A Survey of WebAssembly Usage for Embedded Applications: Safety and Portability Considerations

Tancredi Orlando[1,*,†], Luca D'Agati[1,†], Francesco Longo[1,2,†] and Giovanni Merlino[1,2,†]

[1] *University of Messina, Department of Engineering, Contrada di Dio, 98166 Sant'Agata, Messina, Italy*
[2] *National Interuniversity Consortium for Informatics (CINI), Via Ariosto 25, 00185 Rome, Italy*

Abstract

WebAssembly is the standard technology to accelerate web-based applications that need to perform compute-intensive workloads. Lower-level languages such as C, C++, and Rust can be used to implement functionalities that are executed by user agents, providing an alternative to JavaScript for performance-critical tasks. In this domain, safety and portability are key aspects: the former is essential to protect users from malicious modules while the latter enables having the exact same WebAssembly code to run seamlessly across browsers, operating systems and hardware architectures.

The safety and portability features of WebAssembly have driven an increasing interest in employing it outside of web-enabled environments and now it is possible to use WebAssembly as a compilation target for applications that can be executed in diverse environments such as desktops, servers, cloud and embedded systems. The conciseness and the generality of the specification have made it possible to use the exact same technology unchanged across heterogeneous devices.

This paper explores the current state of the WebAssembly ecosystem, focusing on its potential applicability in embedded environments, where hardware lacks security features such as memory isolation, and portability is generally hard to achieve due to the heterogeneity of the devices.

Keywords

WebAssembly, Embedded Systems, Safety, Portability, Virtual Machines, WASI

1. Introduction

The swift expansion of the Internet of Things (IoT) and advancements in embedded devices have underscored the growing need for secure and portable software solutions across diverse environments. By 2030, the number of embedded systems worldwide is expected to surpass 25 billion [1], reflecting their critical role in industries ranging from healthcare to manufacturing. Despite their widespread adoption, the development of embedded systems remains fragmented, largely due to the diversity of hardware architectures and the lack of standardized software interfaces.

WebAssembly (Wasm), originally designed to enhance the performance of Web applications, has emerged as a promising technology to address the challenges of a fragmented compute ecosystem. Its portable binary instruction format and sandboxed execution model make it a valid option for devices with heterogeneous architectures that require a robust security model. By abstracting platform-specific details through a virtual machine, Wasm enables applications to run consistently across different environments, from browsers to standalone runtimes. However, the transition of Wasm from Web environments to embedded systems introduces new challenges, particularly in achieving seamless cross-platform portability and ensuring safety in resource-constrained devices.

This research aims to explore the benefits and challenges of adopting Wasm in embedded environments, focusing on two critical aspects: safety and portability. Specifically, it investigates how Wasm's safety features can enhance memory isolation in systems lacking hardware memory protection

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
[*]Corresponding author.
[†]These authors contributed equally.
tancredi.orlando@studenti.unime.it (T. Orlando); ldagati@unime.it (L. D'Agati); flongo@unime.it (F. Longo); gmerlino@unime.it (G. Merlino)
0009-0004-2644-0028 (T. Orlando); 0000-0002-9992-4179 (L. D'Agati); 0000-0001-6299-140X (F. Longo); 0000-0002-1469-7860 (G. Merlino)

and evaluates whether its portability promises hold in embedded contexts. Furthermore, it addresses challenges such as the lack of native support for access to hardware resources and proposes potential solutions, including Hardware Abstraction Layers (HALs).

To address the Wasm portability challenges outside of Web environments, the WebAssembly System Interface (WASI) was introduced as an extension to enhance Wasm's applicability beyond the browser. WASI defines a standardized set of APIs for system-level operations, such as file I/O, networking, and time management, aiming to provide a consistent interface for Wasm applications across cloud, desktop, and edge environments. Despite its potential, the applicability of WASI to embedded systems remains limited. Embedded devices often lack full-fledged operating systems and require low-level access to hardware resources, such as GPIOs and peripherals, which WASI does not natively support.

