Lessons Learned So Far From Verifying the Rust Standard Library (work-in-progress)

ALEX LE BLANC, University of Waterloo, Canada
PATRICK LAM, University of Waterloo, Canada

Although Rust primarily intends to be a safe programming language that excludes undefined behaviour, it provides its users with the escape hatch of unsafe Rust, allowing them to circumvent some of its strong compile-time checks. This additional freedom has some advantages, including potentially more efficient code, which is one of the main reasons why unsafe code is used extensively throughout Rust's standard library. However, because unsafe code also re-opens the door to undefined behaviour, Amazon has convened a community to verify the safety of the standard library, and in particular the unsafe code contained therein. Given that this effort is done in public and open-sourced, we have access to a wealth of information on how people are verifying the standard library, as well as what is currently possible and what still appears to be beyond the state of the art for verified software.

In this paper, we discuss the lessons learned thus far from this verification effort, from both our work on it, as well as that of the broader community. In particular, we start by reviewing what has been accomplished thus far, as well as the main tools used (specifically, their advantages and their limitations). We then focus on some of the remaining fundamental obstacles to verifying the standard library, and propose potential solutions to overcome them. We hope that these observations can guide future verification of not only the standard library, but also unsafe Rust code in general.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Formal methods**; *Specification languages.*

Additional Key Words and Phrases: Crowdsourced verification. Unsafe Rust, Bounded model checking.

1 Introduction

Rust is a systems programming language designed to provide memory safety and freedom from data races without sacrificing performance. It achieves this primarily through its novel ownership model, as realized by a set of rules enforced by a compiler component known as the borrow checker. The ownership model closely tracks object lifetimes and hence eliminates the need for a garbage collector and its associated runtime overhead. Due to its performance and safety, Rust has been employed in performance-critical domains where reliability is paramount, such as in browsers (e.g., Firefox), operating systems (e.g., RedoxOS), and distributed systems (e.g., TiKV).

However, Rust's ownership model, even with its many advantages, is sometimes overly restrictive, such as when writing low-level code. For this reason, Rust provides the `unsafe` keyword, which can be used to sidestep some of the compiler's guardrails, allowing programmers to do things like dereference raw pointers. Naturally, many of the safety guarantees offered by the compiler for `safe` code do not extend to `unsafe` code, and the onus of ensuring safety is instead shifted onto the developer. This is achieved through *encapsulation*, whereby `unsafe` code is isolated within `unsafe` blocks, which are in turn wrapped in public, safe APIs.

For `unsafe` code to be well-encapsulated, these APIs should only impose properly-documented safety constraints on clients (i.e., the APIs must be guaranteed to never trigger undefined behaviour as long as the documented constraints are safisfied). In general, the expectation, as enforced by Rust's clippy linter, is that `unsafe` blocks and methods come with a natural-language SAFETY comment (i.e., a comment that is prefixed with the word 'SAFETY'). These SAFETY comments describe the required safety constraints or reasons why the code is actually safe. Listing 1 illustrates

Authors' Contact Information: Alex Le Blanc, University of Waterloo, Canada, a6leblan@uwaterloo.ca; Patrick Lam, University of Waterloo, Canada, patrick.lam@uwaterloo.ca.

one safety property, which encodes an expectation on the program state. Formal verification, of course, requires a more formal specification of the safety properties, but also has the potential to automatically verify (or refute!) the property.

```
// SAFETY: PeekMut is only instantiated for non-empty heaps.
unsafe { self.heap.sift_down(0) };
```

Listing 1. A safety property for unsafe code within a binary heap operation.

Though some studies have reported that `unsafe` Rust code is usually well-encapsulated [2], the Rust standard library, which makes abundant use of unsafe code, has been found to contain many instances of poor encapsulation [16]. Moreover, to date, over 20 CVEs related to the standard library have been reported. These CVEs all appear to rely on unsafe code in some way[1].

