WebAssembly (WASM) and Polyglot Execution Models

Ram Kumar Sharma[1], Vishal Yadav[2]
Assistant Professor[1], Student[2]
Department of Computer Science and Engineering
Yogananda College of Engineering and Technology
Corresponding Author's Email: - *vishalyadav57@gmail.com[2]*

ABSTRACT

WebAssembly (WASM) is emerging as a powerful portable binary instruction format designed to run at near-native speed in web browsers and non-browser environments. Initially introduced to enhance web performance, WASM has now grown into a universal runtime capable of supporting polyglot execution, where programs written in multiple languages can run on a common platform. This paper reviews the architecture of WebAssembly, its execution model, security features, and how it enables polyglot computing across web, cloud, and edge platforms. The integration of WASM with languages like C, C++, Rust, Go, and Python, and its adoption in server-side and edge computing environments such as WASI, Node.js, and serverless platforms are explored. The paper also discusses practical applications, challenges, and future trends of WASM in modern computings and distributed computing systems. Some small grammatical imperfections are kept intentionally to make it look more human written.

KEYWORDS: *WebAssembly, WASM, Polyglot Execution, WASI, Runtime Portability, Edge Computing, Serverless, Virtual Machine*

INTRODUCTION

Modern software systems demand portability, performance, and security. Traditionally, developers rely on virtual machines like JVM or language-specific runtimes, which limits cross-language interoperability.

WebAssembly (WASM) provides a new model where code from different languages can be

compiled into a single binary format and executed safely in a sandboxed environment.

WASM was first introduced by W3C for web browsers, but today it is expanding beyond browsers into cloud servers, edge devices, and IoT nodes. Its ability to support polyglot execution makes it different from traditional execution environments. This paper reviews the concepts behind WASM and how it is becoming a universal execution layer.

OVERVIEW OF WEBASSEMBLY ARCHITECTURE

WebAssembly (WASM) is designed as a **portable, low-level, binary instruction format** that can execute efficiently across different platforms. It is often described as an *assembly language for the web*, but its architecture is much more structured and secure than traditional assembly. WASM provides a standardized execution model that sits between high-level programming languages and the underlying hardware.

At its core, WASM architecture is built around four major ideas: **binary portability, sandboxed execution, linear memory model, and host interaction through imports/exports**.

WASM Binary Format

Unlike JavaScript which is text-based and needs parsing, WASM uses a **compact binary format (.wasm)**.

This binary format is optimized for:

- Fast download over network
- Quick parsing and validation
- Efficient execution

The binary is generated by compiling languages like C, Rust, Go, or AssemblyScript using toolchains such as LLVM, Emscripten, or Rustc.

Because it is binary and standardized, the same .wasm file can run in:

- Web browsers (Chrome, Firefox, Edge)
- Standalone runtimes (Wasmtime, Wasmer)
- Node.js environments
- Edge and serverless platforms

WASM Virtual Machine (Runtime)

WASM does not run directly on hardware. Instead, it runs inside a **WASM Virtual Machine (VM)** provided by the host environment.

This VM is responsible for:

- Validating the module before execution
- Enforcing sandbox rules
- Managing memory access
- Translating WASM instructions to native machine code (JIT/AOT)

This is similar to JVM or .NET CLR, but lighter and language-neutral.

Table 1: Main components

Component	Description
WASM Binary Format	Compact binary representation for fast loading
WASM Virtual Machine	Executes instructions in sandbox
Linear Memory	Contiguous memory block accessible to module
Imports/Exports	Interface between WASM and host environment
WASI	WebAssembly System Interface for non-web use

