

Kani A Model Checker for Rust

Rémi Delmas, Ziad Hassan,
Qinheping Hu, Rahul Kumar,
Felipe R. Monteiro, Thanh
Nguyen, Adrián Palacios
Amazon Web Services
Seattle, WA, USA

Celina Val
Amazon Web Services
Vancouver, BC, Canada

Michael Tautschnig
Amazon Web Services
Seattle, WA, USA
Queen Mary University
London, United Kingdom

Justus Adam*
Brown University
Providence, RI, USA

Daniel Schwartz-Narbonne*
Datadog
New York, NY, USA

Carolyn Zech*
Massachusetts Institute of Technology
Cambridge, MA, USA

Abstract

Rust’s ownership type system prevents memory errors in safe code, but certain desirable properties remain orthogonal to compilation: the soundness of `unsafe` operations (e.g., raw pointer dereferences), functional correctness, and absence of runtime panics. We present Kani, an open-source model checker for Rust that pushes bounded model checking beyond bug-finding to provide correctness guarantees for these properties. Kani compiles proof harnesses from Rust’s Mid-level Intermediate Representation (MIR) into CBMC’s bit-precise verification engine, automatically checking a comprehensive set of safety properties with no user annotation. To extend verification from bounded to unbounded, Kani provides a specification language comprising function contracts, loop contracts, quantifiers, and function stubbing. We demonstrate feasibility through case studies on industrial Rust projects, where contracts upgraded verification from panic-freedom to functional correctness, uncovering six previously unknown bugs. Kani operates at scale in production CI, with over 16,000 harnesses verified per code change in the Rust standard library verification campaign.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; **Formal software verification**.

Keywords

model checking, Rust, formal verification, specification language

ACM Reference Format:

Rémi Delmas, Ziad Hassan, Qinheping Hu, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Adrián Palacios, Celina Val, Michael Tautschnig, Justus Adam, Daniel Schwartz-Narbonne, and Carolyn Zech. 2026. Kani: A Model Checker for Rust. In *Proceedings of 39th IEEE/ACM International Conference on Automated Software Engineering ASE ’26*. ACM, New York, NY, USA, 12 pages.

1 Introduction

Rust has become the language of choice for safety-critical systems software, from embedded operating systems [12, 36] and cryptographic protocols [9] to cloud infrastructure [1]. Its ownership type

system statically prevents data races and many classes of memory errors [29], but there are three classes of correctness properties that the compiler cannot prove: (1) the correctness of `unsafe` operations (e.g., raw pointer dereferences, calls to `unsafe` functions, access to mutable statics, `unsafe` trait implementations, and union field access) where the compiler still enforces borrow checking and type safety but the developer assumes responsibility for the additional safety invariants these operations require [5, 18]; (2) functional correctness properties such as algorithmic correctness and protocol conformance; and (3) absence of runtime panics from operations like `unwrap()`, integer overflow, and out-of-bounds access, in contexts where panics are undesirable. Testing and fuzzing can expose some of these failures, but they explore only a finite sample of the input space and provide no completeness guarantee.

Deductive verification tools for Rust, including Prusti [4], Creusot [22], and Verus [34], can prove rich functional properties. However, they require substantial proof engineering (e.g., separation logic or ghost state) that limits adoption outside specialist teams [26]. At the other end of the spectrum, dynamic tools like Miri [30] detect undefined behavior at runtime but cannot prove its absence. There is a gap between these extremes: developers need a verification approach that can start with high automation and low annotation cost, then scale incrementally toward stronger correctness guarantees as the verification need grows.

Bounded model checking fills the automated end of this gap. A bounded model checker encodes default safety properties (e.g., arithmetic overflow, division by zero, null dereferences, assertion violations) and checks them exhaustively up to a bound, requiring minimal manual proof construction. This makes it an effective entry point for verification: developers write proof harnesses that resemble unit tests, and the tool proves properties over all inputs within the bound. The limitation is that bounded analysis alone cannot guarantee correctness beyond the unwinding depth. To push model checking from bug-finding toward correctness proofs, the technique must be extended with specification constructs that enable unbounded reasoning while preserving the low annotation overhead that makes bounded checking accessible.

This paper presents Kani, an open-source model checker for Rust that realizes this progression from bounded analysis to unbounded correctness guarantees. Kani builds on the code-level model checking methodology [13, 14], which demonstrated that bounded model checking with CBMC [32] can be integrated into continuous development workflows for industrial C codebases. Kani extends this

*All work done while at Amazon Web Services.

methodology to Rust, operating on the Mid-level Intermediate Representation (MIR) to preserve Rust-specific type invariants, and adds a specification language comprising function contracts, loop contracts, quantifiers, and function stubbing. These constructs allow developers to incrementally annotate their code: Kani first proves default safety properties with no annotation, then, as contracts are added, extends the same verification engine to unbounded functional correctness proofs. We demonstrate the feasibility of this approach through a case study on the Hifitime time-management library, where adding contracts upgraded guarantees from panic-freedom to functional correctness at low specification overhead, with AI-assisted specification drafting via an AI coding assistant. Kani is deployed in production CI at scale: proof harnesses run on every code change in Firecracker [1], the virtual machine monitor behind AWS Lambda and AWS Fargate; in s2n-quic, Amazon's Rust implementation of the IETF QUIC transport protocol; in Hifitime [40], a time-management library used in aerospace applications; and in the Rust standard library verification campaign [17], where over 16,000 harnesses are verified per code change.

We make the following contributions:

- (1) We evaluate Kani on multiple industrial Rust projects, where it uncovered eleven bugs missed by testing and fuzzing. Through a detailed case study on the Hifitime library, we show that contract-based verification extends Kani's default safety checks to unbounded functional correctness proofs at low specification overhead (§5).
- (2) We formalize Kani's specification language for function contracts, loop contracts, quantifiers, and function stubbing, grounding the semantics in Floyd-Hoare logic and connecting them to the underlying bounded model checking engine (§4).
- (3) We present Kani's architecture, its MIR-level design, and its integration with the Rust toolchain (§3).

2 Kani by Example

We illustrate Kani's verification workflow using Euclid's greatest common divisor (GCD) algorithm and its use in Firecracker [1], an open-source virtual machine monitor that runs workloads in lightweight microVMs. Firecracker powers AWS Lambda and AWS Fargate, making the correctness of its Rust codebase a security-critical concern. The GCD function is used in Firecracker's rate limiter to simplify token-bucket refill ratios, and its iterative loop makes it a natural target for both bounded and unbounded verification.

2.1 Bounded Verification

Consider the iterative GCD implementation from Firecracker's `rate_limiter` module:

```
fn gcd(x: u64, y: u64) -> u64 {
    let mut a = x;
    let mut b = y;
    while b != 0 {
        let t = b;
        b = a % b;
        a = t;
    }
    a
}
```

A developer can write a Kani *proof harness*, analogous to a unit test but over all possible inputs, to verify that `gcd` returns a common divisor:

```
#[kani::proof]
#[kani::unwind(94)]
fn check_gcd() {
    let x: u64 = kani::any();
    let y: u64 = kani::any();
    kani::assume(x > 0 && y > 0);
    let d = gcd(x, y);
    assert(d != 0 && x % d == 0 && y % d == 0);
}
```

The call `kani::any()` produces a nondeterministic value of the given type, and `kani::assume` constrains the input space. A note of caution: an incorrect assumption (e.g., `assume(false)`) makes the proof vacuously true, so assumptions must be reviewed as carefully as the code itself. Kani unwinds the loop, converts the result to SSA form, and encodes all assertions as a propositional formula (§4.1) that is checked using a SAT solver.

