@
    req thread_token(?t) &*& *self |-> ?self0 &*&
        Allocator(t, self0.alloc, ?alloc_id) &*&
        Nodes(alloc_id, self0.head, None, self0.tail, None, ?nodes) &*&
        length(nodes) == self0.len &*&
        *node.as_ptr() |-> ?n &*&
        alloc_block_in(alloc_id, node.as_ptr() as *u8,
                       Layout::new::<Node<T>>());
    @*/
    /*@
    ens thread_token(t) &*& *self |-> ?self1 &*&
         Allocator(t, self1.alloc, alloc_id) &*&
         Nodes(alloc_id, self1.head, None, self1.tail, None,
               cons(node, nodes)) &*&
        self1.len == 1 + length(nodes) &*&
        (*node.as_ptr()).element |-> n.element;
    @*/
```

Central to the proof is a separation-logic predicate `Nodes` (see Listing 1.2) that recursively asserts exclusive ownership of each heap-allocated node between the list's `head` and `tail` pointers. Because the predicate's clauses are joined by a separating conjunction (`&*&`), the predicate structurally rules out aliasing and cycles. Each function receives a `req`/`ens` contract expressed in terms of `Nodes`; Listing 1.2 shows the contract for `push_front_node`, where the precondition requires separate ownership of the new node and the existing list, and the postcon-

dition returns a list whose node sequence is extended by one element. VeriFast then symbolically executes the function body, checking that every raw-pointer dereference, field mutation, and `Box::from_raw_in` call respects the ownership discipline.

For safe functions such as `clear` and `len`, VeriFast additionally verifies *semantic well-typedness*: the specification implies that the function cannot cause UB for any well-typed caller. The proof currently assumes that the allocator's lifetime equals `'static` (i.e., the global allocator). VeriFast does not yet fully verify Rust's pointer aliasing rules as formalized by Tree Borrows [38]: it verifies immutability of shared references while in use, but does not check the aliasing restrictions for mutable references or boxes, nor that references remain valid for the duration of a function call. This is a known soundness limitation, meaning the proof could miss aliasing-related UB.

A distinctive feature of this proof is its treatment of source modifications. VeriFast requires minor code changes to insert ghost commands (e.g., replacing `for` loops with `loop` loops, or replacing `Option::map` with first-order equivalents). To ensure these changes do not alter program behavior, the proof includes a *refinement checking step* that mechanically verifies that every behavior of the original code is also a behavior of the annotated version. The CI pipeline runs three steps in sequence: VeriFast checks the annotated code, the refinement checker relates it to the original, and a diff ensures the original matches the upstream repository. This three-stage pipeline illustrates the engineering overhead that deductive verification tools impose relative to model checking, but also the deeper guarantees they provide.

4.4 Bugs Found and Fixed

The verification effort has not uncovered any previously unknown memory safety vulnerabilities in the standard library. This null result is itself informative: it speaks to the effectiveness of Rust's existing testing infrastructure and Miri-based dynamic analysis [30] at catching memory safety bugs before they reach production. The primary value of the verification campaign is therefore not bug-finding but the guarantee it provides: a machine-checked proof, for each verified function, that the targeted classes of undefined behavior cannot occur. Testing can demonstrate the absence of bugs on exercised inputs; verification certifies their absence on all inputs within scope.

The effort has, however, revealed concrete specification and documentation issues, summarized in Table 2: missing safety annotations, incorrect `SAFETY` comments, and documentation errors that misrepresented function behavior. These findings illustrate a secondary benefit: the process of writing formal specifications forces a precise articulation of safety requirements that natural-language comments alone do not provide.

Table 2. Issues found through the verification effort.

