Evaluating Rust for a Secure Camera-based HVAC Control System

Benson Har
University of California, Los Angeles

1 Introduction

Haversack Inc. is interested in developing SecureTCE, a Thermal Comfort Estimation (TCE) system for controlling a building's temperature. The system consists of cameras with limited processing power and memory connected wirelessly to base stations that control the heating, ventilation, and air conditioning (HVAC) units. To alleviate customers' concerns over vulnerabilities in C and C++, we are tasked with researching whether Rust would be a suitable language to develop with instead. Specifically, we are interested in Rust's strengths and weaknesses, performance, reliability, security, and applicability towards embedded systems. Here, we consider Rust's latest stable release: Version 1.59.0.

2 Rust as a Language

Rust is a low-level programming language designed for performance and safety. It has a growing community and is supported by many companies. Though Rust is relatively new, it should be stable for the coming years, as it has good documentation and plans for the next 40 years. We talk about some of Rust's notable features here.

2.1 Typing

Rust is primarily a statically typed language. However, unlike C and C++, it is dynamically typed or a mixture of both. For instance, Rust requires function arguments and constants to have explicit types, while type inference can be done inside of function bodies. Rust also has NULL (or None) as an optional type like in ML/OCaml, eliminating errors related to using NULL values. Overall, this system allows Rust code to have the performance that comes from static checking while also the ease in development that comes with dynamic typing.

2.2 Performance

Rust is very fast and memory-efficient. One reason is that it has no runtime and garbage collector, opting for a memory management system that enjoys the speed of C and C++ while mitigating some of the complexity and issues C and C++'s memory management has. This is mainly due to Rust's ownership system, which will be discussed later.

In addition, Rust is a compiled language like C and C++, so it is faster than interpreted languages like Python and JavaScript. It can also directly access hardware and memory, meaning we can make choices that are more efficient in terms of memory management.

Moreover, Rust supports concurrent and parallel programming. C and C++ also support those features but what makes Rust special is that it makes developing concurrent programs safer. This is due to Rust's system of ownership and type checking, causing many runtime concurrency errors to be caught at compile time. This and the elimination of data races (similar to race conditions) result in code with fewer bugs. This all leads to developing fast code in a shorter period of time.

2.3 Reliability

As mentioned before, Rust's system of ownership and type checking result in more reliable code. They guarantee memory-safety and thread-safety, which eliminates many classes of bugs at compile-time. While having many compiler errors can seem daunting, they force us to write more reliable code with less bugs, which is suitable for SecureTCE since it needs to run over long periods of time and has embedded system components (cameras).

2.4 Flexibility

Rust programs are also easy to deploy. Rust's package manager `cargo` allows us to simply execute `cargo build` to compile our application. We also have the option of using external

libraries, called crates, that are published by the Rust community with `crates.io`, a registry of packages like Node.js's npm. It most likely will not be used extensively, if at all, by SecureTCE due to it introducing third party dependencies. What is more interesting is that if we need functionality from a C library, Rust has an easy way to integrate C and C++ code with Rust code.

Moreover, when we find Rust too restricting, we can turn on Rust's `unsafe` mode. This mode grants us additional power like dereferencing a raw pointer or calling an unsafe function, which may be required in scenarios like low-level programming. While this can make the program more vulnerable, an added benefit of this feature is that it isolates risky code, as we know where this unsafe code is. And, using this mode does not disable the borrow checker (discussed later) nor any of the safety checks.

3 Low Level Programming

Rust is well suited for low-level programming. A large part is due to the reliability we explored earlier, which Rust enforces with strong static analysis. This means that more errors will typically be caught at compile time than with C and C++.

In addition, Rust can be used without dynamic memory allocation which is important for reliability in embedded systems. As with C and C++, we can initialize data structures with fixed memory. What makes Rust special here is that we can explicitly check when we are using too much memory. Specifically, we can use static variables and set a maximum size for the call stack that allows the linker to detect if more memory is being used than what is physically available. And, flags like `-Z emit-stack-sizes` will analyze stack usage.

Furthermore, Rust is an ahead-of-time compiled language. This means that once we compile the program, we can run the executable even without having Rust installed. This is good for SecureTCE if we load the compiled Rust programs into the cameras, as it would save additional memory.

4 Programming Freestanding Software and Interfacing with Hardware

Rust provides a lot of support for interfacing with different low-level hardware devices. Rust mainly achieves this through embedded-hal. Embedded-hal is Rust's approach to hardware abstraction layers (HAL). They are a set of "traits" used to define implementation contracts between HAL implementations and firmware (drivers and applications). With this, we can initialize drivers as instances that implement a certain trait which can allow us to interact with the cameras SecureTCE wants to use and access the network that SecureTCE devices would be on.