The `#[kani::unwind(94)]` annotation sets the loop unwinding bound. For 64-bit inputs, the worst-case number of iterations of Euclid's algorithm is 93 (the largest Fibonacci number below 2^{64} is F_{93}); the bound is set conservatively to 94. If the bound is insufficient, Kani reports an *unwinding assertion failure*, alerting the developer that the verification result is inconclusive beyond the given depth. Even with a sufficient bound, the resulting formula is very large: in our experiments, bounded verification of `gcd` over the full u64 range did not terminate within a one-hour timeout. Moreover, the bound is specific to 64-bit integers; changing the input type requires recomputing it.

2.2 Unbounded Verification with Contracts

We annotate `gcd` with a function contract (precondition and postcondition) and a loop contract (invariant and decreases clause). The loop contract eliminates the dependence on the unwinding bound by abstracting the loop via an inductive invariant:

```
#[kani::requires(x > 0 && y > 0)]
#[kani::ensures(|&result| result > 0)]
fn gcd(x: u64, y: u64) -> u64 {
    let mut a = x;
    let mut b = y;
    #[kani::loop_invariant(
        a > 0
        && kani::forall(|d: u64|
            in (1, a.saturating_add(1)) {
                d == 0
                || ((x % d == 0 && y % d == 0)
                    == (a % d == 0 && b % d == 0))
            })
    ]
    #[kani::loop_decreases(b)]
    while b != 0 {
        let t = b;
        b = a % b;
        a = t;
    }
    a
}
```

The `requires` clause states the precondition: both inputs must be positive. The `ensures` clause states the postcondition: the result is positive. Inside the loop, `loop_invariant` asserts two properties across all iterations: (1) `a` remains positive, and (2) for every candidate divisor `d`, `d` divides the original inputs `(x, y)` iff it divides the

current values (a, b) , i.e., the set of common divisors is preserved. The `saturating_add` avoids overflow when computing the upper bound of the quantifier range. The `loop_decreases` clause specifies that b is a well-founded decreasing measure (since $a \bmod b < b$ on each iteration), proving termination (§4.4).

The natural postcondition is the full divisibility property ($x \% \text{result} == 0 \ \&\& \ y \% \text{result} == 0$), which the invariant implies at loop exit; however, Z3 cannot automatically instantiate the quantifier at $d = \text{result}$ over nonlinear 64-bit bitvector remainder (§4.5), so we use the weaker `result > 0`. Verification passes all 202 checks in 0.54 s with Z3, exercising the full pipeline: quantified loop invariant, termination proof, and all automatically generated safety checks.

This invariant is nontrivial: it requires a universally quantified statement over nonlinear arithmetic. In practice, many verification tasks require only simple contracts, which Kani checks automatically with no annotations. The GCD example demonstrates the expressiveness of Kani’s specification language when functional correctness is desired.

Verifying the contract. A dedicated `proof_for_contract` harness sets nondeterministic inputs via `kani::any()` and calls `gcd`; Kani assumes the precondition, executes the function, and asserts the postcondition. The `#[kani::solver(z3)]` attribute selects the Z3 SMT solver, required for the quantified invariant. The loop invariant is verified inductively (§4.4): the loop body executes exactly twice in a single BMC query, regardless of input size.

What failure looks like. When a postcondition is violated, Kani reports the contract expression and source location. For example, a sign error in Hifitime’s `total_nanoseconds()` (§5.2) produced:

```
Check 1: ...total_nanoseconds::(closure#2)
- Status: FAILURE
- Description: "|result| { *result ==
  i128::from(self.centuries) *
  i128::from(NPC) +
  i128::from(self.nanoseconds) }"
VERIFICATION:- FAILED (0.26s)
```

Using the verified contract. Once verified, the contract serves as a sound abstraction for compositional reasoning. In Firecracker, `TokenBucket::new` calls `gcd` to simplify token-bucket refill ratios. Using `#[kani::stub_verified(gcd)]`, the harness replaces `gcd` with its contract: the precondition is asserted at the call site, and a nondeterministic return value satisfying the postcondition (`result > 0`) is assumed. No loop is unrolled, and verification completes in seconds over all 64-bit inputs using a sound abstraction.

3 Architecture

Kani is an open-source verification tool that integrates with the standard Rust toolchain. It can be invoked via `cargo kani` on Cargo¹ packages (analogous to `cargo test`) or via `kani` on individual crates. Internally, Kani reuses the entire `rustc` frontend and replaces the code-generation backend: instead of emitting machine code via LLVM, it translates Rust programs into GOTO programs [31] that CBMC can model-check. A key design decision is to operate on Rust’s Mid-level Intermediate Representation (MIR)

rather than LLVM IR. MIR preserves monomorphized type information, enum discriminant layouts, and Rust’s validity invariants (e.g., that a `bool` is 0 or 1, that a `char` is a valid Unicode scalar) in a form that LLVM IR discards. This enables Kani to automatically check Rust-specific safety properties and to reason about dynamic trait objects and closures, which are erased at the LLVM level [43]. As shown in Figure 1, the pipeline flows from Rust source through the Kani compiler to CBMC and a SAT/SMT solver, with the *kani-driver* orchestrating each stage.

Compilation. A verification session starts with `cargo kani`, which invokes *kani-driver*, the orchestrator (green box in Figure 1). The driver calls *kani-compiler*, a `rustc` plugin that hooks into the compiler via the `rustc_private` interface after MIR generation. The standard `rustc` frontend performs parsing, name resolution, type checking, trait resolution, and monomorphization, producing typed MIR. Kani then applies MIR-to-MIR transformations: reachability analysis, contract and loop-contract instrumentation (§4), stubbing, and safety-check insertion.

Code generation. The transformed MIR is lowered to a GOTO program by the *codegen* backend, which translates MIR statements, `rvalues`, `places`, and `types` into GOTO expressions using the *cprover-bindings* library available in CBMC. Calls to the Kani API (e.g., `kani::any()`, `kani::assume()`, `kani::assert()`, `forall`, or `exists`) are intercepted by *codegen* hooks and translated into their CBMC counterparts: nondeterministic values, assumptions, assertions, and quantified expressions. Quantifier bodies are inlined to produce single CBMC-compliant expressions (§4.5). The resulting GOTO program is serialized to CBMC’s binary IRep format.

Verification. The driver optionally invokes *goto-instrument* to apply Dynamic Frame Condition Checking² (DFCC) for function and loop contracts, instrumenting write-set enforcement. It then invokes CBMC, which performs symbolic execution of the GOTO program, unwinds loops, converts the result to SSA form, and encodes all assertions as a propositional formula (§4.1), which it dispatches to a solver. Kani supports multiple solver backends: MiniSat [23] (default), Kissat and CaDiCaL [11] for SAT, and Z3 [21], `cvc5` [7], and Bitwuzla [38] for SMT. CBMC returns a verdict per property (e.g., assertion, overflow check, memory safety check), which the driver maps back to Rust source locations and presents to the user.

Default properties. Beyond user-specified assertions, Kani automatically verifies a comprehensive set of properties³ without any annotation from the developer. These fall into three categories: *absence of undefined behavior* (i.e., invalid or dangling pointer dereferences, misaligned casts, invalid enum discriminants), *absence of runtime panics* (e.g., arithmetic overflow, division by zero, undefined shift, out-of-bounds indexing, `unwrap()` on `None`), and *resource properties* (e.g., float-to-integer cast finiteness). The Kani compiler instruments the code during MIR transformations and code generation with Rust-semantic checks; CBMC contributes additional C-level memory safety checks. Two optional MIR passes provide deeper checks: the *valid-value pass* asserts that unsafe type

¹<https://doc.rust-lang.org/cargo/>

²<https://diffblue.github.io/cbmc/contracts-dev-spec-dfcc.html>

³Full list at <https://github.com/model-checking/kani>.

