Rust for Certifiable Software: Bridging Communities

Jose Ruiz (AdaCore)

HISC, October 22nd 2024

Outline

- Why Rust for safety-critical embedded systems
- What is needed for safety-critical certifiable Rust
- Conclusion

About Rust

- **Rust is designed for safety and performance**
  - ○ Type-safe and Memory-Safe
  - ○ Supports concurrent and parallel programming

    "fearless concurrency"
  - ○ Provides high performance

    no garbage collection

- Rust is favored by many developers *(Ranked "most loved language" for the last seven years on Stack Overflow)*

stackoverflow

- Rust syntax is easily understood by developers with C or C++ knowledge

US White House Office of the National Cyber Director Report: Back to the Building Blocks

Mentions Rust by name as an example of a memory-safe language that:

- allows code to be close to the kernel
- supports determinism
- does not have a garbage collector

(Of course, Ada does these as well!)

That Rust is the singularly named memory-safe language in the report is telling.

BACK TO THE BUILDING BLOCKS:
A PATH TOWARD SECUR
MEASURABLE SOFTWA
FEBRUARY 2024

★ ★ ★ ★ ★ ★

Memory Safe Programming Languages

Since many cybersecurity issues start with a line of code, one of the most effective ways to address those issues is by examining the programming language itself. Ensuring that a programming language includes certain properties, such as memory or type safety, means software built upon that foundation automatically inherits the security those features provide.

Cybersecurity solutions should be informed by engineering best practices, and technology manufacturers building software can tackle this issue by consistently using secure building blocks; specifically, adopting memory safe programming languages. There is strong evidence that now is the time to make these changes. First, technical solutions already exist; there are dozens of memory safe programming languages that can – and should – be used. Technology manufacturers are already able to design and build new products in memory safe programming languages from day one. Second, the transition to memory safe programming languages has a demonstrably positive effect on cybersecurity. Industry analysis has shown in some cases, that despite rigorous code reviews as well as other preventive and detective controls, up to 70 percent of security vulnerabilities in memory unsafe languages patched and assigned a CVE designation are due to memory safety issues.[viii] When large code bases are migrated to a memory safe language, evidence shows that memory safety vulnerabilities are nearly eliminated.[ix]

For new products, choosing to build in a memory safe programming language is an early architecture decision that can deliver significant security benefits. Even for existing codebases, where a complete rewrite of code is more challenging, there are still paths toward adopting memory safe programming languages by taking a hybrid approach. For example, software developers can identify the critical functions or libraries based on risk criteria and prioritize efforts to rewrite those first.[x]

Building new products and migrating high-impact legacy code to memory safe programming languages can significantly reduce the prevalence of memory safety vulnerabilities throughout the digital ecosystem.[xi] To be sure, there are no one-size-fits-all solutions in cybersecurity, and using a memory safe programming language cannot eliminate every cybersecurity risk. However, it is a substantial, additional step technology manufacturers can take toward the elimination of broad categories of software vulnerabilities. A recent report authored by CISA, the NSA, the FBI, and international cybersecurity agencies entitled *The Case for Memory Safe Roadmaps,* provides guidance for manufacturers with steps to implement changes to eliminate memory safety vulnerabilities from their products.[xii]

Memory Safe Hardware

In April of 1970, an on-board explosion derailed Apollo 13's mission to the Moon. Two days into the astronauts' voyage, an exposed wire ignited a fire and caused one of the ship's two oxygen tanks to burst.[xiii] The astronauts' only hope for survival was for the rocket scientists to react ingeniously – and fast. Employing the laws of physics and the rules of mathematics, the aerospace

8

BACK TO THE BUILDING BLOCKS

Rust is on everybody's mind!

Memory Safety (I)

- **Ensure that data accesses are correct**
  - Consistent with data type and lifetime
  - Doesn’t go beyond the data value’s boundaries
  - Concurrent accesses are protected
  - …, otherwise, undefined behavior

Memory Safety (II)

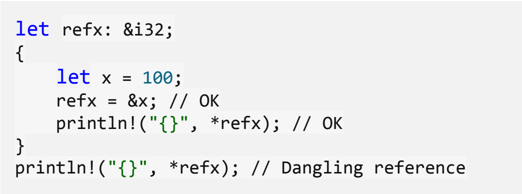
- **Array indexing**
  - Run-time check that index is within the bounds of the array (or slice)

- **Storage overlaying**
  - Rust's *enum* type mechanism adds a tag to indicate the type
  - Run-time check for consistency

```rust
enum E {
    Ptr(Box<i32>),
    Int(i32),
}

let mut e: E;

e = E::Int(100);

match e {
    E::Ptr(p) => println!("{}", *p),
    E::Int(n) => println!("{n}"),

    // other code
}
```

Memory Safety (III)

- **Pointers**
    - No garbage collection in Rust
        - Ownership and lifetime rules enable a simple automatic reclamation policy
    - Safe pointers
        - Check that pointer has a well-defined type
        - Compiler ensures no uninitialized or null pointers
        - Prevents access to dropped value
            - Through conservative compile-time analysis
        - No dangling references
            - A reference is not allowed to outlive its referent
        - Ownership rules
            - Allocated values have unique owners
                - Transfer ownership through allocation and parameter passing
                - Borrow ownership through reference type



```rust
let refx: &i32;
{
    let x = 100;
    refx = &x; // OK
    println!("{}", *refx); // OK
}
println!("{}", *refx); // Dangling reference
```

Concurrency (I)

- **Potential opportunities for violating memory safety**
  - Dangling reference
    - Thread's lifetime exceeds that of a data value that it is accessing
  - Data race / unprotected access
    - One thread is writing to a shared data value while another thread is either reading from or writing to that data
  - Data corruption / aborted update
    - Thread terminates while updating a non-local data value

Concurrency (II)

- **Restrictions on references from threads**
  - Local threads cannot reference outer scopes
  - Scoped threads can borrow reference from outer scope
    - Cannot outlive them
- **Explicit protection with mutexes**
  - Mutex is a wrapper
  - Unlocking is automatic
- **Channels for producers-consumers**
- **Atomic types**

What is needed for certifiable use of Rust?