With this in mind, we can write freestanding software, as we can write Rust code that directly interacts with the hardware. This circumvents the need for an operating system, which may have vulnerabilities, and is perfect for SecureTCE's needs.

5 Security

As the customers have noted, C and C++ programs can be vulnerable to attacks. In fact, according to an article on ZDnet, about 70% of vulnerabilities Microsoft addresses each year in security updates are due to memory safety issues, which C and C++ programs are susceptible to. Rust is designed to mitigate this issue.

5.1 The Ownership System

One of Rust's core concepts is "Ownership". According to the Rust documentation, ownership is the set of rules that a Rust program follows to manage memory. It is Rust's solution to gaining C and C++ speed while having the reliability of a garbage collector. The lack of garbage collector and runtime can make Rust software quite simple and stripped down, eliminating some potential vulnerabilities there too.

At the most basic level, how ownership works is that a value has one variable called its owner and when that variable goes out of scope, the value will be dropped (deleted/freed). This idea of freeing memory after the scope ends is called RAII (Resource Acquisition Is Initialization). It is automatically done when a scope ends and is seen in languages like C++ (usage of destructors). We should also note that bugs like double freeing cannot occur, as values have only one owner. This implies that assigning a value to multiple variables in Rust essentially gives ownership to the last variable (called moving).

Rust variables are by default immutable. Treating them this way results in safer code and will make using concurrency easier. More interesting is the concept of references. Like variables, they are by default immutable and like variables, they can be made mutable. However, instead of holding a value, they are typically used to point to dynamically allocated data, like pointers in C and C++. This provides a way for functions to "borrow" the data in an immutable way. What makes them different from pointers in C and C++ is that they guarantee that the data they point to is valid. What this means is that the data is guaranteed to not be undefined unlike in C and C++ where there can be dangling pointers or data that is being read and written to simultaneously. A lot of this is achieved through what was mentioned earlier (scopes and immutability). However, one thing to note is that Rust makes mutable references safe, as well. It does this by enforcing the rule that you can have only one mutable reference to a particular piece of data at a time. This eliminates risk of data races, where one pointer is being used to read data while another is being used to write data.

Furthermore, Rust has a feature called generic lifetimes. The main purpose of this is to prevent dangling references. Specifically, Rust's compiler has a "borrow checker" that checks whether borrows are valid by comparing scopes (which are essentially the boundaries of a variable's lifetime). When it encounters a case where the lifetime is ambiguous (ie, returning a reference from an if statement), it needs generic lifetime parameters to allow the borrow checker to perform its analysis.

All this is to keep the programmer accountable and most importantly eliminate some of the biggest causes of vulnerabilities in C and C++ code.

5.2 Code Auditing

Rust makes auditing relatively easy. For one, its syntax is similar to C++ so it should be easy to follow for those already experienced in C++ or any other object-oriented language. Rust's typing system also eliminates a number of memory bugs and instances of data races, taking care of some of the work that would be in an audit. Moreover, Rust comes with tools that support package management, auto-formatting, and type-inspection. It also has good testing support through `cargo test`.

6 Anticipated Problems

There are some drawbacks however. As stated before, Rust is a fairly new language. This means that the number of third party libraries is sparse when compared to languages like Python or C++. Moreover, development time may be slower, as Rust's strong type and focus on memory safety will result in more compilation errors. This issue is especially compounded by the novelty of Rust's ownership system, which will take some time to adjust to. That being said, these problems are not too difficult to overcome, as Rust has a growing community and these features result in better code.

7 Conclusion

Overall, Rust would be suitable for SecureTCE. Rust is a high performance language with emphasis on memory safety. As we found in our evaluation, many vulnerabilities originate from memory-related bugs. With Rust's ownership system and strong typing, these can be caught at compile time and lead to more reliable and secure code. At the same time, we evaluated Rust for its use in embedded systems. Given Rust's ample support for developing low-level code and its ability to directly interact with hardware, like network interfaces and cameras, Rust is appropriate for being used to program SecureTCE's devices.

References

Rust Documentation
https://www.rust-lang.org/

Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues
https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safe

Krzysztof Wróbel. Why is Rust programming language so popular?
https://codilime.com/blog/why-is-rust-programming-language-so-popular/

Why a Code Audit is Critical Before Buying or Selling your Business
https://www.clearlaunch.com/code-audit/

Jake Goulding. What is Rust and why is it so popular?
https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/