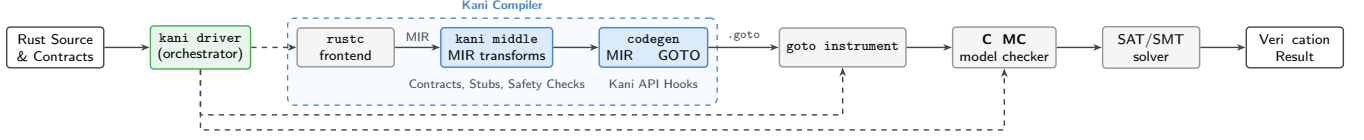


Figure 1: Kani verification pipeline. Blue boxes are Kani components; gray boxes are external tools; green is the orchestrator. White boxes denote input/output. Dashed arrows indicate orchestration control flow.

coercions produce valid inhabitants, and the *uninit-memory pass* uses shadow memory to verify that every dereference accesses initialized memory. Both are opt-in due to solver overhead.

4 Specification Language and Verification Methodology

Kani’s verification model follows a deliberate progression: bounded model checking provides push-button safety analysis for any Rust program; loop and function contracts extend this to *unbounded* verification when loops or recursion make a finite bound insufficient. This section formalizes that progression, beginning with the bounded foundation, then showing how each specification construct extends it. Rust’s ownership model introduces subtleties, particularly around type invariants and memory havocking, that do not arise in C-level model checking; we identify these precisely and explain how Kani addresses them. Throughout, we ground the formalism in the GCD-TokenBucket example of Section 2.

4.1 Bounded Model Checking Foundations

Kani’s verification engine is CBMC [31, 32], a bit-precise bounded model checker (BMC). Unlike classical BMC formulations defined over explicit transition systems [10], CBMC operates directly on programs: given a loop $\text{while } G \{B\}$ with guard G and body B , CBMC syntactically unwinds it up to a bound k , inserting an *unwinding assertion* $\neg G$ after the last copy. The result is a loop-free program P_k . CBMC then converts P_k to Static Single Assignment (SSA) form and encodes all assertions as a propositional formula [13, 16]. If the formula is satisfiable, the satisfying assignment yields a concrete counterexample; if unsatisfiable, all assertions hold for every execution of P_k .

SSA encoding. CBMC converts P_k to SSA form, introducing a fresh variable version $x_{(i)}$ at each assignment point i and encoding branching with path-condition guards g_j . The result is a conjunction of guarded equations where e_j is the right-hand-side expression at assignment j :

$$\text{SSA}(P_k) \quad \bigwedge_j g_j \rightarrow x_n \quad e_j).$$

The propositional formula CBMC constructs is:

$$\psi_k \quad \text{SSA}(P_k) \wedge \bigvee_{p \in \text{Asserts}(P_k)} \neg \varphi_p \quad (1)$$

where $\text{Asserts}(P_k)$ includes user assertions, automatically inserted safety checks, and unwinding assertions.

Completeness. The bounded guarantee becomes practically complete when all unwinding assertions pass, meaning P_k covers all

reachable paths. For programs with input-independent loop bounds this is tractable: in our running example, Euclid’s algorithm iterates at most 93 times, so $\#[\text{kani} : \text{unwind}(94)]$ is a sound, complete bound, but the unwound formula $O(k \cdot |body|)$ SSA equations over 64-bit bitvectors exceeds practical solver limits. For programs with input-dependent bounds, no tractable k may suffice. Function and loop contracts (§4.4) address this gap by replacing exhaustive unrolling with inductive reasoning, achieving unbounded guarantees with a single BMC query.

Kani compiles Rust’s MIR into CBMC’s GOTO program representation, inheriting bit-precise arithmetic and memory reasoning. In the remainder of this section, we use σ for program states and standard Hoare-logic notation. The specification constructs below extend this foundation.

4.2 Specification Language

Kani’s specification language extends Rust boolean expressions with contract annotations. All specification expressions must be deterministic and side-effect free (i.e., no I/O, mutation, or heap allocation), a restriction enforced by a MIR traversal at compile time.

Syntax. The grammar of contract annotations is:

<i>func-spec</i>	::	$\text{requires}(P) \mid \text{ensures}(r Q) \mid \text{modifies}(e)$
<i>loop-spec</i>	::	$\text{loop_invariant}(I) \mid \text{loop_modifies}(e) \mid \text{loop_decreases}(d)$
P, Q, I	::	$\text{rust-expr}_{\text{bool}} \mid P \wedge Q \mid \text{forall}(x:\tau \text{ in } (l, h) P) \mid \text{exists}(x:\tau \text{ in } (l, h) P)$
Q	::	$\dots \mid \text{old}(e) \quad (\text{ensures only})$
e	::	$\text{rust-expr}_{\text{pure}}$

Multiple *requires* and *ensures* clauses are implicitly conjoined. The pseudo-function $\text{old}(e)$ evaluates e in the pre-state and is valid only within *ensures* clauses; quantifiers range over a bounded domain $[l, h)$ (inclusive lower bound, exclusive upper bound) or, if bounds are omitted, over all values of type τ (with backend implications discussed in Section 4.5).

Validity invariants and safety invariants. A Rust-specific challenge is that types carry two classes of invariant. A *validity invariant* $\text{vInv}(\tau)$ must hold at all times (e.g., $\text{bool} \in \{0, 1\}$, valid enum discriminants); the compiler assumes it unconditionally. A *safety invariant* $\text{sInv}(\tau)$ is a semantic property that safe code may assume but the type system does not enforce (e.g., Duration’s nanosecond field $\leq 10^9$). Kani’s $\text{kani}::\text{any}::\text{<T>}()$ respects $\text{vInv}(\tau)$ by construction but does not automatically enforce $\text{sInv}(\tau)$. The Invariant trait allows types to express $\text{sInv}(\tau)$ programmatically; when a *requires* clause includes an *invariant* check, inputs are

constrained accordingly. We write $Arb(\tau)$ for the set of values producible by `kani::any::<T>()`.

Semantic domains. We write $\in$ for program states (mappings from locations to values), $f(\bar{x})$ for a function with parameters $\bar{x}$ and body B_f , and $Mod(f) \subseteq Loc$ for the write set of f . We write $[L \leftarrow havoc_\tau]$ for the state obtained by *havocking* L : replacing every location in L with a nondeterministic value drawn from $Arb(\tau)$, thereby preserving $vInv(\tau)$ while leaving $sInv(\tau)$ unconstrained unless the precondition specifies otherwise.

4.3 Function Contracts

A function contract is a triple $\langle Pre, Post, Mod \rangle$ where $Pre(\bar{x})$ is the precondition, $Post(\bar{x}, r)$ the postcondition over the return value r , and Mod the write set. All contract expressions must be deterministic and side-effect free (enforced at compile time). In the postcondition, bare parameter names refer to their *post-state* values; the `old(e)` construct (described below) is required to reference pre-state values. For parameters passed by value, the post-state and pre-state values coincide (the caller’s copy is unaffected); for parameters passed by mutable reference (`&mut T`), the post-state value reflects modifications made through the reference. The contract establishes the Hoare triple:

$$\{ Pre(\bar{x}) \} r \leftarrow f(\bar{x}) \{ Post(\bar{x}, r) \wedge frame(\bar{}, \bar{}', Mod) \} \quad (2)$$

where $frame(\bar{}, \bar{}', Mod)$ asserts that every location outside Mod is unchanged between the pre-state and post-state $\bar{}'$. In our running example, the GCD contract has $Pre \equiv (x > 0 \wedge y > 0)$ and $Post \equiv (r > 0)$; the write set is empty since `gcd` is a pure function. The loop invariant establishes the stronger property that the set of common divisors is preserved, but the postcondition is simplified to avoid intractable quantifier instantiation over nonlinear bitvector arithmetic; see Section 2. In practice, contract precision involves a trade-off: a precondition that is too strong prevents the contract from being applied at call sites that do not establish the full condition, while a postcondition that is too weak may leave call-site verification unable to prove the desired property. Developers must balance precision against annotation effort.

