Verifying the Rust Standard Library

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

Amazon Web Services, USA
`https://aws.amazon.com/`

Abstract. The Rust programming language is growing fast and seeing increased adoption due to performance and speed-of-development benefits. It provides strong compile-time guarantees along with blazing performance and an active community of support. The Rust language has experienced steady growth in the last few years with a total developer size of close to 3M developers. Several large projects such as Servo, TiKV, and the Rust compiler itself are in the millions of lines of code. Although Rust provides strong safety guarantees for `safe` code, the story with `unsafe` code is incomplete. In this short paper, we motivate the case for verifying the Rust standard library and how we are approaching this endeavor. We describe our effort to verify the Rust standard library via a crowd-sourced verification effort, wherein verifying the Rust standard library is specified as a set of challenges open to all.

Keywords: Rust · standard library · verification · formal methods · safe · unsafe · memory safety · correctness · challenge

1 Rust

Rust [9] is a modern programming language designed to enable developers to efficiently create high performance reliable systems. Rust delivers high performance because it does not use a garbage collector. Combined with a powerful type system that enforces ownership of memory wherein memory can be shared or mutable, but never both. This helps avoid data-races and memory errors, thereby reducing the trade-off between high-level safety guarantees and low-level controls – a highly desired property of programming languages. Unlike C/C++, the Rust language aims to minimize undefined behavior statically by employing a strong type system and an *extensible* ownership model for memory.

The extensible model of ownership relies on the simple (yet difficult) principle of enforcing that an object can be accessed by multiple aliases/references only for read purposes. To write to an object, there can only be one reference to it at any given time. Such a principle in practice eliminates significant amounts of memory-related errors [3]. In spite of the great benefits in practice, this principle tends to be restrictive for a certain subset of implementations that are too low-level or require very specific types of synchronization. As a result, the Rust language introduced the `unsafe` keyword. When used, the compiler may not be able to prove the memory safety rules that are enforced on `safe` code blocks. Alias tracking is not performed for raw pointers which can only be used in `unsafe` code blocks, which enables developers to perform actions that would be rejected by the compiler in `safe` code blocks. This is also referred to as *superpowers* [5] of `unsafe` code blocks. Examples of these superpowers include dereferencing a raw pointer, calling an unsafe function or method, and accessing fields of unions etc. A clear side-effect of this choice is that most if not all memory related errors in the code are due to the `unsafe` code blocks introduced by the developer.

Rust developers use *encapsulation* as a common design pattern to mask unsafe code blocks. The safe abstractions allow `unsafe` code blocks to be limited in number and not leak into all parts of the codebase. The Rust standard library itself has widespread use of `unsafe` code blocks, with almost 5.5K `unsafe` functions and 4.8K `unsafe` code blocks. In the last 3 years, 40 soundness issues have been filed in the Rust standard library along with 17 reported CVEs, even with the extensive testing and usage of the library. The onus of proving the safety and correctness of these `unsafe` code blocks is on the developers. Some such efforts have been made, but there is still a lot of ground to cover [7].

Verifying the Rust standard library is important and rewarding along multiple dimensions such as improving Rust, creating better verification tools, and enabling a safer ecosystem. Given the size and scope of this exercise, we believe doing this in isolation would be expensive and counter-productive. Ergo, we believe that motivating the community and creating a unified crowd-sourced effort is the desirable method, which we hope to catalyze via our proposed effort.

2 Rust Verification Landscape

A common misconception Rust developers have is that they are producing `safe` memory-safe code by simply using Rust as their development language. To counter this, there have been significant efforts to create tools and techniques that enable verification of Rust code. Here we list (alphabetically) some tools:

- **Creusot** [4] is a Rust verifier that also employs deductive-style verification for `safe` Rust code. Creusot also introduces **Pearlite** - a specification language for specifying function and loop contracts.
- **Gillian-Rust** [2] is a separation logic based hybrid verification tool for Rust programs with support for `unsafe` code. Gillian-Rust is also linked to Creusot, but does in certain cases require manual intervention.
- **Kani** [10] uses bounded model checking to verify generic memory safety properties and user specified assertions. Kani supports both `unsafe` and `safe` code, but cannot guarantee unbounded verification in all cases.
- **Prusti** [1] employs deductive verification to prove functional correctness of `safe` Rust code. Specifically, it targets certain type of *panics* and allows users to specify properties of interest.
- **Verus** [8] is an SMT-based tool used to verify Rust code and can support `unsafe` in certain situations such as the use of raw pointers and unsafe cells.