The remainder of this paper is structured as follows. Section 2 analyzes the core principles of Wasm, focusing on its safety mechanisms and portability features. Section 3 examines the unique challenges faced by embedded systems, with a focus on security and cross-platform execution. Section 4 investigates the integration of Wasm in embedded systems, analyzing runtime implementations and emerging solutions for hardware abstraction. Section 5 discusses open challenges in the evolution of Wasm, particularly in the context of embedded applications.

2. WebAssembly: Safety and Portability Features

Wasm combines robust safety guarantees with broad portability features, making it well-suited for heterogeneous computing environments. Its sandboxed execution model ensures that code runs securely within strict boundaries, while its platform-agnostic design enables consistent execution across heterogeneous hardware architectures and operating systems. These complementary attributes form the foundation of WebAssembly's design philosophy, addressing security and cross-platform compatibility as core concerns rather than afterthoughts.

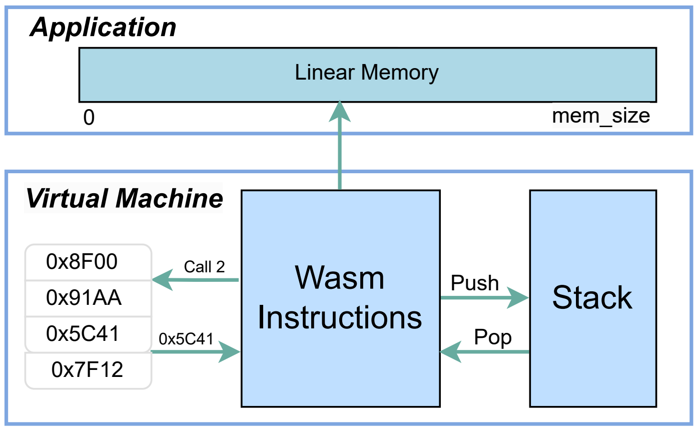
The fundamental deployment and execution unit is a Wasm module, that can be represented in both a human-readable text format (WAT) [2] and a compact binary format (WASM) [3], with the latter being the one executed by Wasm runtimes. Modules structure resemble an ELF file, with distinct sections for different types of content. These sections include type definitions, function declarations, tables for indirect function calls, linear memory specifications, global variables, and export declarations. The explicit boundary between the module and its host environment is maintained through the import/export mechanism, where modules explicitly declare functions they require from the host (imports) and functions they make available to the host (exports). This design allows for predictable interactions while preserving the security guarantees of the execution model. Prior to execution, each module undergoes validation to ensure conformance with the Wasm specification, including type-checking, providing static guarantees that complement runtime safety mechanisms.

Figure 1 illustrates how Wasm operates, highlighting both its internal memory model and its integration within a broader application environment [4]. It shows how Wasm executes code and interacts with other system components.

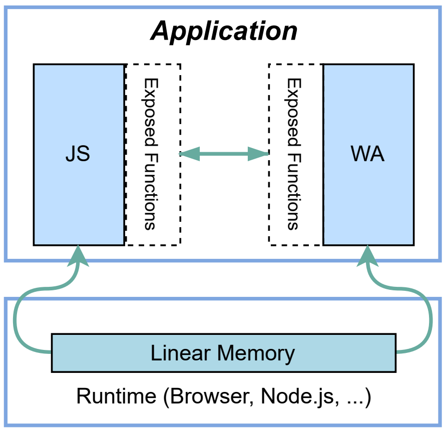
The Wasm memory model, shown in Figure 1a, consists of distinct memory regions. The operand stack is explicitly managed using Wasm instructions and serves as temporary storage for intermediate computations. Most Wasm instructions interact with this stack in some way. For example, instructions such as `local.get` and `i32.const` push values onto the stack. Arithmetic and logical instructions (`i32.add`, `f64.mul`, `i32.and`) take their operands from the stack and push the result back onto it.

Separately, Wasm maintains an internal shadow stack, which is managed by the runtime and used to track function calls, return addresses, and local variables. Unlike the operand stack, the shadow stack is not accessible to Wasm code, ensuring control-flow integrity by preventing direct manipulation of return addresses.