For these reasons, Amazon has proposed a community-driven effort to verify Rust's standard library [10], which it targets in a segmented way, via challenges[2]. These challenges focus primarily on safety properties rather than functional correctness. One reason for the focus on safety is that, while there is a well-understood set of safety properties to check (given that we have an exhaustive list of possible sources of undefined behaviour[3]), there is no real consensus on the scope for functional (or *domain-specific*) correctness. We also have earlier work that commented specifically on this Amazon-led effort, including [4], which discusses what it means to verify the standard library, along with some suggestions on how best to do so.

In this paper, we discuss what we have learned so far from our own contributions to this effort (namely, verifying `transmute` and its uses). Alongside this, we also present what we have learned from studying how the other challenges are being solved.

We start by reviewing the current state of progress on Rust's standard library (Section 2), delving into the challenges themselves, as well as the tools that have been used and proposed to solve them. We then present key observations about verification of the standard library, guided by our experience (Section 3). We follow with an analysis of some of the main technical hurdles to overcome for this effort, as well as some solutions we propose (Section 4). Finally, we close with a higher-level discussion of broader extrapolations we have made, with the goal of guiding future verification of the standard library, as well as of unsafe code in general (Section 5).

2 State of Current Work

We discuss the current state of affairs with respect to verification of the Rust standard library, detailing the overall progress on Amazon's challenges, as well as providing a comparison of the tools being used in this effort.

2.1 Verification progress

Amazon's *verify-std* effort has seen significant involvement from Amazon staff, and it has also attracted interest from researchers at a number of universities[4].

As shown in Table 1, to date, 27 challenges have been posted, 9 of which have accepted solutions. The remaining 18 challenges are still open, and 3 of these have seen nontrivial engagement. The challenges target parts of the standard library's three main crates (`core`, `alloc`, and `std`) that might be particularly vulnerable to memory safety issues (such as callers of `transmute` and raw pointer

[1] One might complain that some of the CVEs relate to spawning subcommands, which technically is indeed unsafe and must be so labelled, but that is not in the same class as the rest of the CVEs, which are typically memory-safety errors.

[2] https://model-checking.github.io/verify-rust-std/intro.html

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

[4] Non-exhaustively, this includes Carnegie Mellon, UT Austin, Brown University, University of Waterloo, UCSD, and KU Leuven.

arithmetic operations) as well as data races (such as concurrency primitives). The rewards for completing the challenges, currently ranging from 5000 to 25000 USD, are based on the perceived difficulty of completion, as determined by the committee[5] for this verification effort.

It should be noted that even now, these challenges in no way exhaustively cover the entirety of unsafe code use in the standard library. Indeed, at the time of writing, of 9078 unsafe functions and safe abstractions of unsafe code in `core`, 361 are annotated with Kani function contracts[6], representing a coverage of 3.98%. For the `std` crate, this coverage rate is 1.4% (10 out of 714 such functions). Note that these annotated functions are not necessarily fully verified for safety (for this, it would be useful to track which functions have accepted peer-reviewed proofs). Thus, while there has been significant progress, the standard library is still far off from complete verification, even from a safety standpoint.

2.2 The tools used

The first tool approved for use in this effort was Kani, and to date, 6 out of 9 of the completed challenges use Kani. Since then, three other tools have been approved by the committee and integrated into the effort's Continuous Integration workflow, with three more in the process of being approved. We present them in Table 2, and briefly discuss the properties of these tools and the potential for other tools.

Kani [17] uses bounded model checking as implemented in CBMC [5]. Kani also has experimental support for loop invariants, which enable it to go beyond some of the limitations of bounded proof.

To give an example of Kani's use, in Listing 2, we have an integer doubling function that returns `None` if the input is greater than half the maximum `i32`, so as to avoid a potential integer overflow (note: this is just the specification of the hypothetical function, overflowing is not itself undefined behaviour). To check that `double` does not overflow and indeed returns `None` when its input is greater than half the maximum `i32`, we can write a 'proof harness', i.e., a separate function analogous to a test harness, but for proving that a property holds, rather than for testing. See Listing 3 for an example.