Issue type	Component	Found via	Status
Incorrect SIMD shift results	`stdarch`	Challenge 15	Fixed upstream
Missing `unsafe` annotations	`core`	VeriFast proofs	Fixed upstream
Incorrect `SAFETY` comments	`core`	Flux proofs	Fixed upstream
Incorrect panic documentation	`core`	Kani proofs	Fixed upstream

5 Lessons learned

We reflect on three retrospective insights from the project: the cost of community coordination, specification design for unsafe code, and the need for tool diversity.

5.1 Planning for Consensus and Community

Our initial planning focused on technical milestones, but we significantly underbudgeted time for activities that depended on community consensus and institutional coordination. Integrating function contracts into the Rust compiler required broad support from the language team, library maintainers, and tool authors; even technically straightforward changes stalled when stakeholders had not been engaged early. Attracting external contributors required not only financially rewarded challenges but also a spectrum of task difficulty, clear documentation, and worked examples. Coordinating with external institutions required formal legal agreements that took months to finalize. The lesson is that upstream language changes, contributor onboarding, and institutional agreements should be treated as first-class milestones with explicit timelines, not as incidental tasks.

5.2 Specification Design for Unsafe Code

One of the more subtle lessons concerns the design of specifications in the presence of immediate undefined behavior. Rust defines[22] certain violations as triggering undefined behavior at the point where an invalid value or reference is created, not when it is first used. This has implications for where and how safety properties can be expressed. Le Blanc and Lam [8] independently identified this challenge; our experience confirms and extends their observation.

Consider `slice::from_raw_parts`, which takes a pointer and a length and returns a slice reference. If the function constructs a misaligned reference, undefined behavior occurs immediately, before any postcondition can be checked. In simple cases one can move the property into a precondition on the inputs, but for functions where the source of undefined behavior arises deeper in the call stack or depends on intermediate computations, expressing the right property as a precondition becomes much harder. We encountered this pattern in pointer-arithmetic functions, slice constructors, and transmute wrappers throughout

[22] `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`

`core`. The lesson is that specification languages aimed at safety verification need a principled way to express *internal* safety properties that must hold at particular program points, not only at function boundaries.

5.3 Tool Diversity in Practice

The initial plan relied almost exclusively on Kani, but the project evolved to include ESBMC, VeriFast, and Flux, driven by concrete technical needs. As discussed in Section 3.2, no single tool suffices: linked data structures require unbounded separation-logic reasoning, while model checking is more convenient for non-heap-intensive code. Beyond technical complementarity, the multi-tool design gives the Rust community comparative evidence on which tools and proof strategies justify their maintenance cost in a continuously evolving codebase.

This diversity has implications for the Rust contract language initiative. The experimental contract syntax adopted by the Rust language provides a shared baseline for expressing preconditions and postconditions as boolean expressions, and tools such as Kani and ESBMC can consume these directly. However, richer verification approaches require specification constructs that fall outside this baseline: VeriFast relies on separation-logic predicates and ghost state to reason about heap ownership, while Flux uses refinement types to track value-level invariants. Our experience suggests that the Rust ecosystem will need to accommodate tool-specific annotation layers alongside the common contract syntax, rather than converging on a single specification language.

6 Open technical challenges

Despite verifying over 10,000 functions, the Rust standard library remains far from fully verified. Many high-impact APIs, including `BTreeMap` internals, atomic types, `String`, iterators, vectors, deques, and reference-counted types, are only partially covered or not yet addressed.

6.1 Intrinsics and Model Coverage

Of the 4,778 harnesses that fail under the standard CI settings with Autoharness, a significant subset traces back to missing models: 71 unsupported Rust intrinsics and 813 unmodeled library functions, 721 of which are LLVM-internal SIMD intrinsics. The Rust standard library relies heavily on these compiler intrinsics and foreign functions whose semantics are defined outside of Rust itself. Any verification tool must either model these operations faithfully or treat them as opaque, which blocks verification of all callers.