Contract checking. A contract is verified in a harness annotated with `#[kani::proof_for_contract(f)]`. Kani instruments the harness to establish (2):

- (1) construct arbitrary inputs $\bar{x}$ via $Arb(\tau_i)$, respecting Rust’s ownership invariants (e.g., no two `&mut T` arguments alias the same memory);
- (2) assume $Pre(\bar{x})$;
- (3) symbolically execute $r \leftarrow f(\bar{x})$;
- (4) assert $Post(\bar{x}, r)$; and
- (5) verify $writes(f) \subseteq Mod$ via CBMC’s frame condition checker.

The instrumented harness is discharged by BMC (1): CBMC searches for an execution that satisfies the precondition but violates the postcondition or frame condition. If loop contracts are also present, the loop body executes only twice in a single BMC query (c.f. Section 4.4), decoupling verification time from iteration count.

Contract replacement. Once (2) is established, the contract soundly replaces f at call sites via `#[kani::stub_verified(f)]`,

applying a *contract abstraction rule* that combines the verified contract with frame reasoning:

$$\frac{\{ Pre \} r \leftarrow f(\bar{x}) \{ Post \wedge frame \}}{\{ P \wedge Pre \} r \leftarrow f(\bar{x}) \{ P|_{\overline{Mod}} \wedge Post \}} \quad (3)$$

where P is the caller’s context and $P|_{\overline{Mod}}$ retains the conjuncts of P that do not depend on locations in Mod ; conjuncts that do are discarded as part of the over-approximation. The rule is admissible, justified by the BMC-verified premise: the replacement body *havocs* Mod and then assumes $Post$, which narrows the nondeterministic values to those satisfying the postcondition. No computation of f is performed, so f ’s internal loops need not be unrolled at call sites. In the TokenBucket verification of Section 2, this reduction drops verification of the caller from hours (i.e., the time required to unroll all 64-bit GCD iterations in context) to seconds, while providing an unbounded guarantee over all 64-bit inputs.

History expressions. The `old(e)` construct captures the pre-state value of e for two-state postconditions, essential when a function receives `&mut T`. For example, `ensures(|r| *v == old(*v) + 1)` asserts that the referent was incremented by exactly one. Kani rewrites `ensures(|r| Q(old(e),  $\bar{x}, r$ ))` into $\{v \leftarrow e\} r \leftarrow f(\bar{x}) \{Q(v, \bar{x}, r)\}$, where $\bar{}$ is the pre-state.

Write sets and havocking. For functions taking `&mut T`, Kani infers Mod from the signature: all memory reachable through mutable references. The *havoc* uses `kani::any()` (preserving $vInv(\tau)$), and the `modifies` macro converts references to raw pointers *after* borrow checking via `untracked_deref`. Explicit `modifies` clauses narrow the inferred set when the reachable state is too large, improving solver completeness. For raw pointers, `modifies` is required: Kani cannot infer which memory is reachable through a raw pointer, so without an explicit clause the write set defaults to empty and any write triggers a DFCC verification failure. The frame condition in (2) assumes that locations inside and outside Mod do not alias. Rust’s borrow checker guarantees this for safe code; for unsafe code, the user must ensure Mod is closed under aliasing.

Recursive functions. Contracts support unbounded verification of *self-recursive* functions via induction on call depth, implementing the standard proof rule [3]:

$$\frac{\{ Pre \} f(\bar{x}) \{ Post \} \vdash \{ Pre \} B_f \{ Post \}}{\vdash \{ Pre \} f(\bar{x}) \{ Post \}} \quad (4)$$

The premise assumes the contract holds for recursive calls and verifies the body under that assumption. Rule (4) is sound for partial correctness unconditionally, and for total correctness provided the recursion terminates under a well-founded measure. *Kani does not currently verify termination for recursive functions*; termination remains the user’s responsibility. Mutual recursion ($f \rightarrow g \rightarrow f$) is not supported; Kani emits a compile-time error when it detects a contracted function involved in a mutually recursive cycle.

Soundness and panic semantics. Kani interprets Hoare triples under *partial correctness with panic safety*: $\{P\} S \{Q\}$ means that if P holds and S terminates normally (without panicking), then Q holds. Panics (from `assert`, `unwrap()`, arithmetic overflow, or out-of-bounds access) are translated into assertion violations and are therefore checked as part of the verification. A successful Kani run

guarantees both the absence of panics and the postcondition, for all inputs satisfying the precondition, up to the following assumptions: harness coverage of reachable inputs, aliasing closure of *Mod* for unsafe code, user-guaranteed termination, sequential execution (no threads or *async*), no Stacked Borrows [28] / Tree Borrows [44] pointer-aliasing modeling (the primary class of UB in unsafe Rust), and FFI calls executing outside CBMC's memory model unless stubbed (§4.6). The last two are the most significant in practice. Miri [30] checks aliasing violations dynamically, making it a natural complement.

4.4 Loop Contracts

For programs where a completeness threshold exists but is too costly to reach, such as the 64-bit GCD loop, loop contracts provide an alternative path to unbounded verification. They replace exhaustive loop unrolling with an application of the Floyd–Hoare *WHILE* rule.

A loop contract consists of three clauses:

- `loop_invariant(I)`,
- `loop_modifies(W)`, and
- `loop_decreases(d)`.

For a loop `while G {B}` with post-loop continuation *C*, the proof obligation is:

$$\frac{\{I \wedge G\} B \{I\} \quad \text{writes}(B) \subseteq W}{\{I\} \text{ while } G \{B\} \{I \wedge \neg G \wedge \text{frame}(_, _, W)\}} \quad (5)$$

Kani verifies rule (5) through a two-level transformation that produces a single BMC query covering all three proof obligations.

At the MIR level, Kani restructures the loop CFG (desugaring for loops and hoisting variables for nesting). At the GOTO level, CBMC's DFCC pass rewrites the loop into four blocks executed in a single BMC run: Prehead (snapshots and initialization), Head (guard evaluation and body execution), Step (base-case assertion, havoc *W*, assume *I*), and Exit (frame assumption). The control flow is Prehead → Head (base case) → Step → Head (inductive step) → cut. The loop body executes exactly twice, making verification time independent of the iteration count.

Two aspects of this transformation are non-obvious. First, the Step block sits *between* the two passes through Head: if the loop guard is false on entry, Step is never reached and the havoc never fires, correctly preserving the program state for zero-iteration loops. Second, the inductive step routes through the actual compiled guard in Head rather than synthetically assuming it, so the body executes under $I \wedge G$ with all safety checks on the guard expression intact.

Once an inner loop's contract is verified, its abstraction (i.e., havoc W_{in} , assume $I_{in} \wedge \neg G_{in}$) over-approximates its concrete behavior. The outer loop's inductive step therefore remains sound against the abstracted inner body, provided $W_{in} \subseteq W_{out}$. Function-level composition follows the same argument: all loop abstractions are applied before checking the function's postcondition.

In our running example, the GCD loop invariant asserts that $a > 0$ and that the set of common divisors of (x, y) equals the set of common divisors of (a, b) at every iteration. The base case holds trivially ($(a, b) = (x, y)$ on entry); the inductive step follows because Euclid's step preserves the set of common divisors; and the abstraction delivers $\neg G \equiv (b = 0)$, from which the postcondition $a > 0$ follows directly from the invariant's first conjunct. The

invariant also implies $a \mid x \wedge a \mid y$ at this point, but verifying that implication automatically requires quantifier instantiation that current SMT solvers cannot perform; see Section 2.