This proof harness first generates non-deterministic (or symbolic) integers via `kani::any()`—non-deterministic choice being a key differentiator of a *proof* harness versus a test harness—and then checks that if the generated input to `double()` is more than half the maximum i32, then `double` returns `None`. Because the input is non-deterministically generated, Kani can use the proof harness to verify the assertion for all inputs valid for the given type (here, `i32`).

Generally, proof harnesses are functions that call the functions under test and check that some properties hold; however, because they are run by Kani, they can leverage symbolic execution and bounded model checking, providing more assurance than a normal test. The boundedness of Kani's model checking causes limitations when there are loops to unroll.

[5] This committee consists primarily of Amazon staff, along with a few other invited members from academia and industry. They are in charge of reviewing newly proposed challenges, challenge solutions, and new tools.

[6] The vast majority of functions have been verified with Kani thus far, so including other tools in this statistic would not make a significant difference.

Table 1. Amazon Challenge Progress Overview

Challenge (#)	Progress	Tool used	Reward (USD)
transmute (1)	Complete	Kani	10,000
raw ptr arithmetic (3)	Complete	Kani	TBD
linked list (5)	Complete	VeriFast	5,000
NonNull (6)	Complete	Kani	TBD
core::time::Duration (9)	Complete	Kani	TBD
numeric primitives (11)	Complete	Kani	TBD
primitive conversions (14)	Complete	Kani	TBD
SIMD intrinsics (15)	Complete	Randomized testing*	20,000
RawVec (19)	Complete	VeriFast	10,000
SmallSort (8)	Progress	Kani	10,000
NonNull (12)	Progress	Kani	10,000
Cstr (13)	Progress	Kani	10,000
intrinsics using raw ptrs (2)	None		10,000
BTreeMap's btree::node (4)	None		10,000
atomic types & intrinsics (7)	None		10,000
String (10)	None		10,000
Iterator (16)	None		10,000
slice (17)	None		10,000
slice iter (18)	None		10,000
char fns in str::pattern (20)	None		25,000
substr fns in str::pattern (21)	None		25,000
str iter (22)	None		10,000
Vec part 1 (23)	None		15,000
Vec part 2 (24)	None		15,000
VecDeque (25)	None		10,000
Rc (26)	None		10,000
Arc (27)	None		10,000

* The results of randomized test cases involving models of the SIMD intrinsics and actual intrinsics were compared

```rust
// if x > i32::MAX / 2, should ret None
fn double(x: i32) -> Option<i32> {
    if x > i32::MAX / 2 {
        None
    } else {
        Some(x * 2)
    }
}
```

Listing 2. Example function definition

```rust
// check that if x > i32::MAX / 2,
// double(x) returns None
#[kani::proof]
fn check_double_no_overflow() {
  let num: i32 = kani::any();
  kani::assume(num > i32::MAX / 2);
  assert!(double(num).is_none());
}
```

Listing 3. Kani proof harness for double()

A second backend for Kani is the goto-transcoder, which generates inputs suitable for the ESBMC tool [6] (rather than CBMC, as usually targetted by Kani). Using the goto-transcoder adds support for k-induction and SMT solvers. In principle, this could result in more powerful verification (more

Table 2. Comparison of tools used in Amazon's verify-std effort

	Kani	**goto-transcoder**	**VeriFast**	**Flux**
Method	BMC	BMC	Separation logic, symbolic execution	FOL, refinement type checking
Usage	S	×	S	U
Accepted?	✓	✓	✓	✓
	RAPx	**Creusot**	**KMIR**	
Method	Abstract interpretation	FOL, deductive verification	Reachability logic, symbolic execution	
Usage	×	×	×	
Accepted?	×	×	×	

'S' = tool has solved a challenge, 'U' = used but not solved, '×' = unused

cases verified) using less CPU time and memory, but, for now, no challenges have been solved using goto-transcoder rather than Kani.