This is not a limitation of any single tool. Bounded model checkers, deductive verifiers, and abstract interpreters alike need accurate models of intrinsics to reason about the code that calls them. Closing this gap requires a shared, tool-agnostic effort: developing formal specifications for the most frequently used intrinsics that any verification backend can consume. For SIMD intrinsics in

particular, the randomized testing approach used in Challenge 15 (cf. Section 3.2) may serve as a pragmatic intermediate step, providing high confidence where full formal models are not yet available. A systematic triage process to distinguish model gaps from genuine specification or code issues would further accelerate progress across all integrated tools.

6.2 History-Dependent and Parametric Specifications

Among failures not attributable to missing models, two recurring patterns stand out.

History-dependent properties. Some functions require that a value has a particular provenance or initialization history: a `MaybeUninit<T>` must be fully initialized, or a pointer must originate from a specific constructor (e.g., `Rc::from_raw` expects its argument was produced by `Rc::into_raw`). Expressing such properties as boundary-level contracts is often insufficient (cf. Section 5); they require tracking state at internal program points. Shadow memory instrumentation (Kani's `uninit-checks`), separation-logic ghost state (VeriFast), and abstract interpretation each address parts of this problem; extending them to cover unions and provenance tracking is future work.

Generic functions. Generic functions represent the single largest category of unverified code: automatic harness generation currently skips 9,635 of them because Rust monomorphizes generics at compile time. Bounded model checkers can enumerate representative type instantiations, but this may miss relevant cases and scales poorly. Deductive verifiers and refinement type systems can reason parametrically, verifying a generic body once for all instantiations, at the cost of richer specifications. A hybrid strategy that routes type-agnostic functions to instantiation-based checking and layout-dependent functions to parametric reasoning is a key open problem.

6.3 Concurrency and Data Races

Two of the 29 challenges target concurrency-related APIs (i.e., Challenge 7: atomic types and Challenge 27: `Arc`), yet neither has received accepted solutions. Data races in unsafe Rust constitute immediate undefined behavior, making this a significant gap in coverage.

VeriFast's separation-logic foundation inherently verifies absence of data races and can check the proof obligations implied by `Send` and `Sync` implementations; its test suite already includes proofs of simplified `Mutex` and `Arc` implementations using sequentially consistent atomics. However, the core difficulty for the standard library is specifying and verifying lock-free data structures under *relaxed* memory [17,25]. Types such as `Arc` and the atomic primitives rely on ordering modes ranging from relaxed to sequentially consistent, whose correctness depends on subtle inter-thread invariants that require a formal memory model to express. Tools such as VeriFast [26], Verus [32], and Gillian-Rust [4] target

multi-threaded Rust, but applying their concurrency reasoning to the standard library's lock-free types is future work.

7 Limitations and threats to validity

We consolidate the main limitations of the verification results reported in this paper.

Bounded reasoning. Kani, the primary verification backend for Autoharness, performs bounded model checking by default: loops are unrolled up to a configurable bound. Kani inserts unwinding assertions that detect when the bound is insufficient; if these assertions pass, the proof covers all reachable paths and is complete for the given function. If the bound is too low, the unwinding assertion fails and the proof is reported as unsuccessful. In practice, proof harnesses may use `kani::assume` to restrict the size of inputs so that unwinding assertions pass; such assumptions are explicit in the harness and limit the proof to the assumed input range. Kani also supports loop contracts, which replace exhaustive unrolling with an inductive argument and yield proofs that do not depend on the unwinding bound. Termination is not yet verified, as Kani does not yet support loop variant (`decreases`) clauses. Loop contracts must currently be supplied either manually or through the LLM-based contract synthesis approach described in Section 4.2.

Partial property coverage. Kani checks a subset of the undefined behaviors listed in the Rust Reference [36]. Notably, it does not detect violations of the pointer aliasing model (Tree Borrows [38]/Stacked Borrows [27], both implemented in Miri [30]), data races, invalid inline assembly, or all forms of provenance-related UB. Nor does it verify that type safety invariants are preserved across unsafe boundaries (e.g., that a `Vec`'s length does not exceed its capacity). A function that passes all current checks may still exhibit undefined behavior under a more complete model.