Rust's for loops desugar into iterator protocol calls with opaque internal state that CBMC cannot inspect. Kani rewrites annotated for loops into while loops over a simplified `KaniIntoIter` abstraction that exposes only the current index and the iteration bounds. This allows invariants to be stated over the loop index rather than private iterator fields. The rewrite is exact for range-based for loops; for custom iterators, the user must provide loop-level contracts that reason about the iterator's abstraction.

Termination. Without `loop_decreases`, rule (5) establishes only partial correctness: the invariant is checked at entry and preserved by each iteration, but non-termination is not ruled out. Kani emits a warning when `loop_decreases` is absent. A `loop_decreases(d)` clause adds the termination proof obligation:

$$\{I \wedge G \wedge d \quad d_0 \wedge d_0 \geq 0\} B \{0 \leq d \quad d_0\},$$

upgrading the result to total correctness. The measure *d* is restricted to non-negative integer expressions; more general well-founded orders (e.g., lexicographic tuples) are not yet supported. In our example, `loop_decreases(b)` is discharged because each iteration replaces *b* with $a \bmod b$, and $b \geq 0$ is preserved. Currently, the underlying CBMC engine supports only simple integer expressions in decreases clauses; struct field projections and tuple-based lexicographic measures are not yet supported.

4.5 Quantifiers

Kani provides first-order quantifiers as procedural macros:

```
kani::forall (|x: T in (lo, hi)| P(x))
kani::exists (|x: T in (lo, hi)| P(x))
```

These compile to CBMC-level quantified expressions. With SAT solvers (the default), CBMC eagerly expands quantifiers over the specified range, so bounds must be compile-time constants; Kani emits a warning when the range exceeds 1000 values. With SMT backends (`-solver z3`, `cvc5`, or `bitwuzla`), quantifiers are passed directly to the solver as first-order formulas, supporting runtime-valued bounds.

4.6 Function Stubbing

Function stubbing replaces a function's implementation with a handwritten mock for the duration of a specific harness, specified via the `#[kani::stub(f, g)]` attribute. The replacement is a MIR-to-MIR transformation: when the compiler requests the MIR for *f*, Kani returns the MIR for *g* instead. Kani validates signature compatibility (matching parameter types, return type, and generic arity) at compile time; without this check, a stub could silently operate on differently typed values, producing spurious proofs.

Stubbing serves three purposes: (1) replacing functions with features unsupported by the verification backend (inline assembly, FFI); (2) replacing expensive implementations with lightweight nondeterministic abstractions; and (3) environmental modeling, where system interactions such as clock reads or random number generation are replaced with unconstrained nondeterministic values.

Stubs versus verified contracts. Plain stubs are *not* checked against the original implementation: the user asserts that g is a sound model of f , and Kani trusts this claim. A verification result that relies on unverified stubs is therefore *conditional*: it is sound only if every stub faithfully over-approximates the behavior of the function it replaces. Verified contracts (Section 4.3), by contrast, machine-check the contract against the implementation before admitting the abstraction; subsequent uses of `stub_verified` are then provably sound by rule (3). The two mechanisms are complementary: contracts for formally established behavioral abstractions, stubs for environmental modeling where no implementation exists to verify against. In the `TokenBucket` example of Section 2, the `gcd` function uses a verified contract, while `Instant::now`, a system call whose semantics Kani cannot model, uses a plain stub that returns a nondeterministic `Instant`. Together, they reduce the verification of `TokenBucket::new` from an intractable loop-unrolling problem to a bounded check with an unbounded correctness guarantee.

5 Evaluation

We evaluate Kani through three research questions grounded in the concerns of industrial practitioners:

- RQ-I** Does Kani find real bugs that testing and fuzzing miss?
- RQ-II** Do function and loop contracts extend bounded verification to stronger, unbounded guarantees in practice?
- RQ-III** Is Kani sustainable in continuous integration across diverse projects over time?

Table 1: Subset of open-source projects using Kani.

Project	Domain	Harnesses	Runtime	Contracts
Firecracker [1]	Cloud Infrastructure	34	21 min	
s2n-quic [2]	Network Protocol	102	23 min	–
Hifitime [40]	Aerospace	153	42 min	
verify-rust-std [17]	Standard Library	16,748	69 min	
zerocopy [37]	Serialization	10	3 min	
lading [20]	Load Testing	22	2 min	–
x86_64 [41]	Hardware	6	1 min	–
rust-sel4 [42]	Microkernel	1	4 min	–

Table 1 lists representative open-source projects across multiple domains that use Kani in CI. Our evaluation focuses on a subset of these: Firecracker, the virtual-machine monitor behind AWS Lambda⁴ and AWS Fargate⁵; s2n-quic, Amazon’s implementation of the IETF QUIC transport protocol [33]; Cedar [19], an open-source authorization policy language and engine (which does not use Kani in its regular CI); Hifitime, a high-precision time management library used in the Moon landing mission Blue Ghost Mission 1⁶; and the Rust standard library verification campaign [17].

5.1 Bug-Finding Effectiveness

We highlight three cases that illuminate qualitatively different failure modes.

⁴<https://aws.amazon.com/lambda/>

⁵<https://aws.amazon.com/fargate/>

⁶<https://fireflyspace.com/missions/blue-ghost-mission-1/>

Table 2: Bugs found by Kani that testing and fuzzing missed.

Project	Bug type	Missed by
s2n-quic	<code>try_fit</code> assertion	Tests, fuzz (16M iter.)
s2n-quic	Pkt. number decode overflow	Tests only
Firecracker	Rate limiter rounding	Tests, fuzz
Firecracker	VirtIO panic from guest	Tests, fuzz
Cedar	<code>contains_at_least_two</code>	Tests, diff. testing
Hifitime	<code>normalize()</code> sign overflow	Tests
Hifitime	<code>i64::MIN.abs()</code> panic	Tests
Hifitime	Float multiply NaN loop	Tests
Hifitime	Epoch Eq/Ord inconsistency	Tests
Hifitime	Duration Eq/Ord zero-crossing	Tests
Hifitime	<code>is_gregorian_valid</code> overflow	Tests

s2n-quic: finding bugs at encoding boundaries. Amazon’s QUIC implementation, s2n-quic, uses Kani alongside the Bolero⁷ property-testing framework, allowing each harness to run as both a fuzz test and a Kani proof with a single attribute. The most striking result involves the `try_fit` function, which determines how many bytes of data to fit into a QUIC Stream frame given a packet’s remaining capacity, a subtle calculation because the variable-length integer encoding of the frame length increases at capacity boundaries. A differential Bolero harness compared the implementation against a reference model. Running the *libfuzzer* engine for over ten minutes (16,777,216 executions) found no failure. Kani found a failing assertion in 20 seconds. The failure lived near the boundary between one-byte and two-byte integer encodings, a sparse region of the input space that coverage-guided fuzzing has no gradient toward; Kani’s symbolic execution reached it directly.

A second s2n-quic case illustrates complementarity: a bug in `decode_packet_number` was found by Kani in 2.8 seconds and independently by fuzzing in under a minute. When both methods find the same bug, the Kani result is more actionable: the failing check identifies a specific named property rather than an unexplained crash, and Kani generates a concrete counterexample (via its *concrete playback*⁸ feature) that the developer can replay as a standard Rust test.