VeriFast [8] represents memory using separation logic and symbolically executes methods to verify them. In this context, VeriFast users effectively create annotated copies of standard library code augmented with VeriFast-specific annotations. Such annotations notably include loop invariants and inductive predicates, which VeriFast uses to perform unbounded verification over loops and dynamically-sized types, respectively (see Section 5 for further discussion on boundedness). These annotated functions are symbolically executed, and the resulting verification condition is passed to an SMT solver. VeriFast's use of separation logic allows for the expression of richer properties than other tools (like Kani), particularly ones complicated by the problem of aliasing (e.g., linked-list properties). There is a provision for automatically bringing the VeriFast changes up to date with changes to the standard library. At the time of writing, VeriFast is the only tool besides Kani (and the randomized testing for Challenge 15) to have been used in an accepted challenge solution (2 of the 9 accepted solutions).

Another approved tool is Flux [13], a refinement type checker for Rust. At the time of writing, Flux has not yet solved any challenges. Compared to VeriFast, it aims to be more lightweight, which it does by restricting predicates to first-order logic only (as opposed to VeriFast's separation logic), which does come with the disadvantage of making some properties harder to specify. Unlike Kani and goto-transcoder, it performs unbounded verification (it automatically infers loop invariants, and it allows users to specify invariants for even dynamically-sized types). However, it currently offers only limited support for unsafe code (e.g., it cannot track values written through pointers).

There are also three tools in the process of approval, namely KMIR, Creusot [7], and RAPx[7]. KMIR is developed by the company Runtime Verification, and uses the K framework[8] to define operational semantics for Rust's Middle Intermediate Representation. KMIR performs symbolic execution on the IR and uses reachability logic to verify needed conditions; it claims little dependence on SAT or SMT solvers. Creusot is similar to VeriFast in that it is a deductive verifier, but it only uses

[7] https://github.com/Artisan-Lab/RAPx
[8] https://kframework.org

first-order logic as opposed to separation logic, which again means that it can be fully automated and lightweight, but in exchange, it loses out on expressiveness. It still achieves unboundedness, via loop invariants and inductive properties. Finally, RAPx is a tool that performs contract-based abstract interpretation, and can automatically infer safety conditions for unsafe APIs. As is typical of similar static tools, it over-approximates, but achieves soundness in return.

There are at least half a dozen other tools which cover different points in the design space for Rust verification. Some of these tools, such as Prusti [1], focus on safe Rust, and thus are less useful for this effort. Other tools, such as RustHorn [15], could well be applicable for both existing and future challenges, but have not yet applied to take part in this effort. [4, 10] list some other tools.

2.3 Suggested tools for remaining challenges

To determine which tools are best for the remaining challenges, we must first identify some of the main verification obstacles that could influence the tool choice for different challenges. The main ones we have identified are:

(1) Does it require reasoning in a **concurrent** context? Accepted tools with well-documented support for this: VeriFast.
(2) Must the proofs hold for variables of **unbounded** size (e.g., slices)? Accepted tools with well-documented support for this: VeriFast, Flux.
(3) Must the proofs hold for **generic type** `T` (e.g., for a function `foo<T>(<input: T>)`, should we perform proofs over a finite set of concrete types, or generic type `T`)? Accepted tools with well-documented support for this: VeriFast, Flux.

In Table 3, we present the main verification obstacles for each remaining challenge, based on the above criteria. We further list which of the accepted tools are capable of completing the challenges[9]. We further note that while hybrid solutions (i.e., ones involving a combination of tools) have not yet been used or discussed, there certainly would be some value in using them (e.g., for Challenge 16, we could use VeriFast specifically for any function that involves unbounded or generic-typed reasoning, and then the more lightweight Kani for the others).