The linear memory is a contiguous, byte-addressable region that is entirely separate from the operand stack and serves as the primary data storage for Wasm programs. Unlike the operand stack, which is automatically managed, linear memory must be explicitly allocated and resized by the Wasm module



(a) Diagram of the WASM memory model illustrating the interactions between the operand stack, linear memory, and instruction execution. The operand stack handles intermediate computations, the linear memory serves as the primary data storage, and the instruction execution manages operations on these components.



(b) Architecture of a typical WASM application demonstrating the integration between WASM modules and JavaScript within a managed execution environment, such as a browser or Node.js.

Figure 1: Comprehensive overview of the WASM execution model, encompassing both the internal memory architecture and its integration within a broader application environment. Figure 1a details the memory model, while figure 1b depicts the interaction between WASM modules and JavaScript [4].

using instructions such as `memory.grow`. Instructions like `i32.load`, `i32.store`, and `memory.fill` allow WASM modules to read from and write to linear memory, providing a structured approach to data storage.

Figure 1b illustrates the design of a typical application integrating WASM with JavaScript. The WASM module runs within a managed execution environment, such as a browser or Node.js, where it interacts with JavaScript through function calls and a shared linear memory space. While this shared memory enables efficient data exchange between WASM and JavaScript, it also requires careful management to prevent unintended access or memory safety violations. The interaction between the two is strictly controlled, with WASM exposing only explicitly defined functions, reducing the risk of unauthorized memory access or corruption, as the shared memory is separated from the host's memory. This mechanism allows WASM to integrate with Web APIs, event-driven architectures, and external resources while preserving the security guarantees of the execution environment.

This architecture ensures a balance between performance and safety. The sandboxed design isolates WASM modules while enabling efficient communication with host environments through shared linear memory and exposed function interfaces.

2.1. Safety

WASM's security model is designed with a multi-layered approach that protects users from potentially buggy or malicious modules while providing developers with powerful primitives for creating secure applications. At its core, WASM represents a carefully designed typed assembly language [5] that enforces strict safety guarantees through a combination of static validation and runtime mechanisms.

Sandboxing WASM modules operate within a sandboxed environment that employs fault isolation techniques to prevent applications from directly interacting with the host system. This isolation ensures that modules execute independently and cannot access system resources without going through controlled APIs [6]. For instance, in web browsers, WASM modules are isolated from the Document Object Model (DOM) [7] and other browser and operating system resources, ensuring that untrusted programs cannot manipulate sensitive user data or execute arbitrary commands. This principle extends to other environments such as serverless computing platforms, where secure execution of untrusted code is essential to prevent cross-function interference or system breaches.

Static Validation and Type Safety Before execution, the WASM modules undergo a comprehensive validation to ensure conformity with the WASM specification. The specification explicitly requires that all WASM code must be validated to guarantee safe execution [8]. WASM modules must explicitly declare all accessible functions along with their associated types, ensuring that functions and their parameters are well-defined and type-checked before execution. Unlike most native architectures, WASM enforces a strong typing system for all globals, locals, and function arguments and results. This static type-checking prevents type confusion vulnerabilities and ensures the integrity of function calls [9].

Separation of Memory Spaces WASM enforces a strict separation between the operand stack (used for function execution) and linear memory (used for general-purpose storage). The operand stack is used exclusively for temporary computation and is not directly addressable by WASM code, making it immune to direct corruption. This prevents common exploit techniques that rely on overwriting control data, such as stack-smashing attacks. In contrast, linear memory is mutable and accessible to WASM code, making it a potential target for memory corruption vulnerabilities. Although WASM enforces bounds checking to prevent out-of-bounds reads and writes, vulnerabilities such as buffer overflows, use-after-free, and data corruption within the allocated memory region remain possible [9]. Developers must carefully manage memory operations to prevent unintended modification of data structures or security-sensitive information.