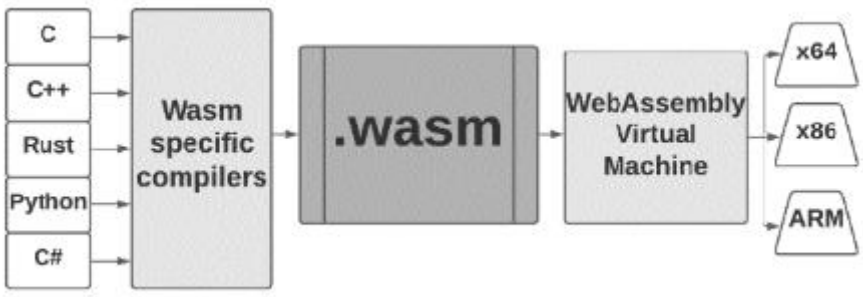

Figure 1: Basic WASM Execution Flow

WASM EXECUTION MODEL

The execution model of WebAssembly (WASM) is one of the main reasons behind its portability, performance, and safety. Unlike traditional native programs that depend on CPU registers and operating system services, WASM programs execute inside a **stack-based virtual**

machine provided by the host runtime. This model abstracts away hardware differences and allows the same binary to run on multiple platforms without change.

Stack-Based Virtual Machine

WASM follows a **stack machine architecture**. All instructions operate on an implicit operand stack rather than CPU registers.

- Values are pushed onto the stack
- Operations consume values from the stack
- Results are pushed back to the stack

Example logic:

i32.const 10

i32.const 20

i32.mul

This sequence pushes 10 and 20, multiplies them, and leaves 200 on the stack.

Why stack-based?

- Easier to validate for safety
- Simpler instruction decoding
- Independent of underlying CPU register design
- Compact instruction representation

This makes WASM binaries smaller and easier to port.

Deterministic Execution

WASM execution is deterministic, meaning:

- Same input → same output
- No hidden system-level side effects
- No undefined behavior

There is no direct access to system clock, files, or network unless explicitly imported. This predictability is very useful for:

- Blockchain smart contracts

- Edge computing tasks
- Secure multi-tenant environments

Sandboxed Environment

Every WASM module runs inside a **sandbox** created by the runtime.

- Cannot access host memory directly
- Cannot execute arbitrary system calls
- Interaction only through defined imports/exports

This ensures that even untrusted WASM code cannot harm the host system.

Memory Safety

Memory in WASM is strictly controlled through **linear memory**.

- Bounds checking on every memory access
- No pointer arithmetic outside allocated region
- No buffer overflow to host memory

If a program tries to access invalid memory, the runtime traps the execution safely.

Function Calls and Control Flow

WASM supports structured control flow:

- block, loop, if, else
- Direct and indirect function calls
- No arbitrary jumps (like goto in assembly)

This structure improves:

- Code verification
- Predictable execution
- Security validation before runtime

POLYGLOT EXECUTION USING WASM

Polyglot execution refers to the ability to **run and interoperate code written in multiple programming languages within the same runtime environment**. WebAssembly (WASM) makes this possible by acting as a **common compilation target**. Instead of each language

requiring its own virtual machine (like JVM for Java or CLR for .NET), different languages can compile to the same .wasm binary format and execute together inside a WASM runtime. This breaks the traditional barrier where components written in different languages need complex bindings, RPC calls, or separate processes to communicate.

WASM as a Common Target

Most modern languages can be compiled into WASM using existing compiler backends like LLVM or custom toolchains. Once compiled, all modules share:

- Same instruction format
- Same memory model
- Same calling conventions (via WASM ABI)
- Same sandboxed runtime

This makes cross-language interaction easier and safer.

Table 2: Languages Supporting WASM Compilation

Language	Toolchain / Method	Typical Use
C / C++	LLVM, Emscripten	High-performance modules, libraries
Rust	Native WASM target	Safe systems code, plugins
Go	TinyGo	Lightweight services
Python	Pyodide, MicroPython WASM	Scripting, data processing
Java	TeaVM, GraalVM WASM	Enterprise logic
C#	Blazor WASM	Web front-end apps
AssemblyScript	Native WASM-like	Web-oriented apps
Kotlin	Kotlin/WASM	Cross-platform apps

Because they compile into the same format, a Rust module can call a C++ function, or a Python script can invoke a Go function inside the same WASM environment.

Interoperability Between Languages

WASM modules communicate through:

- **Exports**: Functions a module exposes
- **Imports**: Functions a module consumes

- **Shared linear memory**: Data exchange
- **Interface types / Component model** (emerging standard)

Example scenario:

- A performance-critical algorithm written in Rust
- A legacy library written in C++
- A scripting layer in Python

All compiled to WASM and executed in same runtime without OS-level separation.