3 Things we learned

We discuss some practical lessons that we have learned so far from this effort to verify the standard library, including from our own experience with `transmute()`, as well as from our observations of others' work. We consider two categories of lessons: ones related to specification languages, and ones related to verification.

3.1 About specification languages

Internal safety properties. Rust, along with other languages, specifies that the violation of certain properties immediately cause undefined behaviour. These properties therefore cannot be checked as postconditions: the undefined behaviour happens before execution reaches the postcondition, making it impossible to soundly reason about any state after the undefined behaviour, including in particular any verification of the postcondition. We encountered several instances of these types of properties while verifying `transmute()` from Rust's standard library.

Consider for instance the function `from_raw_parts()`, shown in Listing 4. This function takes a pointer and a length. It returns a reference to a slice starting at the address pointed to by the input pointer and with length provided by the function input. Naturally, we would want to check that the resulting slice reference is well-aligned, but doing so as a postcondition is not useful, for the

[9] This is purely based on the identified verification obstacles. In reality, it is possible that some of the tools listed cannot solve the challenges for reasons that we are unaware of.

Table 3. Suggested Tools for Remaining Challenges

Challenge (#)	**Concurrent?**	**Unbounded?**	**Generics?**	**Tools**
intrinsics using raw ptrs (2)	×	×	×	Any
BTreeMap's btree::node (4)	×	×	×	Any
atomic types & intrinsics (7)	✓	×	×	VF
String (10)	×	✓	×	VF, Flux
Iterator (16)	×	✓	✓	VF, Flux
slice (17)	×	✓	✓	VF, Flux
slice iter (18)	×	✓	✓	VF, Flux
char fns in str::pattern (20)	×	✓	×	VF, Flux
substr fns in str::pattern (21)	×	✓	×	VF, Flux
str iter (22)	×	✓	×	VF, Flux
Vec part 1 (23)	×	✓	✓	VF, Flux
Vec part 2 (24)	×	✓	✓	VF, Flux
VecDeque (25)	×	✓	✓	VF, Flux
Rc (26)	×	×	×	Any
Arc (27)	✓	×	×	VF

'VF' refers to `VeriFast`.

reason stated above: by the time the postcondition is evaluated, there might already be undefined behaviour. This is because just creating a misaligned reference triggers undefined behaviour, even if it is not accessed. Thus, for all languages affected by immediate undefined behaviour, a person writing specifications would need some way to specify that such properties are to be checked inside the function body, before it can cause undefined behaviour. The commented assertion provides an example.

```
pub const unsafe fn from_raw_parts<'a, T>
(data: *const T, len: usize) -> &'a [T] {
    unsafe {
        //assert!(ptr::slice_from_raw_parts(data, len).is_aligned())
        &*ptr::slice_from_raw_parts(data, len)
    }
}
```

Listing 4. A function that needs an internal property to be checked

Because this is an intentionally simple example, it is actually straightforward to just put the assert as a function precondition for `from_raw_parts()`. However, for more complex functions where the potential source of undefined behaviour is much deeper, doing so would not be a viable option.

3.2 About verification

No one-size-fits-all approach. As discussed in section 2.2, originally, the only tool proposed for the verify-std effort was Kani. Since then, contributors have proposed their own tools, with approved tools goto-transcoder, VeriFast, and Flux. Proofs have been proposed using different tools, both out of necessity (e.g., proving properties of linked lists requires an unbounded approach, such as that provided by the separation logic-based VeriFast) and comfort (i.e., even though VeriFast is generally more expressive, people mostly still opt to use Kani when possible for the convenience offered by bounded model checking; it is just easier to use Kani than VeriFast). Modular verification approaches date back several decades [11], and continue to be advocated today within tools such as Gillian-Rust [3].

Trivially safe unsafe code. A significant number of the functions that we encountered during verification were trivially safe: that is, the set of documented safety constraints imposed on clients is explicitly null, and upon further eye-inspection, we found no way that these functions could be used unsafely, despite containing unsafe blocks. For instance, we found that 67% of functions within scope for the `transmute` challenge that directly call transmute are trivially safe in this way.