Data Execution Prevention WASM enforces Data Execution Prevention (DEP) by strictly separating executable code from writable memory. The linear memory space is never executable, preventing code injection attacks. Additionally, WASM code itself is stored in an immutable, read-only section inaccessible to the application, meaning that an attacker cannot overwrite existing instructions or introduce new executable code at runtime. This design makes common exploitation techniques such as shellcode injection, just-in-time spraying, and self-modifying code attacks infeasible.

Structured Control Flow Instructions within a function are organized into well-defined nested blocks, with branching operations restricted to jumps to the end of surrounding blocks within the same function. This strict control prevents jumps to arbitrary addresses and disallows the use of unstructured goto statements [6, 9, 10]. As a result, executing data in memory as bytecode instructions is not possible, mitigating exploit techniques like return-oriented programming (ROP) [9].

Shadow Stack Protection WASM enhances control-flow security by employing a shadow stack, managed by the WASM virtual machine and containing return addresses and local variables. Unlike native execution environments where return addresses are stored directly on the program's stack, WASM's shadow stack is inaccessible to the executing module, preventing adversaries from modifying return addresses to hijack control flow. This mitigates common attack techniques such as return address corruption and ROP, which rely on tampering with saved return addresses to execute arbitrary code. By enforcing an immutable call-return discipline, the shadow stack significantly strengthens WASM's control-flow integrity (CFI) [6].

Table 1 summarizes the key security mechanisms of WASM, illustrating how each component contributes to its comprehensive security model by addressing specific classes of vulnerabilities.

2.2. Portability

WASM is designed as a portable binary instruction format, intended to execute consistently across diverse environments, operating systems, and hardware architectures [11]. Initially conceived to accelerate web applications [12], WASM has since expanded beyond browsers to run in cloud environments,

Table 1
Summary of WASM's security mechanisms and the threats they mitigate.

Mechanism	Description	Mitigated Threats
Sandboxing	Isolates WASM modules by confining their execution within a restricted environment, preventing access to system resources without explicit permissions [6].	Privilege escalation, unauthorized system access.
Module Validation	Statically verifies that a module conforms to the WASM specification before execution, ensuring that all operations are well-formed and type-safe.	Execution of malformed or malicious binaries, undefined behavior, type confusion attacks.
Type Safety	Enforces strict typing for function calls, memory accesses, and data, ensuring that operations adhere to declared types [9].	Type confusion vulnerabilities, improper memory accesses, type-based exploitation.
Memory Isolation	Enforces separation between the operand stack and linear memory, preventing direct corruption of execution state.	Stack smashing, arbitrary memory corruption, control data overwrite.
Control-flow Integrity	Enforces structured control flow by restricting branches to valid targets within well-defined blocks [6, 9].	Control-flow hijacking, arbitrary code execution, jump-oriented programming.
Shadow Stack	Maintains an internal, inaccessible stack for storing return addresses and call metadata, preventing tampering with the call/return sequence.	Return address corruption, ROP attacks, call stack manipulation.
Data Execution Prevention	Ensures linear memory is never executable and code sections are immutable, preventing the execution of injected code.	Code injection, shellcode execution, self-modifying code attacks.

edge computing platforms, and embedded systems with minimal changes to its core specification, demonstrating its remarkable portability.

At its foundation, WASM achieves portability through a carefully designed stack-based virtual machine architecture. This approach enables developers to compile code from languages like C, C++, and Rust into WASM modules that can execute wherever a compatible runtime exists, eliminating the need to tailor applications for each target environment. By providing a consistent runtime environment, WASM ensures that applications behave predictably regardless of the underlying platform.

Well-defined Specification The specification outlines a well-defined module structure organized into various sections, including type declarations, imports, exports, and code definitions. This standardized structure ensures that WASM modules can be reliably interpreted across different environments. WASM provides a compact, well-defined instruction set that includes operations for arithmetic, control flow, memory access, and function calls. The limited and precisely specified instruction set contributes to consistent execution behavior across platforms. The specification includes a rigorous type system with basic scalar types such as integers and floating-point numbers, along with a type-checking algorithm that ensures type safety. This type-checking helps prevent platform-specific behaviors that could compromise portability.