Specification and model gaps. Rust does not yet have a ratified formal specification; key aspects such as the aliasing model have competing proposals [27,38], and a future specification change could invalidate proofs that are sound under today's assumptions. At the implementation level, verification of functions that call compiler intrinsics or foreign functions depends on the fidelity of the models provided for those operations (cf. Section 6.1). Where models do exist, any inaccuracy can lead to unsound verification results.

Randomized testing for SIMD intrinsics. Challenge 15 was completed using randomized testing of executable models for 565 SIMD intrinsics, not formal proof. While the testing used logged seeds for reproducibility and uncovered two upstream bugs, it does not provide the same guarantee as model checking or deductive reasoning. These results should be interpreted as high-confidence validation rather than formal verification.

Tool-chain soundness. All verification results depend on the correctness of the underlying tool chains, including Kani's MIR-to-GOTO translation, CBMC's bounded model checker, the SAT/SMT solvers invoked during verification, and VeriFast's symbolic execution engine. A soundness bug in any of these components could cause a genuine undefined behavior to be missed. None of these tools have been formally verified themselves; their trustworthiness rests on extensive testing, fuzzing, and years of use in production settings.

Harness-level vs. function-level claims. Each Autoharness proof verifies a single function in isolation, called on fully nondeterministic inputs. This does not account for calling-context constraints: a function that is UB-free on all inputs is also UB-free in any calling context, but a function that relies on preconditions established by its callers may fail the harness even though it is safe in practice. Conversely, when an unsafe function's preconditions are specified only in natural-language comments, the proof does not verify that callers satisfy them. However, when preconditions are expressed as `#[kani::requires]` contracts, Kani does check them at each call site: a harness for a caller will fail if the caller does not satisfy the callee's contract. Compositional reasoning, where caller proofs discharge callee preconditions, is supported through manual function contracts but is not yet automated.

7.1 Related work

Landmark verification projects have delivered strong guarantees for compilers [34], microkernels [31], file systems [10], concurrent OS kernels [22], full software stacks [23], and cryptographic libraries [5]. A survey of 32 deployed verified systems [24] confirms that these efforts verify single, fixed-version artifacts in depth, carried out by dedicated teams over multiple years. Our work targets a different point in the design space: broad coverage of an evolving, multi-component library through a community-driven, multi-tool campaign.

Within the Rust ecosystem, RustBelt provides a semantic foundation for the type system and unsafe abstractions [28], and the aliasing discipline is formalized by Stacked Borrows [27] and its successor Tree Borrows [38]. Empirical studies characterize the prevalence and documentation challenges of unsafe code [3,15]. Verification tools with complementary strengths include Prusti [2], Creusot [18], Flux [33], Verus [32], Gillian-Rust [4], RefinedRust [20], and hax [6]. Rather than proposing a new tool, we study how multiple existing tools can be combined to verify a large, evolving library.

Complementary to static verification, the Miri interpreter [30] dynamically detects undefined behavior, including aliasing violations under Stacked Borrows and Tree Borrows, use of uninitialized memory, and data races. Miri operates on concrete inputs and supports a broader subset of Rust than Kani, including generic code and concurrency primitives, but does not support foreign functions. Miri has found dozens of bugs in the standard library and is integrated into its CI; integrating it into the verification workflow for functions beyond model checking's reach is a promising direction.

The closest line of work comes from continuous formal verification in industrial settings: prior efforts have integrated model checking and continuous assurance into CI workflows [12,11,14,13], and AutoCorres [21] automatically abstracts C code into higher-level proof representations. Our work extends this perspective to a community-driven setting, combining crowdsourced verification with automated harness generation and multi-tool integration. Le Blanc and Lam [8] independently study the early phase of this project; our paper covers the full campaign through March 2026, including automated harness generation and quantitative results across all three crates.