Firecracker: verifying properties invisible to testing. The Firecracker virtual machine monitor applies Kani to two security-critical components. For the I/O rate limiter, which uses a token-bucket algorithm where tokens accumulate over time and each I/O operation consumes tokens, the property of interest is that a microVM cannot exceed its configured I/O bandwidth in any one-second interval. This is inherently time-dependent: the number of tokens replenished depends on when `auto_replenish` is called relative to the system clock, making it impossible to test exhaustively. Kani resolves this by stubbing `libc::clock_gettime` with a nondeterministic stub that returns monotonically non-decreasing `Instant` values, turning the temporal dimension into a symbolic variable. The resulting harnesses found several bugs in the rate-limiter implementation, the most significant being a rounding error that allowed a guest to exceed its I/O budget by up to 0.01 in adversarially timed invocations. The error depends on the exact sub-millisecond timing of replenishment calls relative to the system

⁷<https://github.com/camshaft/bolero>

⁸<https://model-checking.github.io/kani/reference/experimental/concrete-playback.html>

clock, a condition that testing cannot control deterministically and that fuzzing cannot target without an explicit time model.

Firecracker also implements the VirtIO device emulation protocol [39], where shared memory between host and guest creates an adversarial attack surface: an untrusted guest can write arbitrary values into the shared descriptor and ring buffers. Kani verified conformance to Section 2.6.7.2 of the VirtIO specification using nondeterministic guest memory to represent this adversarial scenario. Kani found an additional bug: a guest could cause Firecracker to panic on boot by placing a VirtIO queue component’s starting address in the MMIO gap.

Cedar: a logic bug in policy evaluation. Cedar is an open-source authorization policy language and engine. A Kani experiment on Cedar’s evaluator found a bug in the `contains_at_least_two` function, which checks whether a string contains at least two occurrences of a substring. When called with a multibyte character as the search query, the function could slice a `&str` across a non-character boundary, resulting in a runtime panic. The bug was silent under all existing tests. Kani’s symbolic execution over all possible collection inputs produced a counterexample that violated the function’s intended invariant; the fix was merged⁹. This case is notable because Cedar already uses differential testing against a formal reference model, yet the bug survived into production code.

Answer to RQ-I: Kani found eleven bugs across four production Rust codebases that testing and fuzzing missed, including bugs that survived millions of fuzz iterations (`s2n-quic`), bugs in time-dependent code invisible to testing without a time oracle (Firecracker), and differential testing against a formal reference model (Cedar). Beyond bug-finding, the harnesses that detected these bugs also serve as machine-checked specifications that persist in CI, foreshadowing the stronger correctness guarantees explored in RQ-II.

5.2 Contract-Based Unbounded Verification in Practice

We present a detailed case study of Kani’s specification language applied to Hifitime, a high-precision time-management library used in aerospace and astrodynamics applications. Hifitime stores durations as a pair (`centuries: i16`, `nanos: u64`) where the nanosecond field must satisfy the normalization invariant `nanos ≤ NPC` (i.e., the number of nanoseconds in one century) except at the saturation extremes `Duration::MIN` and `Duration::MAX`. All arithmetic operations call `normalize()` to re-establish this invariant, making it the library’s central safety-critical function.

5.2.1 Phase I: Panic Freedom Harnesses. Hifitime’s original Kani integration, developed by the library’s author in 2022–2023 with no (substantial) prior formal verification experience, consisted of 11 harnesses checking only *panic-freedom*: that operations do not trigger arithmetic overflows, null dereferences, or out-of-bounds accesses. Despite the absence of postconditions, this baseline found 8 bug categories in a single pull request¹⁰, including overflow at

boundary values of `normalize()`, sign errors in `neg()`, and overflow in `PartialEq` and arithmetic operations near `Duration::MIN` and `Duration::MAX`, all of which survived 78 existing tests.

By 2024 the harness count had grown to 168 through a combination of manual and automated harness generation. This expansion revealed a fundamental scalability limit: of 158 automatically generated harnesses, 57 timed out at the 60-second CI budget. The root cause was that bounded model checking re-verified the `normalize()` loop inside every caller, with the loop’s nanosecond adjustment requiring many unrolling iterations in the worst case.

5.2.2 Phase II: Contract Based Functional Verification. To address both the scalability problem and the expressive gap, we upgraded the Hifitime verification using Kani’s function and loop contracts. This phase was conducted using an AI coding assistant, providing a concrete data point on AI-assisted formal specification.

The central idea is compositional verification via contracts. We expressed the normalization invariant (`nanos ≤ NPC`) as a post-condition on `normalize()`, enabling callers to use `stub_verified` (Equation 3) to replace the implementation with its contract abstraction i.e. eliminating the need to re-unroll the internal loop at each call site. Building on this, we annotated 84 functions with ensures postconditions and 41 with requires preconditions, converted 12 of the existing 168 panic-freedom harnesses into `proof_for_contract` harnesses that verify functional properties, and added 61 new `proof_for_contract` harnesses and 80 standalone proofs with explicit functional assertions. This compositional approach enabled us to go beyond panic-freedom and prove six new classes of functional properties, from normalization invariants and algebraic laws to encode–decode identities and specification consistency (see Table 3).

Table 3: Classes of functional properties proved with contracts in Hifitime Phase II.

Property class	Representative harnesses	Max time
Normalization	<code>normalize: nanos ≤ NPC</code> ; idempotence	0.2s
Algebraic laws	Commutativity of +; identity; $a + a = 0$	1.1s
Encode–decode	<code>to_parts/from_parts</code> ; epoch add/subtract	4.5s
Calendar validity	Gregorian ranges; leap-year rule	16.4s
Loop termination	<code>Mul<f64></code> ; <code>is_gregorian_valid</code> (4 invariants)	13.6s
Spec. consistency	<code>Eq/Ord</code> agreement	0.5s
Total	153 proofs 73 contract + 80 standalone	

For loops that require unbounded reasoning, Kani’s loop contracts (§4.4) establish guarantees independent of iteration count. As an example, the `Duration::Mul<f64>` precision loop iteratively scales a multiplier `new_val` (computed as $q \cdot 10^p$) to find the required decimal precision. We annotated it with a loop invariant and a decreases clause:

```
let mut p: i32 = 0;
let mut new_val: f64 = q;
...
#[kani::loop_invariant(p >= 0 && p <= 19)]
#[kani::loop_decreases(19i32.wrapping_sub(p))]
while new_val.is_finite()
    && (new_val.floor() - new_val).abs() >= f64::EPSILON
    && p < 19
{
    p += 1;
```

⁹More information available at <https://github.com/cedar-policy/cedar/pull/1037>.

¹⁰More information available at <https://github.com/nyx-space/hifitime/pull/192>.

```
new_val = q * ten.powi(p);
}
```

The invariant is inductive and the decreases clause proves termination, together establishing *total correctness*: the loop terminates with $p \in [0, 19]$ for all f64 inputs, with no assumptions on q . In isolation, Kani verifies the loop contract in 0.2 s; without it, CBMC unrolls 19 iterations and takes 91.8 s. In the full crate context, verification of `Mul<f64>` completes in under 70 s with the contract, compared to intractable without it.

The result is 153 active harnesses (up from 168 in Phase I), of which all prove functional correctness properties beyond panic-freedom. The most consequential change from Phase I is in the *class of guarantee*: Phase I proves that operations do not crash; Phase II proves that they compute correctly. The timeout problem that surfaced in 2024 (57 harnesses exceeding the CI budget) is addressed by contracts, which allow callers to reason about `normalize()` via its specification rather than re-verifying its internal loop.