1. **Engineering Considerations**
    - language and toolchain stability
    - toolchain integrity
    - target and platform support
2. **Support Considerations**
    - availability of professional training
    - long-term maintenance of the toolchain and its supporting tools
    - professional support of the toolchain and its supporting tools
3. **Certification Considerations**
    - qualification of the compilation toolchain
    - certification of the language runtime
    - availability of qualified support tools

Language and Toolchain Stability

Open-Source Rust

- **Six-week release cycle, with**
  - new language version
  - new toolchain version
- **Moves fast intentionally**
  - try out new features
  - keep community energized

Safety-Critical Rust

- **Yearly release cycle**
- **Provides a stable foundation for**
  - long-term development
  - long-term support
  - qualification

Toolchain Integrity

Open-Source Rust

- **Tier 1: “guaranteed to work”***
  - native only
- **Tier 2: “guaranteed to build”**
  - some common cross targets
- **Security working groups**
  - policy and reporting
  - using Rust to write secure software

*this is a community commitment; there’s no warranty

Safety-Critical Rust

- **Provider’s warranty**
  - Service Level Agreement (SLA)
  - for all targets, native & cross
- **Guaranteed supply-chain security**
  - Software Bill of Materials (SBOM)
  - security reporting

Target Support

Open-Source Rust

- **Common native targets**
- **Various embedded / cross targets**
  - of broad interest -or-
  - niche, hacker-friendly

Safety-Critical Rust

- **Common native targets**
- **Relevant embedded / cross targets**
  - bare metal
  - RTOS
  - custom ports as requested
- **Restricted runtimes**

Long-Term Maintenance

Open-Source Rust

- **No commitment to backporting fixes**
- **No LTS version of the language**
- **Follows a Nightly-Beta-Stable paradigm**

Safety-Critical Rust

- **Fixes delivered throughout the year for the current release**
- **Long-Term Support (LTS) available**
  - back-port of fixes to your selected stable branch

Professional Support

Open-Source Rust

- **Best-effort troubleshooting**
  - online forums
  - no guaranteed response time
- **Possible fixes to identified bugs**
  - in a subsequent release
  - if the interest of community maintainers aligns with the problem

Safety-Critical Rust

- **Technical support delivered within deadlines**
- **Review customer ITAR materials if needed** (request specific guidance before sending)
- **Support by toolchain maintainers**
  - offering workarounds -or-
  - bug fixes
- **Predictable integration of bug fixes**

Toolchain Qualification

- **Three significant pieces**
  - cargo: build orchestration
  - rustc: compilation
  - (gcc) ld: linking
- **GNAT Pro for Rust will offer a qualkit covering all three of these**
- **AdaCore has significant experience in toolchain qualification**
  - GNAT Pro qualkits for Ada, C and C++ — (including gcc ld)
  - planned & led the certification activities for the first ISO 26262 **rustc** qual

Runtime Certification

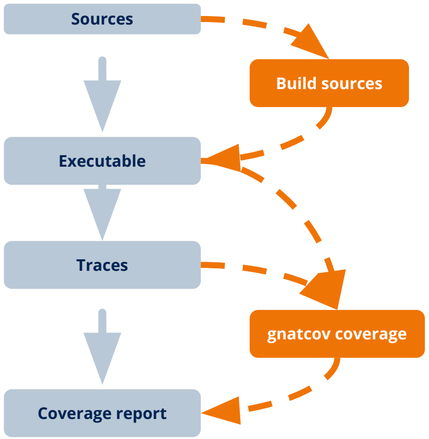
- **Rust cannot be used without its runtime libraries**
  - libcore + liballoc ← much of these are likely required
  - libstd ← significant portions likely to be desired
- **GNAT Pro for Rust will offer certified runtime libraries**
- **AdaCore has extensive experience in runtime certification**
  - GNAT Pro certified runtimes for Ada, C and C++

Support Tools

- **Certifiable use of a language requires qualified support tools**
  - code coverage - including to MC/DC
  - static analysis - including for conformance to coding standards
- **These tools will be available to support GNAT Pro for Rust**
- **AdaCore has extensive experience in building and qualifying support tools**
  - gnatcoverage: qualkits available for Ada and C
  - gnatcheck: qualkits available for Ada

Source Coverage Analysis

- **Instrument generated object code to dump execution traces**
  - Instrument LLVM IR
- **Traces are generated when running the executable(s)**
- **Traces are analyzed and coverage results reported on source code**
- **Working together with the Rust community to have it upstream**



Conclusions

- **Rust is a promising language**
  - Safety and security
  - Performance
  - Community

- **Safety-critical Rust ecosystem is developing**

Rust can be considered as a part of a complete solution for high-performance safety-critical systems

What’s New in GNAT Pro 25.0

Scan to discover what’s new in the 25.0 release

AdaCore

www.adacore.com/hisc-2024