WASM Interpreter for Safety – White Paper

Wanja Zaeske
Department Safety Critical Systems & Systems Engineering
German Aerospace Center (DLR)
wanja.zaeske@dlr.de

1. Introduction

This white paper provides an overview over our WebAssembly interpreter [1]. It Primarily serves to highlight and explain design decisions and the goals we aim to achieve.

The development of this project currently takes place as a joint effort between DLR-ft [2] and OxidOS [3]. Being truly Open Source, the project already is willing to accept external contributions. As the project matures, it might be worth considering moving the project to a vendor-agnostic foundation, to assure for a fair process allowing contributions from multiple commercial entities. However, for the time being the project is not mature enough to justify the overhead doing so, thus for the time being the project remains under control of DLR & OxidOS.

1.1. Design Drivers

The implementation of our interpreter is driven by several design goals. After all, there are dozens of WebAssembly interpreters around, why would one have to implement yet another one?

1.1.1. Pure Interpretation

In fact, the prior sentence is somewhat imprecise. Most WebAssembly interpreters are actually compilers; they compile the WebAssembly down to a different format that seems more suitable for execution. A prominent example would be WasmTime [4], which compiles WebAssembly down to object code, so called Ahead-of-time (AOT) compilation. Browsers commonly mix in Just-in-time (JIT) compilation as well, starting with a quick-to-compile but not so fast-to-run AOT baseline build, which then on demand is partially swapped for optimized sections obtained by JIT compiling code sections in the hot path (example: [5]).

But even if neither AOT nor JIT are at hand (no object code for the physical processor is created), there still is a common pattern of rewriting the WebAssembly (WASM) bytecode into a custom Intermediate Representation (IR) (example: [6]). There are multiple reasons for this, this allows for gentle optimizations, simpler code, and foremost explicit jumps. In ordinary WASM bytecode, branches are indirect; e.g. a branch could be “jump out of the innermost three scopes”. In practical terms, a branch boils down to modifying the program counter/instruction pointer, i.e. advance it 15 instructions forward. To avoid costly linear search at runtime (for where the innermost three scopes end), many interpreters rewrite the WASM bytecode into their own IR, calculating the direct offsets on the instruction pointer for the branches once, so that in their IR branches are direct.

This is a good decision for many use-cases, but tricky when safety-certification comes into play. Now, the currency in safety certification is evidence, and for the special case of avionics, a significant amount of evidence has to be provided for the object code [7]. Thus, generating object code at run-time (for example when JIT compiling) is prohibitive – it simply does not fit together with the assumptions baked into current certification guidelines such as DO-178C [8]. Finally, DO-332 contains specific language that allows for the directly interpreted format executed by a virtual machine/bytecode interpreter to be treated as object code. Hence, when compiling WASM bytecode into a custom IR, one would have to provide certification evidence on that IR.

Our design alleviates all this; by directly interpreting the WASM bytecode, certification evidence has to be produced for the WASM directly, and thus is not tied to implementation details (as in particular IR) used by our interpreter. A comprehensive discussion of this can be found in our publication [7].

To avoid the problem of indirect branching, we borrow ideas from Ben L. Titzer's wizard [9], a pre-computed side-table that for all branches stores the direct offset on the instruction pointer. A detailed discussion of the technique and further implementation details of wizard can be found in his paper [10].

1.1.2. Certification Friendly Source Code

Our implementation is written in Rust, which on its own poses a challenge: As of now, we are not aware of any high-assurance deployment of Rust in the aviation sector. Similarly, there is only little information on Rust being deployed in automotive (such as [11]). At the same time, there is a keen interest to push Rust in safety critical domains, and companies like ferrous systems [12] with their ISO 26262 ASIL D certified Rust compiler [13] and AdaCore with the GNAT Pro for Rust [14] pave the way for Rust.

Now, there some techniques often found in Rust programs pose risk for a smooth certification. One such thing would be macros. While we are not aware of a precedence case, we assume that Rust's various flavors of macros might be treated like tools, after all they are computer programs that generated code, which in term is compiled in to the final application. As such, they might be subject to tool qualification as per DO-330 [15], and in since macros can easily sneak in broken code, they are likely to be treated as criteria 1 tool [15]. If these assumptions hold, macros would have to be tool qualified to the matching Tool Qualification Level (TQL) of an application's Design Assurance Level (DAL). As testing macros is more complicated than testing normal code, we restrict our usage of macros to the bare minimum.

Another risk to certification is third party code. Thus, to keep our code closure (and subsequently the Software Bill of Materials (SBOM) compact), we refrain from using dependencies which get compiled into the code[1].

Bibliography

[1] DLR e. V. and OxidOS, "wasm-interpreter." [Online]. Available: https://github.com/DLR-FT/wasm-interpreter

[2] DLR e. V., "DLR | Institute of Flight Systems." [Online]. Available: https://www.dlr.de/en/ft

[3] OxidOS, "Welcome to OxidOS Automotive." [Online]. Available: https://oxidos.io/

[4] Bytecode Alliance, "A fast and secuire runtime for Webassembly." [Online]. Available: https://wasmtime.dev/

[5] Lin Clark, "Making WebAssembly even faster: Firefox's new streaming and tiering compiler." [Online]. Available: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/

[6] Robin Freyler et al., "Making WebAssembly even faster: Firefox's new streaming and tiering compiler." [Online]. Available: https://docs.rs/wasmi_ir/latest/wasmi_ir/enum.Instruction.html

[7] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "WebAssembly in Avionics : Decoupling Software from Hardware," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023, pp. 1–10. doi: 10.1109/DASC58513.2023.10311207.

[1] Currently, two carefully selected run-time dependencies are allowed, however, there is a roadmap to phase them out, and each such exception is tracked in our requirements

[8] RTCA, “DO-178C Software Considerations in Airborne Systems and Equipment Certification,” 2011.

[9] Ben L. Titzer et al., “The Wizard Research Engine.” [Online]. Available: https://github.com/titzer/wizard-engine

[10] B. L. Titzer, “A fast in-place interpreter for WebAssembly,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563311.

[11] Dion Dokte and Julius Gustavsson, “Rust is rolling off the Volvo assembly line.” [Online]. Available: https://tweedegolf.nl/en/blog/137/rust-is-rolling-off-the-volvo-assembly-line

[12] “Ferrous Systems.” [Online]. Available: https://ferrous-systems.com/

[13] “This is Rust for critical systems..” [Online]. Available: https://ferrocene.dev/

[14] “GNAT Pro for Rust.” [Online]. Available: https://www.adacore.com/gnatpro-rust

[15] RTCA, “DO-330 Software Tool Qualification Considerations,” 2011.