Canonical Execution Environment The WASM specification assumes that execution environments offer characteristics such as 8-bit wide bytes, single byte memory addressing granularity, support for unaligned memory accesses, two's complement integers, IEEE 754 floating point numbers, little endian byte ordering, lock-free atomic operations on 8, 16 and 32-bit values and secure isolation between WASM modules. Platforms that do not natively provide these capabilities can still execute WASM code by having the virtual machine emulate the required behavior, although this may impact performance [11].

Table 2
Summary of Wasm's portability mechanisms and the benefits they bring.

Mechanism	Description	Portability Benefits
Binary Format	Compact, architecture-neutral bytecode representation.	Ensures consistent interpretation across diverse hardware architectures [12].
Virtual Machine	Stack-based execution model with standardized semantics.	Creates predictable runtime behavior independent of host environment [11].
Host Integration	Controlled import/export interfaces.	Enables platform-specific functionality while maintaining core portability [13].
System Interface	WASI standardized APIs	Provides consistent access to system resources across diverse environments [14].

Host Environment Integration Wasm achieves portability while still allowing interaction with host environments through a controlled mechanism known as embedding [13]. Wasm modules can declare functions they require from the host environment. The Wasm runtime must provide implementations for these imported functions, allowing modules to access host capabilities. Modules can export functions, enabling the host environment to call into Wasm code. This bidirectional communication facilitates integration with diverse host platforms while maintaining a clean separation between the Wasm execution environment and platform-specific details. This design creates a clear boundary between portable Wasm code and platform-specific functionality, enhancing portability while enabling practical application development.

Limitations of WebAssembly The core Wasm specification intentionally omits built-in system calls and common interfaces for accessing underlying system resources such as file systems, network interfaces, or hardware peripherals [12]. While this design choice enhances security and portability, it introduces significant limitations for applications requiring system-level interactions, particularly in embedded contexts. To address this limitation, developers often leverage host-specific APIs or custom runtime extensions to implement system calls. While this approach provides flexibility, it introduces fragmentation as different environments may require distinct integration strategies.

WebAssembly System Interface To address these limitations without compromising portability, WASI [14] has emerged as a standardized solution for Wasm modules to interact with the host environment in a consistent way across different platforms. WASI provides a set of standardized APIs for essential system-level operations such as file I/O, networking, and time management. By abstracting platform-specific details, WASI enhances portability for applications running on general-purpose operating systems. WASI is designed around a capability-based security model where applications must be explicitly granted access to specific resources, enhancing security while providing necessary functionality.

Table 2 summarizes the key mechanisms that enable Wasm's portability across diverse computing environments, from browsers to embedded systems, highlighting how each feature contributes to consistent cross-platform execution.

3. Embedded Systems: Safety and Portability Challenges

The increasing connectivity of embedded systems has created significant security vulnerabilities that traditional enterprise and desktop computing security techniques fail to address adequately. Resource constraints represent a fundamental challenge, as embedded systems typically employ low-cost processors with limited CPU capacity, memory, and power [15]. These constraints often preclude the implementation of advanced security mechanisms such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Control-Flow Integrity (CFI) [16].

Market pressures compound these technical limitations. Manufacturers prioritize producing feature-rich chips at minimal cost, often neglecting long-term security and maintenance considerations. This economic model creates little incentive to update older devices once deployed, leaving them vulnerable to emerging threats. Even newly deployed embedded systems frequently run software several years behind current versions, and available security patches are seldom applied [17].

Programming errors constitute the primary source of vulnerabilities in embedded systems, typically exploited through buffer overflows or malicious input injection. Weak access control mechanisms frequently fail to properly authenticate users or enforce secure access policies, creating opportunities for privilege escalation or impersonation attacks. Additionally, implementation errors in protocol handling represent another common vulnerability source, more prevalent than design flaws in the protocols themselves [18].