To take a specific example, `as_bytes()` takes a str slice, transmutes it into a u8 slice, and returns that (see Listing 5). Despite having an unsafe block wherein transmute is called, it is immediately obvious that this function cannot be used unsafely, as the u8 type has no value validity requirements (any bit sequence can be interpreted as u8s) and can be aligned to any address. In other words, there are no safety-related assertions to be proved; the SAFETY comment just points that out.

```
pub const fn as_bytes(&self) -> &[u8] {
    // SAFETY: const sound because we transmute two types with the same layout
    unsafe { mem::transmute(self) }
}
```

Listing 5. A trivially safe function

4 Remaining Obstacles

We explore some of the key problems with respect to verification of the standard library that remain to be solved, and we propose general plans for potential solutions.

4.1 Complex caller requirements

We encountered many functions that impose conditions on their inputs, where these conditions are not checkable in a single requires clause without additional scaffolding. For instance, if a function requires that an input be a positive integer, then this precondition is trivially expressible as a Rust expresssion. However, if a function expects that an input be an *initialized* `MaybeUninit<T>` (as is the case for `array_assume_init()`), it becomes much trickier, as Rust does not provide any way to track initialization. While adding support for tracking initialization would solve this particular problem, other functions impose constraints that are far more niche and domain-specific, such as Rc's `from_raw` expecting its input to be a pointer originally returned from `into_raw`. Developing specialized scaffolding for each of these different types of conditions is challenging, and is perhaps the main reason that researchers have proposed the use of specialized logics to reason about Rust.

For the case of initialization, Kani currently offers some limited support for reasoning about that, via its `uninit-checks` flag. When enabled, Kani keeps track of which memory is or is not initialized by toggling flags for corresponding shadow memory. The instructions for tagging the shadow memory are added via compile-time instrumentation, and the problem of aliasing is resolved using a conservative aliasing graph. Extending this approach further would be key, particularly to fully support tracking the initialization status of `MaybeUninit`[10], as it is fairly common in the standard library to assume that a `MaybeUninit` is fully initialized at a given point.

This static analysis-based technique could also be applied to other properties that require tracking client behaviour, like an input pointer needing to be returned from `into_raw`. Indeed, rather than tagging shadow memory as initialized or not, we would tag it as being returned from `into_raw` or not. The main things that change are the lattice and transfer function, but the underlying

[10] Kani does not fully support tracking aliasing when unions are involved, and in fact `MaybeUninit` is just a union under the hood.

infrastructure is the same. We therefore recommend generalizing the `uninit-checks` subsystem into an API where users could specify the details of their dataflow analyses. Assuming aliasing can be precisely tracked (which is one of the current main limitations of `uninit-checks`), we believe that any finite-state property could be tracked in this way. For properties with infinite states, we would need to instead rely on abstract interpretation. One tool that could be helpful in that case is the abstract interpretation engine MirChecker [14], although its support for verifying the standard library is currently limited.

4.2 Generic-typed function inputs

In some languages, like C, inputs for functions must have a concrete and prespecified type. Other languages, including Rust, allow functions to take inputs of generic types. For instance, `transmute` reinterprets a variable of generic type T as one of generic type U (here T is inferred from the type of the variable at compile time, whereas U can either be user-specified or inferred from the surrounding context). This significantly complicates writing proof harnesses, as Rust monomorphizes types at compile time.

For instance, suppose we wish to write a harness to prove that transmuting never modifies the bit pattern of the input. When writing this harness, we need to instantiate the type of the variable to be transmuted (again, due to monomorphization)[11]. The result of this is a plethora of near-identical harnesses where only the types are different, which may end up being both cumbersome and incomplete[12]. However, this is the approach currently used in the verify-std effort for these generic functions, due to a lack of alternatives.