- There are several other tools which are in the related space, but we do not list them here explicitly.

3 Verifying the Rust Standard Library

We are proposing the creation of a crowd-sourced verification effort, wherein verifying the Rust standard library is specified as a set of challenges. Each challenge describes the goal and the success criteria. Currently, we are focusing on doing verification for memory-safety. The challenges are open to anyone. This effort aims to be *tool agnostic* to facilitate the introduction of verification solutions into the Rust mainline and making verification an integral part of the Rust ecosystem. Towards this, we have been working with the Rust language team to introduce function and loop contracts into the Rust mainline and have created a fork of the Rust standard library repository `https://github.com/model-checking/verify-rust-std/` wherein all solutions to challenges and verification artifacts are stored. Challenges can come in various flavors: 1/ specifying contracts for a part of the Rust standard library, 2/ specify and verify a part of the Rust standard library, and 3/ introduce new tools/techniques to verify parts of the Rust standard library. The repository provides templates for introducing new challenges, new tools, and instructions on how to submit solutions to challenges. To date, we have over 20 students, academics, and researchers engaging.

As part of this effort, we are also creating challenges. For example, we have created a challenge to verify the String library in the standard library [6]. In this challenge, the goal is to verify the memory safety of `std::string::String` and prove the absence of undefined behavior (UB). Even though the majority of `String` methods are safe, many of them are safe abstractions over unsafe code. For instance, the insert method is implemented as follows :

Listing 1.1. Unsafe usage in String

```
  pub fn insert(&mut self, idx: usize, ch: char) {
    assert!(self.is_char_boundary(idx));
    let mut bits = [0; 4];
    let bits = ch.encode_utf8(&mut bits).as_bytes();

    unsafe {
      self.insert_bytes(idx, bits);
    }
  }
```

The goal also specifies the *success criteria* that must be met for the solution to be reviewed and merged into the CI pipeline.

Listing 1.2. Success criteria for the String challenge.

```
Verify the memory safety of all public functions that are
safe abstractions over unsafe code:
    unbounded: from_utf16le, from_utf16le_lossy,
               from_utf16be, from_utf16be_lossy,
               remove_matches, insert_str,
               split_off, replace_range, retain
    others: pop, remove, insert, drain, leak,
            into_boxed_str
Ones marked as unbounded must be verified for any
string/slice length.
```

Example of a solution for a challenge can be found in [11]. This particular solution introduces new contracts for `char` and `ascii_char`. The contracts are also verified using Kani.

Our call to action to you is to come and be a part of this effort and contribute by solving challenges, introducing new challenges, introducing new tools, or helping review and refine the current processes!

Acknowledgments. We would like to thank all the academic partners that have helped us shape challenges, started contributing to challenges, and provide invaluable advice throughout the process of jump starting this initiative. We also would like to thank Niko Matsakis, Byron Cook, and Kurt Kufeld for their support and leadership.

References

1. Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The Prusti project: Formal verification for Rust. In: NASA Formal Methods Symposium. pp. 88–108. Springer (2022)
2. Ayoun, S.É., Denis, X., Maksimović, P., Gardner, P.: A hybrid approach to semi-automated rust verification. arXiv preprint arXiv:2403.15122 (2024)
3. Biesterbosch, C., Vatolina, V.: The Impact of Rust on Security Development (2024), `https://www.riscure.com/the-impact-of-rust-on-security-development`
4. Denis, X., Jourdan, J.H., March'e, C.: Creusot: a foundry for the deductive verification of Rust programs. In: International Conference on Formal Engineering Methods. pp. 90–105. Springer (2022)
5. Documentation, R.: Unsafe Rust, `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html`
6. Hassan, Z.: Memory Safety of String (2024), `https://model-checking.github.io/verify-rust-std/challenges/0010-string.html`
7. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–34 (2017)
8. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (November 2024)
9. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014)

10. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in Rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 321–330 (2022)
11. Zech, C.: char and ascii_char contracts (2024), https://github.com/model-checking/verify-rust-std/pull/48