Internship Proposal: Static Analysis of Rust Programs

Location of the internship: Prosecco Team, Inria Paris, France
Advisor: Aymeric Fromherz

Context. Over the past decade, the Rust programming language has gained traction both in academia and in industry, consistently ranking as the most beloved langauge by developers for the past 8 years [1]. A large part of this success stems from several key features of the language: Rust provides both the high performance and low-level idioms commonly associated to C or C++, as well as promises of memory-safety by default thanks to its rich, borrow-based type system. The safety guarantees of Rust are particularly appealing for security-critical systems: leading governments now also recommend Rust [9].

Unfortunately, despite being safer than C or C++, Rust programs are not immune to bugs and vulnerabilities. Case in point, between January 1, 2024 and October 1, 2024, 76 security advisories against Rust crates were filed on RUSTSEC, a vulnerability database for the Rust ecosystem. These vulnerabilities arose due to several reasons. First, Rust programs are allowed to *panic* at runtime, e.g., after an out-of-bounds array access, thus guaranteeing memory safety at the expense of a (clean) aborted execution. Second, the limits imposed by Rust's type checker are in some cases too restrictive, leading to the use of the *unsafe* escape hatch. Code marked as *unsafe* allows the use of unchecked C-like pointer and object operations, which are not checked by the Rust compiler to follow Rust's ownership discipline ensuring memory safety. As a consequence, the leading causes of memory safety violations in Rust are mishandlings of unsafe code, commonly leading to high-profile vulnerabilities. To enforce those properties that fall outside the scope of Rust's borrow-checker, we propose to investigate the use of static analysis to analyze and reason about Rust programs.

Goals. The goal of this internship is to develop novel static analyses to reason about Rust programs and detect possible vulnerabilities. After a literature review of existing work on analyzing Rust projects [6, 3, 5] and a survey of common Rust vulnerabilities as reported on RUSTSEC, the work will particularly focus on unsafe Rust code, and especially on memory and pointer analysis to identify memory safety violations.

Many previous works studied pointer analysis in the context of memory-unsafe languages such as C or C++, leading to an extensive literature and several points-to analysis being proposed [7, 8, 2]. However, the Rust programming language provides several invariants as part of its borrow-checker that restrict possible aliasing patterns, and more broadly, authorized usages of memory. We therefore propose to investigate adapting known analyses to Rust, studying how Rust invariants can be leveraged to simplify, improve, and make more efficient automated memory reasoning.

As part of this work, a successful intern will be expected to design a prototype implementation of the analysis technique able to reason about small test programs containing unsafe memory accesses. To assist with the development of the analysis, the intern will rely on the Charon platform [4], a recent framework mainly developed in the Prosecco team that simplifies interacting with the Rust compilation pipeline. The intern will benefit from regular interactions with the team developing Charon and working on related projects targeting formal verification of Rust programs. Code developed as part of this project will be released under an open-source license.

A possible internship plan is therefore the following:

- Background reading on Rust analysis tools, Rust tutorial
- Survey of the RUSTSEC vulnerability database
- Background reading on pointer analysis
- Development a first Rust pointer analysis inspired by existing work
- Implementation of the analysis, extensions to a larger class of patterns
- Benchmarking of the implementation, evaluation on small testcases
- Evaluation on real-world examples, including public Rust crates

Qualifications. This internship of 4 to 6 months would be hosted by the Prosecco Team at Inria Paris, and is particularly well-suited to a student pursuing a master's degree in computer science. The preferred qualifications for the student at the beginning of the internship would be:

- Knowledge of the Rust programming language
- Experience with static analysis, ideally pointer analysis
- Knowledge of program semantics
- Strong implementation skills, ideally in OCaml to interact with Charon
- Motivation to work with, and improve experimental research tools

References

[1] 2023 Stack Overflow Survey: Rust is the most admired programming language, making it the most loved language for 8 years in a row. https://www.reddit.com/r/rust/comments/149cu1k/2023_stack_overflow_survey_rust_is_the_most/.

[2] Lars Ole Andersen. Program analysis and specialization for the c programming language. 1994.

[3] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.

[4] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. Charon: An analysis framework for rust. *arXiv preprint arXiv:2410.18042*, 2024.

[5] Miri Contributors. Miri, an Undefined Behavior detection tool for Rust. https://github.com/rust-lang/miri.

[6] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. Yuga: Automatically detecting lifetime annotation bugs in the rust language. *IEEE Transactions on Software Engineering*, 2024.

[7] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[8] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[9] The White House. Back to the Building Blocks: a Path Toward Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.