Rust and WebAssembly for fast, secure and reliable software

Urgau & The Rust and WebAssembly Communities

Abstract

Rust is a new systems programming language that promises to overcome the seemingly fundamental tradeoff between high-level safety guarantees and low-level control over resource management. WebAssembly (abbreviated Wasm) is a new binary instruction format for a stack-based virtual machine, it is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications. This paper explore these two new technologies and how they interact with each other. Our findings showed that the combination of those two technologies could revolutionalize the way the entire industry design fast and secure software.

Keywords: Rust; WebAssembly; Realible software; Fast; Secure;

1. Introduction

Systems programming languages like C and C++ give programmers low-level control over resource management at the expense of safety, whereas most other modern languages give programmers safe, high-level abstractions at the expense of control. It has long been a **holy grail** of programming languages research to overcome this seemingly fundamental trade-off and design a language that offers programmers both high-level safety and low-level control. Rust comes closer to achieving this holy grail than any other industrially supported programming language to date, compared to mainstream **safe** languages, Rust offers both lower-level control and stronger safety guarantees [1]. Some safe languages like Java or C# use a virtual machine but unlike WebAssembly, who is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start making it a strong contender for secure and reliable software [2].

2. A Tour of Rust

In this section, we give a brief overview of some of the central features of the Rust programming language. We do not assume the reader has any prior familiarity with Rust.

2.1. Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector.

The rules for ownership are quite simple:

- Data is assigned to a variable.
- The variable becomes the 'owner' of the data.
- There can only be one owner at a time.
- When the owner goes out of scope, the data will be dropped.

Primitive types are popped from stack memory automatically when they go out of scope (e.g. when a function block ends), while complex types must implement a drop function which Rust will call when out of scope (to explicitly deallocate the heap memory).

Those ownership rules are quite different from any other programming languages, like C/C++/Python … where there isn't generally a clearly defined owner of a data and it's up to the developers to know who owns the data.

2.2. Borrowing

The concept of borrowing is designed to make dealing with ownership changes easier. It does this by avoiding the moving of owners, by letting your program provide a 'reference' to the data. This means the receiver of the reference (e.g. a function, struct field or a variable etc) can use the value temporarily without taking ownership of it.

There are two mains rules to the concept of borrowing:

- At any given time, you can have either (but not both of) one mutable reference or any number of immutable references.
- References must always be valid.

Those rules prevents concurrent access to the same underline memory location and ensure that the underline memory location is safe to access at any time.

These effectively makes data races impossible [3].

2.3. Lifetimes

Lifetimes are tightly coupled to 'references'. They are a way for the compiler to know how long a reference lives for, to be sure that any reference that is currently active doesn't refer to data that no longer exists (i.e. a 'dangling pointer').

In the language level lifetimes are just an annotation that has a specific naming convention: '<T> where <T> is a letter like 'a or 'b. The letters don't mean anything special, they're just a way for the user to differentiate them. [4]

The below code example highlights how defining a single lifetime called 'a and assigning it to both arguments (and to the return value) allows the compiler to track these references and ensure they both live long enough to prevent any errors at runtime.

```
fn longest<'a>(x: &'a str, y: &'a str) → &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Code 1: Exemple of usage of lifetimes in a Rust code

2.4. Interior Mutability

Interior mutability is a design pattern in Rust that allows you to mutate data even when there are immutable references to that data; normally, this action is disallowed by the borrowing rules. To mutate data, the pattern uses unsafe code inside a data structure to bend Rust's usual rules that govern mutation and borrowing.

The basic idea behind Interior Mutability is **UnsafeCell [5]** a basic primitive in Rust that allow checks normally done by the borrow checker to be defer (not bypass) to the library code.

RefCell [6] is a type that is based on **UnsafeCell** and do the checks at runtime. This still provides the same safety guarantees as the borrow checker would give at compile-time. This allow developers to still be safe even when outside of the standard borrowing rules.

3. A Tour of WebAssembly

In this section, we give a brief overview of some of the central features of the WebAssembly. We do not assume the reader has any prior familiarity with WebAssembly.

WebAssembly is based on the principle of an stack-based virtual machine. It's an abstraction of a computer, that emulates a real machine. In this case the emulation is based on a stack pointer that follows the instructions of code.

3.1. Modules

A Wasm program is designed to be a separate module containing collections of various Wasm-defined values and program type definitions. Every module is separated from the others modules. This provide native isolation between the code, meaning that the code from a module A cannot access anything from module B that isn't explicitly exported/allowed by the module B. This is particularly useful in our modern world where malware trieds to access memory of a program to give them privileges.

3.2. Linear Memory

The main storage of a WebAssembly program is a large array of bytes referred to as a linear memory or simply memory. This technique is fairly standard and as some advantage as CPU can directly (and linearly) address all of the available memory locations without having to resort to any sort of memory segmentation or paging schemes [7].

3.2.1. Security

The linear memory is disjoint from code space, the execution stack, and the engine's data structures; therefore compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behavior. At worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory [8]. This means that even untrusted modules can be safely executed in the same address space as other code. It also allows a WebAssembly engine to be embedded into any other managed language runtime without violating memory safety, as well as enabling programs with many independent instances with their own memory to exist in the same process.

3.3. Determinism

The design of WebAssembly has sought to provide a portable target for low-level code without sacrificing performance. Where hardware behavior differs it usually is corner cases such as out-of-range shifts, integer divide by zero, overflow or underflow in floating point conversion, and alignment.

WebAssembly gives deterministic semantics to all of these across all hardware with only minimal execution overhead. However, there remain three sources of implementation-dependent behavior that can be viewed as non-determinism [9]:

- NaN Payloads: WebAssembly follows the IEEE-754 standard for floating point arithmetic [10]. However this standard doesn't define the representation of the NaN payload and instead only put rule for what it cannot be. This leads to CPUs handling and representing NaN payload differently.
- Resource Exhaustion: Available resources are always finite and differ wildly across devices. An "out of memory" error is unpredictable and therefore nondeterministic.
- Threads: Executing concurrently some code is highly dependent on the CPU Scheduler and subject to difference between runs as the CPU as different schedule profile [11].

Nevertheless these nondeterminism are limited and local, meaning that WebAssembly is as deterministic or even better than every other programs.

4. Memory Safety

4.1. In Rust

As we saw in A tour of Rust 1, Rust has many features to make the code safe but how do these features ensure memory safety? Let's take a look at a few examples of programs that should be memory unsafe and see how Rust identifies their errors.

4.1.1. Dangling pointers