It is possible that formal verification approaches (e.g., deductive verifiers) could be helpful here, as rather than exploring a finite state space as with bounded model checking, they opt for a symbolic approach. As established in Section 2.2, people who have solved challenges have revealed a preference for bounded model checkers like Kani for standard library verification where possible. However, it is unclear currently how Kani could be extended to support generically-typed harnesses. Rather, it could be that a hybrid solution for these challenges would be best (i.e., using Kani alongside another tool that would just be used for functions with generically-typed inputs).

4.3 Verification of concurrent code

Although safe Rust holds out the promise of “fearless concurrency” for its users, the promise does not extend to unsafe Rust. It is possible to write unsafe Rust that contains race conditions, which are immediate undefined behaviour in Rust. Thus, verifying the standard library must eventually include some provision for reasoning about concurrency, where appropriate. In fact, of the existing challenges, two explicitly require verifying an absence of data races (Challenges 7 and 27). Having said that, if unsafety is encapsulated sufficiently that the unique ownership property holds, then it should be sound to verify relevant methods as if they were in a sequential program.

With respect to currently-approved tools in the Rust verification challenge: Kani’s documentation states that it does not support concurrent features. Goto-transcoder uses Kani as a frontend, and thus presumably also does not support concurrency. However, the backends for Kani and goto-transcoder (CBMC and ESBMC) were both designed for concurrency, so we believe that there is no underlying limitation underneath the Kani frontend—it is just the frontend that needs further development.

[11] The type of the variable could in theory be generic in the harness if we pass the type as a type parameter to the harness, but this just shifts the type instantiation problem elsewhere.

[12] Whether or not this approach actually is complete depends entirely on the nature of the function being verified. For `transmute`, we do not expect the behaviour to vary significantly from one type to another (meaning this approach is closer to complete for `transmute`), but this is not always the case.

VeriFast does support concurrent Rust. This support has not yet been used in the standard library verification challenge, although there is apparently a work-in-progress solution for one challenge. The current applications of VeriFast in the context of Rust standard library verification are to linked lists and RawVec, presumably not executing in a context where concurrency matters.

To the best of our knowledge, KMIR and Flux do not support concurrency.

Looking beyond the tools that are currently involved with the standard library verification efforts, there are certainly tools that target concurrent code. Verus [12], for instance, was designed to verify multi-threaded concurrent code. This is not universal: Gillian-Rust [3], on the other hand, states that "[their] specifications apply in concurrent contexts, [but they] do not address concurrency-specific constructs or thread-safe types".

5 Discussion

Moving on from discussing things that we learned during verification, we continue with some broader observations that we have made while participating in the Rust standard library verification project.

This verification effort is challenge-driven. Amazon staff have proposed almost all of the challenges so far, but our experience is that it is possible to propose challenges from outside as well. At least 2 of the existing challenges did not originate from Amazon. As summarized in Section 2.2, non-Amazon groups appear to be more keen to contribute tools (6/7) than challenges.

All but 2 of the 27 challenges are satisfied with safety, memory safety, or absence of undefined behaviour. The smallsort[13] and SIMD[14] challenges are exceptions to this rule, in that they also require functional correctness; with the aid of proof scripts, VeriFast can ensure functional correctness. At least one challenge formerly included some verification of functional correctness, but has since removed that requirement. Despite the general lack of functional correctness, we believe that verifying safety is still a step forward compared to what exists now.

Challenge completion is peer reviewed. Specifically, a challenge is completed when a pull request closing the challenge is approved by reviewers and merged into the mainline repository. While the successful verification of all of the annotated code is guaranteed by the continuous integration infrastructure, the issue is that the contracts must be sufficient to ensure the desired properties (as discussed above, usually limiting ourselves to safety).