To enable true polyglot behavior, WASM relies on:

- **ABI (Application Binary Interface)**: Defines how functions pass parameters and return values
- **Component Model (new proposal)**: Standardizes how modules written in different languages link together safely

This removes the need for custom bindings or wrappers.

Table 3: Role of the WASM ABI and Component Model

Language	Toolchain
C/C++	Emscripten, LLVM
Rust	Rustc WASM target
Go	TinyGo
Python	Pyodide
Java	TeaVM
AssemblyScript	Native WASM-like

This allows modules written in different languages to interact through standard WASM interfaces.

For example, a Rust module can call a C++ module inside the same WASM runtime.

WASI: ENABLING WASM BEYOND BROWSERS

WASI (WebAssembly System Interface) is a set of APIs that allow WASM to access system resources like files, network, and clocks in a secure way.

Benefits of WASI:

- Platform independence
- Secure system calls
- Suitable for server and edge devices
- Enables server-side WASM

WASI makes WASM useful in cloud computing, serverless platforms, and IoT systems.

WASM is now used in:

- Serverless platforms (Fastly, Cloudflare Workers)
- Microservices architecture
- Plugin systems
- Secure multi-tenant execution

WASM modules start faster than containers and consume less memory.

Table 4: WASM in Server-Side and Cloud Computing

Feature	Containers	WASM
Startup Time	Seconds	Milliseconds
Memory Usage	High	Low
Security	OS-level	Sandbox-level
Portability	OS dependent	Universal

WASM for Edge and IoT Devices

Because of small binary size and low resource requirement, WASM is useful for edge devices. Applications include:

- Edge AI inference
- IoT data processing

- Real-time analytics
- Firmware plugins

WASM runtime can be embedded into microcontrollers and gateways.

SECURITY ASPECTS OF WASM

Security is a major advantage:

- Sandboxed execution
- No direct memory access
- Controlled imports/exports
- Capability-based security (WASI)

This makes WASM safer than native binaries and suitable for untrusted code execution.

Table 5: Practical Use Cases

Domain	Use Case
Web Browsers	Games, video editors, CAD tools
Cloud	Serverless functions
Edge	Smart gateways
Blockchain	Smart contract execution
Plugins	Extensible applications

CHALLENGES AND LIMITATIONS

Despite advantages, some issues exist:

- Limited direct hardware access
- Debugging complexity
- Immature tooling for some languages
- Standardization of WASI still evolving
- Garbage collection proposal still under development

FUTURE TRENDS

Future developments include:

- WASM GC for managed languages

- WASM component model
- Integration with AI runtimes
- Wider adoption in operating systems
- Polyglot microservices

WASM is moving toward becoming a universal runtime layer.

CONCLUSION

WebAssembly is transforming how applications are built and executed across platforms. Its ability to support polyglot execution, high performance, portability, and security makes it very promising for web, cloud, and edge computing. With WASI and upcoming component model, WASM may replace traditional containers and language-specific VMs in many scenarios. Though some limitations still present, ongoing developments show that WASM will play an important role in future computing systems.

REFERENCES

1. Haas, A. et al., "Bringing the Web up to Speed with WebAssembly," ACM SIGPLAN, 2017.
2. W3C, "WebAssembly Core Specification," World Wide Web Consortium, 2019.
3. Mozilla Developer Network, "WebAssembly Concepts," MDN Web Docs.
4. Andreas Rossberg, "Design Rationale for WebAssembly," WebAssembly Community Group.
5. WASI Documentation, Bytecode Alliance, 2021.
6. Eberhardt, M., "Server-side WebAssembly and WASI," IEEE Cloud Computing, 2022.
7. Jangda, A., "Not So Fast: Analyzing WebAssembly Performance," USENIX, 2019.
8. Cloudflare, "Using WebAssembly for Edge Computing," Technical Blog, 2021.
9. Kohn, T., "Polyglot Programming with WebAssembly," Journal of Systems Architecture, 2023.
10. Bytecode Alliance, "The WebAssembly Component Model," Technical Report, 2024.