Bugs found by contracts. As a side effect of establishing these functional properties, Kani uncovered six previously unknown bugs, properties that the original panic-freedom harnesses could not express.

i) `total_nanoseconds`) sign error. The implementation used subtraction instead of addition for durations spanning more than one century in the negative direction, producing incorrect values for any `Duration` with centuries $\neq 1$. The encode–decode identity (`from_total_nanoseconds(d.total_nanoseconds()) == d`) condition immediately falsified the postcondition, providing a concrete counterexample. The fix is a one-character source change. Notably, one existing integration test was *asserting the buggy behavior* and had to be corrected alongside the implementation. The postcondition was derived from the `Duration` type’s encoding invariant (documented in the struct definition), providing an independent specification against which both the implementation and the test were checked. The test oracle, having been written against the implementation’s output, encoded the same error.

ii) `i64::MIN.abs`) arithmetic panic. `Unit::Mul<i64>` called `.abs()` on `i64::MIN`, which panics because the result does not fit in `i64`. The normalization contract found this; the fix uses `.unsigned_abs()`.

iii) NaN in float multiply. When `new_val` overflows to infinity in the `Mul<f64>` precision loop, the loop invariant (§5.2) falsified on the overflow iteration. Two source-level guards were added.

iv) `Epoch::PartialEq` and `Ord` inconsistency. The method `Duration::PartialEq` deliberately treats opposite-sign values as equal (representing the same interval length regardless of direction), but `Epoch::PartialEq` delegated to this implementation. Meanwhile, `Epoch::Ord` used a lexicographic comparison that is sign-aware. The Rust standard library requires that `PartialEq` and `Ord` agree: $a == b$ must imply $a.cmp(b) == Ordering::Equal$. This invariant was violated for any pair of epochs differing in sign.

v) `Duration::PartialEq` and `Ord` inconsistency zero-crossing). `Duration::PartialEq` has a custom zero-crossing special case that treats opposite-sign durations as equal, but `Duration::Ord` is derived (lexicographic on fields) and does not. Two durations can therefore satisfy $a == b$ and $a < b$ simultaneously, violating Rust’s `Eq/Ord` contract. This bug was discovered

during the investigation of Bug 4: tracing the Epoch inconsistency to its root cause in `Duration`.

vi) `is_gregorian_valid` overflow. The Gregorian calendar validity check overflows at `year == i32::MAX` due to an unchecked addition in the leap-year computation. The contract on the calendar validation function detected this boundary case.¹¹

Bugs (iv) and (v) are the most semantically significant findings because they violate a requirement of the Rust standard library itself, not merely a domain-specific invariant. Bug (iv) was invisible to all existing tests because no test compared an epoch to its sign-negation, a conceptually unusual operation for a “point in time.” Kani’s symbolic execution generated this input automatically by exploring all possible Epoch pairs. The `contract_epoch_eq_symmetric` harness expressed the consistency requirement as a postcondition and immediately produced a counterexample. The fix rewrites both `Epoch::PartialEq` and `Epoch::Ord` to use `total_nanoseconds()` for a sign-aware scalar comparison. This case illustrates why exhaustive symbolic analysis over all inputs matters: the failing input lies in a region of the state space that no human tester would explore, yet it represents a real semantic violation.

AI-assisted formal specifications. The Phase II specifications were developed using an AI coding assistant with shell access to the codebase. Because Kani’s specification language is an extension of Rust itself (i.e., contracts are Rust boolean expressions; harnesses are Rust functions), the assistant could leverage its existing Rust knowledge to produce specifications without learning a separate formal language. The workflow was a tight loop: the assistant proposed contract annotations, wrote them into the source files, ran `cargo kani`, observed the result (i.e., success, failure, timeout, or compile error), and refined the specification in response. The human researcher directed which functions to target, provided domain context, and reviewed the proposed contracts for correctness.

Crucially, the specifications do not need to be trusted: Kani verifies every contract against the implementation. This was demonstrated concretely when three AI-generated contracts failed verification, and Kani caught all three: (1) an intractable postcondition that called the function under verification in its own ensures clause, doubling the SAT formula and causing a solver timeout (the assistant simplified it to a direct arithmetic expression); (2) an overly broad symbolic input (`kani::any::<Epoch>()`) that included the `TimeScale::ET` variant, pulling `sin()` into the SAT formula and causing a timeout (the assistant restricted the harness to `TimeScale::TAI`); and (3) a `const fn` incompatibility where Kani’s macro expansion generated non-`const` code, producing a compile error (the assistant switched to a standalone proof harness). These errors classify as one specification design error, one harness design error, and one Kani limitation. In all three cases, the assistant diagnosed the failure from Kani’s output and proposed a fix within the same session.

¹¹<https://github.com/nyx-space/hifitime/issues/475>

Answer to RQ-II: Contracts extend bounded panic-freedom guarantees to stronger functional properties with unbounded coverage. In Hifitime, 153 functional proofs covered six new classes of functional properties. Contracts enabled compositional reasoning and uncovered six bugs that panic-freedom checks could not express, including two violations of Rust’s own Eq/Ord contract.

5.3 Continuous Integration Sustainability

All projects listed in Table 1 run Kani in CI, and several have done so for years: Firecracker integrated Kani in 2022 [14], and s2n-quick and Hifitime followed shortly after. This sustained adoption builds on the code-level model checking methodology established by Chong et al. [13, 14] for C codebases at AWS, which Kani extends to Rust. The practical CI feasibility of Kani has been demonstrated at scale: the Rust standard library verification campaign runs 16,748 Autoharness-generated harnesses via parallelization and incremental caching, achieving a 3.97× speedup over a naïve sequential pipeline [17]. Across the projects in Table 1, CI runtimes range from under one minute (x86_64) to approximately 69 minutes (verify-rust-std), with most completing in under 25 minutes, which are practical budgets for pull-request workflows.

A key enabler of CI adoption is Kani’s cargo kani interface, which integrates identically with cargo test. In s2n-quick, the Bolero framework allows the same harness to run as both a fuzz test and a Kani proof via a single `#[cfg_attr(kani, kani::proof)]` attribute, adding Kani’s exhaustive analysis to existing fuzz infrastructure at zero additional annotation cost. The breadth of adoption across domains (from VMMs and network protocols to aerospace, microkernels, and OS infrastructure) is evidence that Kani’s cargo-native workflow lowers the barrier to verification across a wide range of industrial Rust contexts.

Answer to RQ-III: Kani is sustainable in CI across diverse projects and domains. All projects in Table 1 run Kani on every code change, with CI runtimes ranging from under one minute to 69 minutes. The Rust standard library campaign verifies over 16,000 harnesses per code change. The cargo kani interface and proof-harness-as-code model enable verification to evolve alongside the codebase.

Threats to validity. Our case studies are not a random sample: the evaluated projects self-selected into Kani adoption, potentially over-representing favorable cases; the Cedar experiment partially mitigates this as a deliberate evaluation on a project not previously using Kani. Specifications can be wrong: three AI-generated contracts required correction (all caught by Kani, not human review). Harnesses without loop contracts provide bounded guarantees only; we report unbounded claims only where contracts or passing unwinding assertions justify them. The specification language does not address generics (contracts are verified per monomorphization), dynamic trait dispatch, or concurrent execution.

6 Related Work

Rust verification is an active area spanning dynamic analysis, deductive verification, and model checking. On the dynamic side, Miri [30] detects undefined behavior at runtime, including Stacked

Borrows [28] and Tree Borrows [44] aliasing violations (the primary class of UB that Kani does not model; §4.3). Rudra [6] scales to the full crates.io ecosystem via over-approximation, finding 264 bugs (76 CVEs). On the foundational side, RustBelt [29] and RefinedRust [24] provide machine-checked soundness proofs in Rocq at the cost of substantial expertise. Kani’s proof-harness approach builds on code-level model checking for C [13–15], extending it to Rust with type-aware nondeterminism and contract-based unbounded verification.