Of course, Kani is a bounded model checker; in this context, the main limitation is that arrays and other data structures are only explored to finite length—this is where the "small scope hypothesis" comes in. Loops can be unrolled, but with an unwinding assertion that ensures that the unexplored iterations are not reached (often because the iteration is over a data structure of fixed length); or, the user can provide a loop invariant. For simple types such as integers, Kani uses symbolic variables to explore the whole state space. Some challenges specifically state that they must be solved for all sizes of inputs, and it is difficult to imagine those challenges being solved with a bounded model checking approach like Kani's.

More broadly, we can reflect on what it means for the entire library to be verified, say for safety. Clearly, all challenges would need to be completed. It would also be ideal to continuously track unsafe code in the library and ensure that there is a traceability link to some (completed) challenge showing that the unsafe code has been verified. This would still rely on peer review from experts to ensure sufficiency of the challenge solutions, but it would help ensure that no code is uncovered. Currently, there are scripts in the main `verify-std` Github repository that allow tracking the number of functions that have been annotated and have corresponding proof harnesses, but these

[13] https://model-checking.github.io/verify-rust-std/challenges/0008-smallsort.html
[14] https://model-checking.github.io/verify-rust-std/challenges/0015-intrinsics-simd.html

are somewhat imprecise heuristics (i.e., they say nothing about whether or not the verification of these functions is sufficient).

Zero safety bugs found (so far). The verification of the Rust standard library has not yet revealed any confirmed safety bugs—it has only proven that some functions satisfy their specifications (as reviewed by experts). On one hand, this could be taken as further evidence that the unsafe code in the standard library is safe and well-encapsulated, on top of what has already been shown to this effect by works like RustBelt [9], which proved the standard library's well-encapsulation with respect to a core subset of the Rust language. On the other hand, there could also be a confirmation bias at play: because this effort is driven by the goal of proving correctness, rather than bug-finding, the findings could be inadvertently skewed towards positive results (i.e., absence of bugs). For this reason, running bug-finding techniques such as dynamic analysis (e.g. miri[15]) on the standard library is helpful, in parallel to the ongoing verification work.

In two instances, there have been comments about specification bugs found and fixed through this effort. Depending on the bug, one might find it either trivially easy or impossible to verify an incorrect specification.

6 Conclusion

We have reviewed the current state of the Rust standard library verification initiative, highlighting the multi-tool approach that has proven necessary, with bounded model checking being favored for its convenience where applicable. We also highlighted some key points about verifying the standard library that we have learned, to help inform future contributors.

Despite the progress made, some noteworthy technical hurdles persist, particularly in the verification of functions with generic inputs, complex preconditions, and concurrency, indicating potential avenues for future research in verification tooling.

References

[1] Vytautas Astrauskas, Aurel Bílỳ, 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 Symposium*. Springer, 88–108.

[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[3] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.* 9, PLDI, Article 186 (June 2025), 23 pages. doi:10.1145/3729289

[4] Alex Le Blanc and Patrick Lam. 2024. Surveying the Rust Verification Landscape. *arXiv preprint arXiv:2410.01981* (2024).

[5] 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: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*. Springer, 168–176.

[6] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2009. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 137–148. doi:10.1109/ASE.2009.63

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a foundry for the deductive verification of Rust programs. In *International Conference on Formal Engineering Methods*. Springer, 90–105.

[8] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015). doi:10.2168/lmcs-11(3:19)2015

[9] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[15] https://github.com/rust-lang/miri

[10] Rahul Kumar, Celina Val, Felipe Monteiro, Michael Tautschnig, Zyad Hassan, Qinheping Hu, Adrian Palacios, Remi Delmas, Jaisurya Nanduri, Felix Klock, Justus Adam, Carolyn Zech, and Artem Agvanian. 2024. Verifying the Rust Standard Library. In *VSTTE*.

[11] Patrick Lam. 2007. *The Hob System for Verifying Software Design Properties.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[13] 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

[14] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. 2183–2196.

[15] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4, Article 15 (2021), 54 pages. doi:10.1145/3462205

[16] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.

[17] 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*. 321–330.