Embedded systems present unique portability challenges due to their diverse operating environments. Many devices operate without general-purpose operating systems, instead relying on bare-metal programming or lightweight real-time operating systems (RTOS). This diversity requires developers to interact directly with varied hardware components across different devices and manufacturers, significantly complicating the development process.

HALs partially address these portability challenges by providing standardized interfaces for hardware resource access [19]. However, many HAL implementations remain specific to particular hardware families or vendors, creating inconsistencies when porting applications across different platforms. Even with a HAL, peripheral configuration differences and other low-level functionalities often necessitate device-specific code.

The fundamental portability challenge lies in balancing abstraction with performance. While HALs abstract certain functionalities, underlying configurations remain tied to specific hardware architectures. For example, common peripherals like UART or I2C exist across devices, but their initialization and configuration vary significantly due to differences in interrupt handling, clock gating, and direct memory access capabilities. This variance means that while application code may be portable, the HAL itself often requires substantial adaptation between platforms.

Vendor-provided HALs frequently lead to vendor lock-in, making platform migration costly and time-consuming. The embedded-hal Rust project [20] represents one approach to enhancing cross-platform portability by providing standardized interfaces (traits) for common peripherals such as GPIO, USART, SPI, and I2C. This approach allows developers to write hardware-independent code that can operate across diverse platforms. However, even this promising approach cannot entirely eliminate platform-specific code, particularly for peripheral initialization and configuration, which remain highly dependent on specific hardware architectures [21].

4. WebAssembly on Embedded Systems

Embedded systems present unique challenges for software deployment due to their resource constraints, diverse hardware architectures, and critical safety requirements. These systems traditionally rely on native code compiled specifically for their target platforms, creating significant barriers to software portability. Wasm, originally designed for browser environments, offers promising capabilities for these specialized computing contexts by providing a secure execution model.

The potential appeal of Wasm for embedded development lies in its ability to address two fundamental challenges in the field. First, its sandboxed execution model provides valuable security guarantees. Second, its binary format and virtual machine architecture offer a path toward greater code portability across the fragmented landscape of embedded hardware. While these benefits align well with embedded systems requirements, the introduction of Wasm into this domain requires navigating significant technical constraints and standards gaps that differ from those encountered in web or server environments.

4.1. Safe Execution of WebAssembly in Embedded Systems

In embedded environments, security is not just about protecting data: it is about ensuring the correct operation of critical system functions. Many embedded devices lack hardware-based security measures such as memory management units that provide memory isolation and protection, making software-based security mechanisms especially important.

WASM's sandboxed execution model addresses these concerns by providing strong isolation between untrusted code and core system components. This isolation is particularly valuable for embedded devices that may support dynamic updates or third-party modules. Each WASM module undergoes static validation before execution, and its program code can't change after being loaded. This design prevents runtime modifications or code injection attacks, ensuring that even if a module is compromised, the attack remains contained and cannot alter the system's fundamental behavior.

By requiring modules to declare all imported and exported functions, WASM enforces fine-grained control over which parts of the system can be accessed. This mechanism is particularly important in embedded devices, where direct access to peripherals or sensor data must be carefully restricted. The explicit boundaries help ensure that only validated, authorized code can interact with critical hardware, reducing the risk of unauthorized control or exploitation.

The security benefits of WASM in embedded systems do come with trade-offs. The virtual machine introduces additional overhead for resource allocation, context switching, and isolated memory management. In resource-constrained environments, this can lead to reduced execution efficiency and increased power consumption—exchanging some performance for enhanced security. These trade-offs must be carefully evaluated based on the specific requirements of each embedded application.

4.2. Challenges in Embedded WebAssembly Deployment

While WASM provides a portable execution environment for computational logic, its application in embedded systems faces significant challenges that go beyond those encountered in general-purpose computing environments.

A fundamental limitation is WASM's lack of standardized interfaces for hardware access. Embedded systems often require direct, low-level access to hardware resources such as GPIO pins, UART interfaces, sensors, and actuators. The current WASM specification does not provide native mechanisms for interacting with such hardware peripherals. This gap means that while the core computational aspects of applications can be portable, the hardware interaction layer remains platform-specific.

