Translation Between C Style Languages and Rust

*

1st Daniel Dean
The College of Information Sciences and Technology
The Pennsylvania State University
State College, PA, United States
dpd5518@psu.edu

***Abstract*—C and C++ have been dominant languages in the systems programming space since the 1970s. Both of these languages are immensely powerful, but this power comes at the cost of security. Memory security issues have plagued this style of programming since its inception. The tradeoff between speed and security is a false dichotomy, however. Rust is a programming language with the speed of C/C++ and the memory safety guarantees of a high-level language. The tradeoff that Rust makes is for complexity and speed of development. The problem we are presented with is the transfer from a C dominated space to a safer Rust dominated space. This paper will give a background on the security mechanisms of rust, the current state of automatic translation from C to Rust, and discuss what the future may entail for security minded individuals in the systems programming space.**

***Index Terms*—C/C++, Rust, Memory Safety, C2Rust**

I. Introduction

In the realm of systems development, C/C++ (henceforth referred to as C for brevity) are the typical choice when it comes to writing complex yet highly efficient systems. While this offers fast software with relatively expedient speed of development, it comes with one major cost: safety. More specifically, C can be written in such a way that allows for memory issues which in the best case could lead to slower software and in the worst case could lead to a bad actor hijacking the execution of the application. Some of these memory issues include buffer overflow attacks, dangling pointers, and memory leaks.

There have been many proposed solutions to these memory safety issues. But the simplest is re-implementing code bases written in C in a memory safe language. While this could be any memory safe language, Rust has recently gained popularity, evident in its appearance in the Linux kernel [1]. In early 2024 the United States Government released a report on the dire nature of software security as it relates to national cyber security goals. The report emphasizes the importance that programming languages play in the development of secure software. They offer the Rust programming language as a potential solution to decades of dubious memory management caused by the C languages [2]. This may signify a push from the government to require that new applications must be written in Rust, or legacy C applications must be rewritten in Rust – a simple solution, but a logistical nightmare.

While Rust offers high performance memory safety, many of the use-cases in which it is needed already have a C implementation. The transition from C to Rust poses a massive logistical problem. This problem is compounded by Rust's relatively steep learning curve, limiting access to talent that could feasibly implement these C programs in Rust. Furthermore attempts to fully automate this process have not yet proven fruitful.

II. Background

The reason that codebases such as the Linux kernel are written in C is the high level of control these languages offer over the machine, allowing for highly performant code. Two principal aspects of C that aid in performance are the manual memory management and low runtime overhead. The traditional approach to ensure memory safety is to automate away memory management with a garbage collector; Lisp and Java are examples of garbage collected languages. Notwithstanding bugs in the implementation of the garbage collector, this guarantees memory safety, but results in more overhead, as the garbage collector must exist at runtime, causing performance degradation. This approach to memory management creates a false dichotomy between memory safety and performance.

Rust achieves memory safety without a garbage collector through the use of strict compile time rules: Lifetimes, Ownership, Borrowing, and Exclusive Mutability, all of which are checked at compile time [3]. Lifetimes simply denote how long a variable is valid in a program, which the rust compiler's borrow checker typically implicitly derives. For more complex lifetimes, the borrow checker will fail to automatically denote the lifetime and thus it must be done manually. This is a point of contention for new Rust programmers, as demonstrated through an analysis of stack overflow questions [4]. Borrowing Ownership and Borrowing are two related ideas in Rust where each variable owns a value, which changes when another variable takes ownership, or the value is passed to a function. Borrowing is the passing of a reference to a variable or function, without changing ownership. Exclusive mutability is ensured through the compiler by enforcing that there is either one mutable reference or an arbitrary amount of immutable references [4], [5]. There are also checks that are compiled into the machine code for each Rust program, which could be considered overhead. Performance wise, the most impactful of

these is the bounds checker, which checks for out-of-bounds accesses [3]. This suite of compile time checks guarantee memory safety so long as they are not deactivated using an unsafe block [5].

III. Example of Vulnerabilities

In this section, I will provide examples of some common C vulnerabilities. I will offer two examples: stack buffer overflows, and dangling pointers. This section will include code, compilation errors, and explainations.

IV. Literature Review

A. C2Rust

In this section I will describe the current capabilities of the C2Rust, an automated translation software between C and Rust. This includes different iterations that have enumerated on this idea to try to automatically generate safe Rust.

1) Limitations: The reason that translation between C and Rust is so difficult is the inability to automatically translate into safe Rust. The crux of this problem is the translation from unsafe Rust to safe Rust. In this section, I will touch upon some of the technical issues that come up when trying to solve this problem.

B. Use of LLMs

In this section, I will discuss the possibility of using Large Language Models (LLMs) to aid in translation between these two language paradigms.

V. Alternative Solutions

In this section I will discuss possible alternative solutions to switching between C and Rust.

A. C++ Memory Safety

One solution is to lean into C++ security, and enforce certain coding practices that guarantees that memory safety. Specifically I will research the use of compile time safety guarantees that can be offered for C++. This may run into the same issues as Rust, however.

B. Zig

Zig is another systems programming language that is more comparable to C than C++. In this section I will discuss its safety features and use cases.

VI. Discussion and Conclusion

This final section of the paper will include a short summary of the paper and offer final remarks.

References

[1] L. Torvalds, "Linux/Rust at master," GitHub, 2024. Available: https://github.com/torvalds/linux/tree/master/rust. [Accessed: Mar. 18, 2024]

[2] The United States Government, "Back to The Building Blocks: A Path Towards Secure and Measurable Software," 2024. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[3] Y. Zhang, Y. Zhang, Georgios Portokalidis, and J. Xu, "Towards Understanding the Runtime Performance of Rust," 37th IEEE/ACM International Conference on Automated Software Engineering, Oct. 2022, doi: https://doi.org/10.1145/3551349.3559494

[4] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust," Proceedings of the 44th International Conference on Software Engineering, May 2022, doi: https://doi.org/10.1145/3510003.3510164

[5] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," ACM Transactions on Software Engineering and Methodology, vol. 31, no. 1, pp. 1–25, Jan. 2022, doi: https://doi.org/10.1145/3466642