Verifying the Rust Standard ibrary

Rahul Kumar, Celina Val, Felipe Monteiro, Michael Tautschnig, Zyad Hassan, Qinheping Hu, drian Palacios, Remi Delmas, Jaisurya Nanduri, Felix Klock, Justus dam, Carolyn Zech, and rtem gvanian

mazon Web Services, US
https:// ws. m zon.com/

bstract. 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. lthough Rust provides strong safety guarantees for `s fe` code, the story with `uns fe` 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. s 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. lias 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. 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 *enc psul tion* 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 andscape

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:

- **reusot** [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 *p nics* 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.