```
fn foo() -> &i32 {
  let n = 0;
  &n
}
```

Code 2: Example of a Rust code creating a local variable and returning a reference from it

```
error[E0106]: missing lifetime specifier
 --> test.rs:1:13
  |
1 | fn foo() -> &i32 {
  |             ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for
it to be borrowed from
  = help: consider giving it a 'static lifetime
```

Text 3: Error message from the compiler about Code 2 [4]

Here, we attempt to return a pointer to a value owned by n, where the pointer would nominally live longer than the owner. Rust identifies that it's impossible to return a pointer to something without a "value for it to be borrowed from."

4.1.2. Use-after-free

```
use std::mem::drop; // equivalent to free()

fn main() {
  let x = "Hello".to_string();
  drop(x);
  println!("{}", x);
}
```

Code 4: Example of a Rust code who create a variable and then pass it to the drop function

```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |        - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
  |
  = note: move occurs because `x` has type `std::string::String`, which does not
implement the `Copy` trait
```

Text 5: Error message from the compiler about the Code 4 [4]

Due to Rust's ownership semantics, when we free a value, we relinquish ownership on it, which means subsequent attempts to use the value are no longer valid. This also protects against double frees, since two calls to drop would encounter a similar ownership type error.

4.1.3. Iterator invalidation

```rust
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
  for elem in from.iter() {
    to.push(*elem);
  }
}

fn main() {
  let mut vec = Vec::new();
  push_all(&vec, &mut vec);
}
```

Code 6: Example of a Rust code than tries to borrow the same container multiples times

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
 -→ test.rs:9:23
  |
9 |   push_all(&vec, &mut vec);
  |            ---       ^^^- immutable borrow ends here
  |            |         |
  |            |         mutable borrow occurs here
  |            immutable borrow occurs here

error: aborting due to previous error
```

Text 7: Error message for the Code 6 [5]

Because Rust enforces uniqueness of mutable borrows, it's impossible to accidentally cause an iterator invalidation above where the **from.iter()** would become invalid after pushing to to when from == to.

4.1.4. Malloc bugs

It's worth noting that because Rust automatically inserts allocs/frees, the compiler will always get the size of the allocation correct, and always free the memory at some point [12].

4.2. In WebAssembly

4.2.1. Memory isolation

The Memory Isolation API provides facilities to protect against shared memory writes between two modules with different permissions (e.g., one module can read and change data while another cannot) [13].

This protects against vulnerabilities such as dangling pointers where an attacker might use knowledge of a pointer location from one context to attack it in another.

The Memory Isolation API takes advantage of type system features available only when compiling to WebAssembly, such as linear types, which provide transparent compile-time enforcement at runtime.

4.2.2. Control-Flow Integrity

Within WebAssembly, Control-flow integrity (CFI) is enforced by the Memory Safety API. CFI protects against code-reuse attacks, such as Return Oriented Programming (ROP) [14]. The efficacy of control-flow integrity can be measured based on its completion. Generally, three external control-flow transitions need to be safeguarded because the caller may not be trusted. Those are direct function calls, indirect function calls, and returns.

To ensure memory safety, WebAssembly will use an expected control flow graph when it is compiled. Vectors of computer-readable instructions can be made safe by inserting runtime instrumentation at every call site to verify that the transition is safe.

5. Users Safety

Each WebAssembly module executes within a sand-boxed environment separated from the host runtime using fault isolation techniques. This implies:

- Applications execute independently, and can't escape the sandbox without going through appropriate APIs.
- Applications generally execute deterministically with limited exceptions.