8 Conclusion

This paper reports on the first large-scale, community-driven effort to verify the Rust standard library. The campaign integrated four verification tools into continuous integration, produced 16,748 automatic proof harnesses of which 11,970 were verified against Kani's supported classes of undefined behavior, and established 989 contract-verified proofs across automatic and manual harnesses. Along the way, the effort uncovered specification inconsistencies, missing safety annotations, and documentation errors, and contributed to the adoption of function contracts as an experimental Rust language feature.

Significant challenges remain: verifying generic functions, modeling compiler intrinsics, reasoning about concurrency under relaxed memory, and keeping proofs synchronized with an upstream repository that ships a new release every six weeks; scaling this process will require upstreaming contracts and proofs so that proof maintenance becomes part of the standard development workflow. These are problems where advances in deductive verification, abstract interpretation, type-theoretic reasoning, and dataflow analysis could each make a substantial contribution. Our experience also suggests that the Rust ecosystem will need a layered specification approach, with a shared contract syntax complemented by tool-specific annotation layers, rather than a single universal specification language.

Rust is now deployed in operating system kernels, browser engines, cryptographic libraries, and safety-critical embedded systems. By verifying foundational libraries like the standard library, we can help ensure that developers are building on solid foundations. The tools, infrastructure, and community assembled by this project provide a concrete starting point toward that goal.

Acknowledgments. We thank the Rust Foundation and all open-source contributors for their support. An AI assistant was used during the preparation of this manuscript for prose editing and bibliographic verification. All technical content, experimental data, and scientific claims are the sole work of the authors, who reviewed and approved the final text.

References

1. Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., Sapin, S.: Engineering the servo web browser engine using rust. In: Proceedings of the 38th International Conference on Software Engineering Companion. p. 81–89. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2889160.2889229
2. Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The prusti project: Formal verification for rust. In: NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings. p. 88–108. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-031-06773-0_5
3. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe rust? Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428204
4. Ayoun, S.E., Denis, X., Maksimović, P., Gardner, P.: A hybrid approach to semi-automated rust verification. Proc. ACM Program. Lang. **9**(PLDI) (Jun 2025). https://doi.org/10.1145/3729289
5. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hriţcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., Maillard, K., Parno, B., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Zanella-Béguelin, S.: Project Everest: Perspectives from developing industrial-grade high-assurance software. ACM Transactions on Programming Languages and Systems (2025), https://project-everest.github.io/assets/everest-perspectives-2025.pdf, to appear
6. Bhargavan, K., Buyse, M., Franceschino, L., Hansen, L.L., Kiefer, F., Schneider-Bensch, J., Spitters, B.: hax: Verifying security-critical Rust software using multiple provers. In: Verified Software. Theories, Tools and Experiments (VSTTE). pp. 96–119. Springer (2025). https://doi.org/10.1007/978-3-031-86695-1_7
7. Bhargavan, K., Hansen, L.L., Kiefer, F., Schneider-Bensch, J., Spitters, B.: Formal security and functional verification of cryptographic protocol implementations in rust. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. p. 2729–2743. CCS '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3719027.3765213
8. Blanc, A.L., Lam, P.: Lessons learned so far from a community effort to verify the Rust standard library (work-in-progress). https://arxiv.org/abs/2510.01072 (2025)
9. Boos, K., Liyanage, N., Ijaz, R., Zhong, L.: Theseus: an experiment in operating system structure and state management. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. OSDI'20, USENIX Association, USA (2020)
10. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the FSCQ file system. In: Proceedings of SOSP. pp. 18–37 (2015). https://doi.org/10.1145/2815400.2815402
11. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at amazon web services. Software: Practice and Experience **51**(4), 772–797 (2021). https://doi.org/https://doi.org/10.1002/spe.2949

12. Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. p. 11–20. ICSE-SEIP '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377813.3381347
13. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 430–446. Springer International Publishing, Cham (2018)
14. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from aws data centers. Form. Methods Syst. Des. **57**(1), 34–52 (Jul 2021). https://doi.org/10.1007/s10703-020-00344-2
15. Cui, M., Sun, S., Xu, H., Zhou, Y.: Is unsafe an achilles' heel? A comprehensive study of safety requirements in unsafe Rust programming. In: Proceedings of ICSE (2024). https://doi.org/10.1145/3597503.3639136
16. Cui, M., Sun, S., Xu, H., Zhou, Y.: 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. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639136
17. Dang, H.H., Jourdan, J.H., Kaiser, J.O., Dreyer, D.: RustBelt meets relaxed memory. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371102
18. Denis, X., Jourdan, J.H., Marché, C.: Creusot: A foundry for the deductive verification of Rust programs. In: Formal Methods and Software Engineering (ICFEM). LNCS, vol. 13478, pp. 90–105 (2022). https://doi.org/10.1007/978-3-031-17244-1_6
19. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: Esbmc 5.0: an industrial-strength c model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 888–891. ASE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3240481
20. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: RefinedRust: A type system for high-assurance verification of Rust programs. Proc. ACM Program. Lang. **8**(PLDI) (2024). https://doi.org/10.1145/3656422
21. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of C code without the pain. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 429–439 (2014). https://doi.org/10.1145/2594291.2594296
22. Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proceedings of OSDI. pp. 653–669 (2016)
23. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 165–181. USENIX Association (2014)
24. Huang, L., Ebersold, S., Kogtenkov, A., Meyer, B., Liu, Y.: Lessons from formally verified deployed software systems. ACM Comput. Surv. **58**(8) (Feb 2026). https://doi.org/10.1145/3785652

25. Jacobs, B., Fasse, J.: An approach for modularly verifying the core of Rust's atomic reference counting algorithm against the (Y)C20 memory consistency model. Journal of Object Technology **25**(3) (2025). https://doi.org/10.5381/jot.2025.25.3.a5
26. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: a powerful, sound, predictable, fast verifier for c and java. In: Proceedings of the Third International Conference on NASA Formal Methods. p. 41–55. NFM'11, Springer-Verlag, Berlin, Heidelberg (2011)
27. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: an aliasing model for rust. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371109
28. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158154
29. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Safe systems programming in rust. Commun. ACM **64**(4), 144–152 (Mar 2021). https://doi.org/10.1145/3418295
30. Jung, R., Kimock, B., Poveda, C., Sánchez Muñoz, E., Scherer, O., Wang, Q.: Miri: Practical undefined behavior detection for Rust. Proc. ACM Program. Lang. **10**(POPL) (2026). https://doi.org/10.1145/3776690
31. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of SOSP. pp. 207–220 (2009). https://doi.org/10.1145/1629575.1629596
32. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying Rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA2) (2023). https://doi.org/10.1145/3586037
33. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: Liquid types for rust. Proc. ACM Program. Lang. **7**(PLDI) (Jun 2023). https://doi.org/10.1145/3591283
34. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
35. Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P., Levis, P.: Multiprogramming a 64kb computer safely and efficiently. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 234–251. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132786
36. Rust Team: The rust reference: Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html (2024)
37. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in Rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 321–330. ICSE-SEIP '22, Association for Computing Machinery (2022). https://doi.org/10.1145/3510457.3513031
38. Villani, N., Hostert, J., Dreyer, D., Jung, R.: Tree borrows. Proc. ACM Program. Lang. **9**(PLDI) (2025). https://doi.org/10.1145/3735592
39. Wang, F., Song, F., Zhang, M., Zhu, X., Zhang, J.: KRust: A Formal Executable Semantics of Rust . In: 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 44–51. IEEE Computer Society, Los Alamitos, CA, USA (Aug 2018). https://doi.org/10.1109/TASE.2018.00014