The adoption of WASI has begun to address similar challenges in general-purpose computing environments by standardizing APIs for file I/O, networking, time management, and random number generation. However, WASI was designed with the assumption that the host environment includes an operating system providing these standard services; such an assumption that doesn't hold for many embedded systems that operate without a full-fledged OS.

Efforts to extend WASI for embedded-specific needs are still emerging. Interfaces for common embedded protocols like I2C and SPI are under development but remain in early proposal stages [22, 23]. The absence of a standardized HAL specifically designed for WASM in embedded contexts presents a significant barrier to widespread adoption in resource-constrained environments.

Projects like *Embedded WASM* [24] have attempted to create embedded-hal [20] implementations for WASM, but these solutions are still experimental and, in some cases, development appears to have stalled. Until these challenges are addressed, WASM's application in embedded systems will likely remain limited to specific use cases where its security benefits outweigh the additional complexity of implementation.

5. Conclusions

This paper has examined WASM's potential and limitations in embedded systems contexts. WASM offers significant benefits for embedded applications, particularly through its inherent security features like sandboxing and isolation. These capabilities are especially valuable in embedded environments where hardware-based memory protection is typically unavailable due to the absence of virtual memory systems. Despite being originally designed for web browsers, Wasm's architecture has demonstrated remarkable adaptability, enabling its deployment across vastly different computing environments from desktop systems to cloud infrastructure and resource-constrained embedded devices.

However, our analysis reveals that Wasm falls short of achieving true "write once, run everywhere" portability in embedded contexts. While the specification successfully defines a virtual instruction set architecture and execution model, it deliberately avoids standardizing how modules interact with host resources. This design choice, which enables flexibility across diverse platforms, simultaneously creates significant challenges for embedded applications that require consistent hardware access. The import/export mechanism provides the technical foundation for host interaction, but without standardized interfaces, true portability remains elusive.

This limitation becomes particularly evident when comparing Wasm's usage patterns across different environments. In web applications, Wasm typically serves as an acceleration mechanism for compute-intensive tasks within a JavaScript-driven application, requiring only limited host system interaction. By contrast, standalone Wasm applications require comprehensive system interfaces comparable to POSIX APIs to support real-world functionality. The WASI proposal by the W3C WebAssembly Community Group attempts to address this gap by providing standardized APIs for application development.

Yet WASI's current specification primarily targets desktop and cloud environments, offering limited value for embedded systems that operate without an underlying operating system. In such environments, conventional operations like filesystem access or socket communication may be entirely irrelevant. The emerging efforts to extend WASI with embedded-specific interfaces for protocols like GPIO, SPI, and I2C represent promising developments, but these standards remain in early stages and insufficient for comprehensive embedded deployment.

The diversity of embedded hardware presents additional challenges beyond interface standardization. Embedded devices vary significantly across manufacturers and even within product families, necessitating customized initialization code for microcontrollers and peripherals. To mitigate this challenge, we propose architectural approaches that strongly decouple business logic from hardware-specific implementation, allowing for targeted customization while preserving application portability. Furthermore, the development of lightweight, modular runtime environments specifically optimized for embedded workloads could significantly improve both performance and portability in resource-constrained contexts.

By addressing these technical and standardization challenges, Wasm could emerge as a transformative technology for embedded systems development, enabling more secure, portable, and efficient applications. Such advancement would be particularly valuable for meeting the evolving requirements of IoT and edge computing ecosystems, where secure code execution and cross-device deployment represent persistent challenges. The continued evolution of Wasm standards, particularly those focused on embedded-specific requirements, will largely determine whether this promising technology can fulfill its potential in the embedded systems domain.

Declaration on Generative AI: The author(s) have not employed any Generative AI tools.

Funding: This work was partially supported by the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU through Project "SERICS – Security and Rights in CyberSpace" under Grant PE00000014.