Additionally, each module is subject to the security policies of its embedding making every WebAssembly module much more safe than any non sandbox program (ie. every program).

Coupled with the security of Rust, users have increased security of their programs.

6. Developer Safety

rustc (the Rust official compiler) won't compile programs that attempt unsafe memory usage. Most memory errors are discovered when a program is running. Rust's syntax and language metaphors ensure that common memory-related problems in other languages null or dangling pointers, data races, and so on—never make it into production. This makes the developers more confident about the safety and reliability of their code [15].

As for WebAssembly, the determinism of it makes developer also more confident and more productive as they don't have to worry about non-determinism and platform specific methods/conversions or APIs.

And because WebAssembly methods are typed misuse of them (eg passing a short instead of an int) is impossible reafirming the safety to the developers.

7. Speed

As stated previous WebAssembly is based on the principle of an stack-based virtual machine. This technique isn't new and in fact is used in many implementation like the JVM (Java Virtual Machine) or by the CLR (.NET Common Language). These implementation of an stack-based virtual machine proves to be extremely high performant with only a small performance penalty [16]. However unlike most JVM or CLR implementation many WebAssembly interpreter has also a compiler mode to compile WebAssembly to the native binary format saving the cost of the interpretation and allowing many optimizations making these binary close to native speed.

Rust is based on the now robust LLVM project a backend for compiler optimization, it's most famous frontend is clang a C/C++ compiler. LLVM caries many optimization accumulated over more than 2 decades. Studies as found that in general the same C/C++ program compiled with GCC and Clang only as a +5% percent difference in performance [17]. This greatly helps Rust achieve the same or better performance than an equivalent C++ program compiled with Clang.

Note that there is ongoing work to provide GCC as an alternative backend for the Rust compiler. This could potentially make Rust programs even faster than with LLVM [18].

8. Conclusions

Rust and WebAssembly have both been designed with security and reliability in mind making the combinations of them trough their features like borrow checking and ownership for Rust and modularization and determinism in WebAssembly makes them as or even more (data-races, invalidation, …) secure than any other alternative like C/C++ and JVM or CLR.NET even trough most of them can be compiled to WebAssembly. The speed of WebAssembly and Rust also greatly improve their possible massive adoption as a secure combination for fast, secure and reliable software. We expect to see in the future a massive increase in the usage of those two technologies. Future work could try to determine if the security of these technologies is a decisive favor of adoptions.

Acknowledgments

We wish to thank the Rust and the WebAssembly communities in general and every peoples and organizations listed:

- Mozilla
- The Rust
- The WebAssembly Communities
- Ralf Jung
- ByteCode Alliance

References

1: RALF JUNG, JACQUES-HENRI JOURDAN, ROBBERT KREBBERS, DEREK DREYER, RustBelt: Securing the Foundations of the Rust, 2018, https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf
2: The WebAssembly Contributors, , 2022, https://webassembly.org/docs/non-web/
3: The Rust Contributors, References and Borrowing, 2022, https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#mutable-references
4: The Rust Contributors, Lifetime Syntax, 2022, https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html
5: The Rust Contributors, UnsafeCell, 2022, https://doc.rust-lang.org/core/cell/struct.UnsafeCell.html
6: The Rust Contributors, RefCell, 2022, https://doc.rust-lang.org/std/cell/struct.RefCell.html
7: radu, https://radu-matei.com/blog/practical-guide-to-wasm-memory/, 2021, https://radu-matei.com/blog/practical-guide-to-wasm-memory/#webassembly-memory
8: Nick Lewycky, Why is WebAssembly safe and what is linear memory model, 2021, https://stackoverflow.com/a/65933986
9: The WebAssembly contributors, Nondeterminism in WebAssembly, , https://github.com/WebAssembly/design/blob/main/Nondeterminism.md
10: IEEE, IEEE-754 Standard, 2019, https://standards.ieee.org/ieee/754/6210/
11: Sun Microsystems, Inc, Thread Scheduling, 1995-2005, https://www.iitk.ac.in/esc101/05Aug/tutorial/essential/threads/priority.html
12: The Rust Contributors, Destructors, , https://doc.rust-lang.org/reference/destructors.html#destructors
13: The WebAssembly Contributors, WebAssembly - Security, , https://webassembly.org/docs/security/
14: The WebAssembly Contributors, Control Flow Integrity, , https://webassembly.org/docs/security/#control-flow-integrity
15: Ryan Donovan, Why the developers who use Rust love it so much, , https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/
16: Eric Lippert, Why have a stack?, 2008, https://web.archive.org/web/20120328082950/http://blogs.msdn.com/b/ericlippert/archive/2011/11/28/why-have-a-stack.aspx
17: kornel.ski, Speed of Rust vs C, 2021, https://kornel.ski/rust-c-speed
18: rust-lang, antoyo, rustc_codegen_gcc, 2020, https://github.com/rust-lang/rustc_codegen_gcc