Verus [34] is the closest tool to Kani in ambition. It embeds specifications and proofs directly in Rust syntax, using linear ghost types for ownership reasoning and targeting Z3. The key difference is the verification paradigm: Verus requires proof code (lemma functions, triggers, ghost state), while Kani requires only contracts and harnesses. Verus’s SOSP 2024 case studies report approximately 5 lines of proof per line of implementation (6.1K impl., 31K proof); Kani’s Hifitime case study adds 125 contract annotations to 153 harnesses covering a 9.5K-line library. Kani is easier to adopt incrementally (zero annotations yield panic-freedom checks) but is limited to properties expressible within BMC or inductive contracts; Verus can express richer properties (e.g., complex data-structure invariants with ghost state) at higher annotation cost.

Other Rust verifiers include Prusti [4], a deductive verifier for safe Rust built on the Viper infrastructure; Creusot [22], which translates MIR into Why3’s WhyML with prophecy variables for mutable borrows; Flux [35], which brings liquid types to Rust with low annotation overhead; Aeneas [25] and hax [8], which extract Rust into theorem provers (F*, Lean, Rocq) for foundational or security-critical verification; and VeriFast [27], a separation-logic verifier for unsafe Rust that handles the most pointer-intensive modules in the Rust standard library campaign [17]. Kani differs from these tools by operating on MIR, providing cargo-integrated workflows, and supporting unbounded verification through function and loop contracts.

7 Conclusion

We presented Kani, an open-source model checker for Rust that provides both bounded and unbounded verification through a specification language of function contracts, loop contracts, quantifiers, and function stubbing. Our evaluation demonstrates that Kani finds bugs that testing and fuzzing miss (eleven bugs across four production codebases), that contracts upgrade bounded panic-freedom checks to unbounded functional correctness proofs (153 functional proofs in Hifitime), and that verification integrates sustainably into CI across diverse projects (over 16,000 harnesses in the Rust standard library campaign). Model checking requires minimal proof expertise, yet provides exhaustive guarantees that testing cannot. Kani makes this capability accessible to any Rust developer through cargo kani.

Data Availability. Kani and all evaluated projects are available as open source; bug fixes are referenced by pull request number in the text.

Acknowledgements. We thank all open-source contributors.

References

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Santa Clara, CA, USA, 419–434.
- [2] Amazon Web Services. 2024. s2n-quic: An implementation of the IETF QUIC protocol. <https://github.com/aws/s2n-quic>.
- [3] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3rd ed.). Springer, Berlin, Heidelberg. doi:10.1007/978-1-84882-745-5
- [4] Vytautas Astrauskas, Aurel Bily, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings* (Pasadena, CA, USA). Springer-Verlag, Berlin, Heidelberg, 88–108. doi:10.1007/978-3-031-06773-0_5
- [5] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (2020), 27 pages. doi:10.1145/3428204
- [6] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proc. ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, New York, NY, USA, 84–99. doi:10.1145/3477132.3483570
- [7] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) LNCS, Vol. 13243*. Springer, Cham, 415–442. doi:10.1007/978-3-030-99524-9_24
- [8] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. hax: Verifying Security-Critical Rust Software Using Multiple Provers. In *Verified Software: Theories, Tools and Experiments (VSTTE)*. Springer, Cham, 96–119. doi:10.1007/978-3-031-86695-1_7
- [9] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (Taipei, Taiwan) CCS '25*. Association for Computing Machinery, New York, NY, USA, 2729–2743. doi:10.1145/3719027.3765213
- [10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) Lecture Notes in Computer Science, Vol. 1579*. Springer, Berlin, Heidelberg, 193–207. doi:10.1007/3-540-49059-0_14
- [11] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions Department of Computer Science Report Series B, Vol. B-2020-1*, Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Jarvisalo, and Martin Suda (Eds.). University of Helsinki, Helsinki, Finland, 51–53.
- [12] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. 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, Article 1, 19 pages.
- [13] Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Software: Practice and Experience* 51, 4 (2021), 772–797. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2949 doi:10.1002/spe.2949
- [14] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. 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 (Seoul, South Korea) ICSE-SEIP '20*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/3377813.3381347
- [15] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification (CAV) Lecture Notes in Computer Science, Vol. 10982*. Springer, Cham, 430–446. doi:10.1007/978-3-319-96142-2_26
- [16] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) LNCS, Vol. 2988*. Springer, Berlin, Heidelberg, 168–176. doi:10.1007/978-3-540-24730-2_15
- [17] Byron Cook, Remi Delmas, Ziad Hassan, Bart Jacobs, Ranjit Jhala, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Rebecca Rumbul, Michael Tautschnig, Celina Val, and Carolyn Zech. 2026. Verifying the Rust Standard Library. In *NASA Formal Methods (NFM)*. Springer, Cham, 1–20. To appear.
- [18] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is Unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proc. IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA, 13 pages. doi:10.1145/3597503.3639136
- [19] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Keshia Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (2024), 28 pages. doi:10.1145/3649835
- [20] Datadog. 2024. lading: A load-testing framework. <https://github.com/DataDog/lading>.
- [21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) LNCS, Vol. 4963*. Springer, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24
- [22] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering (ICFEM) LNCS, Vol. 13478*. Springer, Cham, 90–105. doi:10.1007/978-3-031-17244-1_6
- [23] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT) LNCS, Vol. 2919*. Springer, Berlin, Heidelberg, 502–518. doi:10.1007/978-3-540-24605-3_37
- [24] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (2024), 26 pages. doi:10.1145/3656422
- [25] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. doi:10.1145/3547647
- [26] Li Huang, Sophie Ebersold, Alexander Kogtenkov, Bertrand Meyer, and Yinling Liu. 2026. Lessons from Formally Verified Deployed Software Systems. *ACM Comput. Surv.* 58, 8, Article 212 (Feb. 2026), 37 pages. doi:10.1145/3785652
- [27] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (NFM) Lecture Notes in Computer Science, Vol. 6617*. Springer, Berlin, Heidelberg, 41–55. doi:10.1007/978-3-642-20398-5_4
- [28] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. doi:10.1145/3371109
- [29] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154
- [30] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (2026), 31 pages. doi:10.1145/3776690
- [31] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2023. CBMC: The C Bounded Model Checker. arXiv:2302.02384.
- [32] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems, Erika Ábrahám and Klaus Havelund (Eds.)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26
- [33] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasice, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication SIGCOMM '17*. Association for Computing Machinery, New York, NY, USA, 183–196. doi:10.1145/3098822.3098842
- [34] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs Using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 286 (2023), 32 pages. doi:10.1145/3586037
- [35] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (June 2023), 25 pages. doi:10.1145/3591283
- [36] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems*

- Principles* (Shanghai, China) *SOSP '17*. Association for Computing Machinery, New York, NY, USA, 234–251. doi:10.1145/3132747.3132786
- [37] Joshua Liebow-Feeder and Jack Wrenn. 2024. zerocopy: Zero-cost memory manipulation. <https://github.com/google/zerocopy>.
- [38] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification CAV LNCS, Vol. 13965*. Springer, Cham, 3–17. doi:10.1007/978-3-031-37703-7_1
- [39] OASIS. 2019. Virtual I/O Device (VIRTIO) Version 1.1. <https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html>.
- [40] Christopher Rabotin. 2024. Hifitime: A High-Fidelity Time Management Library. <https://nyxspace.com/hifitime/>.
- [41] rust-osdev contributors. 2024. x86_64: Library for x86_64 specific functionality. https://github.com/rust-osdev/x86_64.
- [42] seL4 Foundation. 2024. rust-sel4: Rust bindings for the seL4 microkernel. <https://github.com/seL4/rust-sel4>.
- [43] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice ICSE-SEIP '22*. Association for Computing Machinery, New York, NY, USA, 321–330. doi:10.1145/3510457.3513031
- [44] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrowers. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (2025), 28 pages. doi:10.1145/3735592