References

[1] J. Shehu Yalli, M. Hilmi Hasan, A. Abubakar Badawi, Internet of Things (IoT): Origins, embedded technologies, smart applications, and its growth in the last decade, IEEE Access 12 (2024) 91357–91382. doi:10.1109/ACCESS.2024.3418995.

[2] WebAssembly Community Group, WebAssembly text format, 2024. URL: https://webassembly.github.io/spec/core/text/index.html.

[3] WebAssembly Community Group, WebAssembly binary format, 2024. URL: https://webassembly.github.io/spec/core/binary/index.html.

[4] T. Brito, P. Lopes, N. Santos, J. F. Santos, Wasmati: An efficient static vulnerability scanner for webassembly, Computers & Security 118 (2022). doi:10.1016/j.cose.2022.102745.

[5] B. C. Pierce, Advanced Topics in Types and Programming Languages, The MIT Press, 2004.

[6] WebAssembly Community Group, WebAssembly documentation - security, 2017. URL: https://webassembly.org/docs/security/.

[7] M. W. Docs, WebAssembly concepts, 2024. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

[8] M. Vassena, WebAssembly foundations, 2024. URL: https://ics-websites.science.uu.nl/docs/vakken/mlbs/lectures/lec5/notes/1-core-wasm.md.html.

[9] D. Lehmann, J. Kinder, M. Pradel, Everything old is new again: Binary security of WebAssembly, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 217–234. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[10] M. Vassena, WebAssembly control-flow, 2024. URL: https://ics-websites.science.uu.nl/docs/vakken/mlbs/notes/2-control-flow.md.html.

[11] WebAssembly Community Group, WebAssembly documentation - portability, 2015. URL: https://webassembly.org/docs/portability/.

[12] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien, Bringing the web up to speed with webassembly, SIGPLAN Not. 52 (2017) 185–200. URL: https://doi.org/10.1145/3140587.3062363. doi:10.1145/3140587.3062363.

[13] WebAssembly Community Group, WebAssembly specification - appendix - embedding, 2024. URL: https://webassembly.org/docs/portability/.

[14] WebAssembly Community Group, WebAssembly System Interface, 2024. URL: https://github.com/WebAssembly/WASI.

[15] P. Koopman, Embedded system security, Computer 37 (2004) 95–97. doi:10.1109/MC.2004.52.

[16] X. Zhou, P. Wang, L. Zhou, P. Xun, K. Lu, A survey of the security analysis of embedded devices, Sensors 23 (2023). URL: https://www.mdpi.com/1424-8220/23/22/9221. doi:10.3390/s23229221.

[17] B. Schneier, Security risks of embedded systems, 2014. URL: https://www.schneier.com/blog/archives/2014/01/security_risks_9.html.

[18] D. Papp, Z. Ma, L. Buttyan, Embedded systems security: Threats, vulnerabilities, and attack taxonomy, in: 2015 13th Annual Conference on Privacy, Security and Trust (PST), 2015, pp. 145–152. doi:10.1109/PST.2015.7232966.

[19] K. Popovici, A. Jerraya, Hardware Abstraction Layer, Springer Netherlands, Dordrecht, 2009, pp. 67–94. URL: https://doi.org/10.1007/978-1-4020-9436-1_4. doi:10.1007/978-1-4020-9436-1_4.

[20] rust embedded, A Hardware Abstraction Layer (HAL) for embedded systems, 2024. URL: https://github.com/rust-embedded/embedded-hal.

[21] M. Geisler, Comprehensive Rust – embedded-hal, 2024. URL: https://google.github.io/comprehensive-rust/bare-metal/microcontrollers/embedded-hal.html.

[22] WebAssembly Community Group, I2C API for WASI, 2024. URL: https://github.com/WebAssembly/wasi-i2c.

[23] WebAssembly Community Group, SPI API for WASI, 2024. URL: https://github.com/WebAssembly/wasi-spi.

[24] Embedded WASM Working Group, WebAssembly for embedded devices, 2023. URL: https://github.com/embedded-wasm.