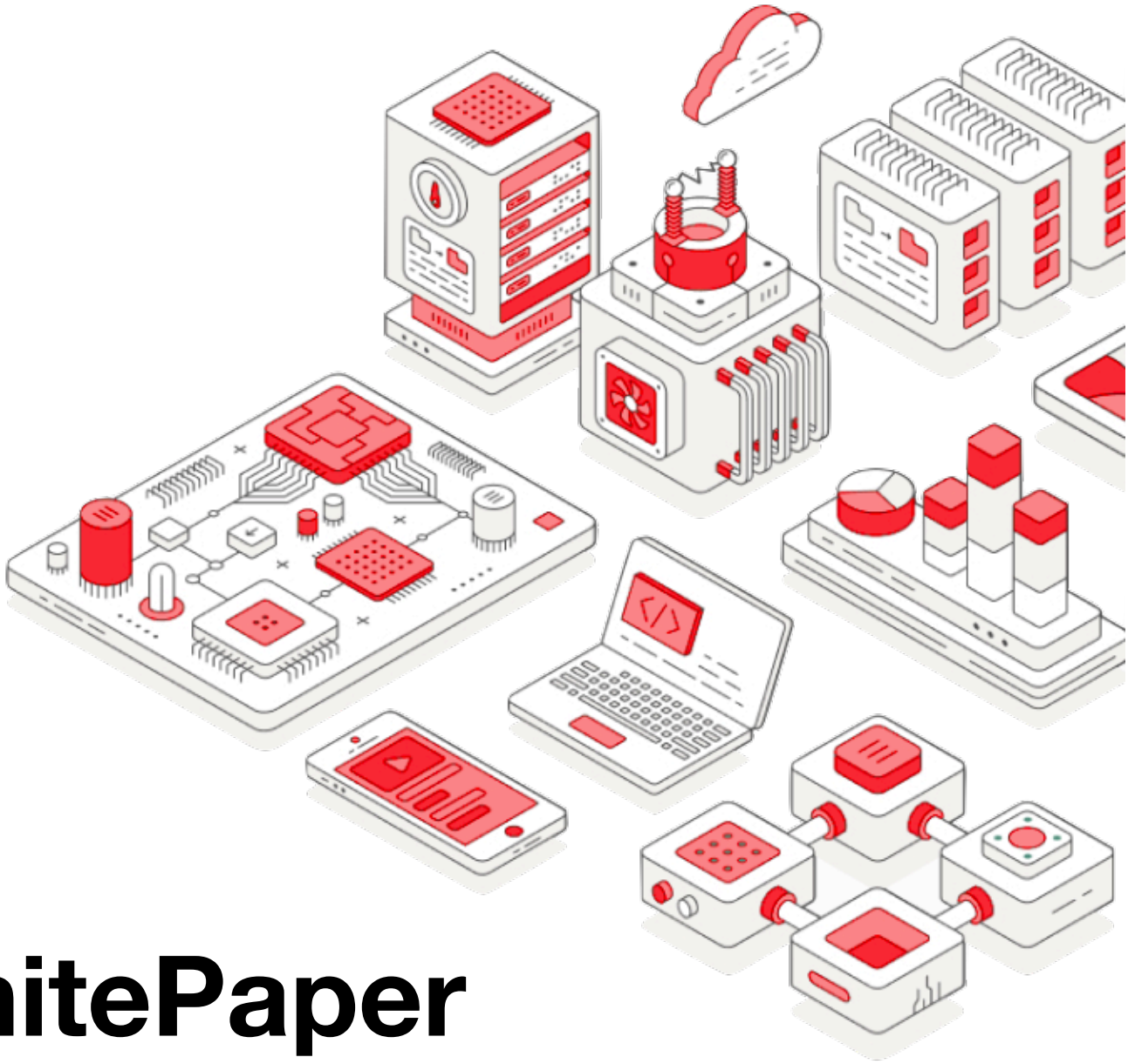

WhitePaper

The Next IoT Cycle
WebAssembly at the Bottom of the Stack

Standardizing Portable Secure Execution for Real-World IoT Using WebAssembly

October 2025
Author: Aaron Ardiri, CEO - RIoT Secure AB

Abstract

WebAssembly (WASM) and the WebAssembly System Interface (WASI) are emerging as the first credible candidates for a universal execution layer that spans browser, cloud, edge, and embedded environments. In IoT, where devices operate under extreme resource constraints and long deployment lifetimes, this shift is strategically significant.

This white paper documents RIoT Secure's approach to bringing WebAssembly to micro-controller class devices, including the development of a **fully-compliant lightweight runtime**, a proposed **universal device I/O ABI** for hardware access, and the planned integration of WebAssembly execution as a first-class deployable unit inside an already **commercial lifecycle management platform** used in production for over five years. We present measured results demonstrating feasibility on constrained hardware, outline the implications for portability, security, lifecycle economics, and industry standardization, and argue that the decisive window to align on an open execution standard for IoT is now.

Executive Summary

IoT is entering a phase where long-lived devices must run securely isolated and verifiable logic securely at the edge - yet the industry still relies on firmware models designed for **static, single-vendor deployments**. At the same time, WebAssembly (WASM) and the WebAssembly System Interface (WASI) have emerged as credible candidates for a universal execution layer. What WebAssembly did for the browser - portability, sandboxing, and determinism - needs to happen at the device level for IoT.

RIoT Secure has been preparing for this transition **long before the ecosystem caught** up. For over five years our lifecycle management platform has been commercially deployed in production, delivering and maintaining secure MCU-based firmware at scale for a high-profile customer. The work described in this white paper extends into the WebAssembly domain: we have built and validated a lightweight, standards-compliant WebAssembly runtime for micro-controllers, investigated the practical constraints of toolchains and binary size, and published a proposal for a universal device I/O ABI - the missing layer that enables truly portable WASM workloads on real hardware.

This is not a **conceptual or academic exercise** - RIoT Secure is executing at the critical inflection point: before the ecosystem hardens around incompatible vendor-specific runtimes. By aligning early with WebAssembly and publishing a concrete device I/O proposal, we aim to help shape the standard before fragmentation becomes irreversible, while positioning RIoT Secure to lead and commercialize the resulting gap in the market.

WebAssembly is moving from browsers into edge and IoT, and because standardization is not yet fixed, this is a **rare shaping window**. RIoT Secure enters that window with a fully-compliant, micro-controller sized WASM runtime **already built and proven on constrained hardware** - not theorized - and with an existing lifecycle platform already in **commercial use for over five years**, making WebAssembly an extension of a deployed product rather than a new experiment. The single largest gap in the market is the absence of a universal hardware I/O ABI for WebAssembly, which is forcing every vendor to build their own and guaranteeing fragmentation.

Problem Space - Why This Matters Now

The IoT industry has reached a structural constraint: devices are expected to live 10–20 years in the field, yet most still ship firmware the way we did in 2005 - **vendor-locked, monolithic, and non-portable**. Every hardware vendor exposes its own SDK, its own abstractions, its own update tooling, and its own security posture. When hardware changes, the software must be rewritten. When vulnerabilities appear, updates are slow, vendor-dependent, and costly. When third-party logic needs to run at the edge, it is welded into firmware rather than executed in an isolated environment.

This fragmentation is not just inconvenient - it is a strategic barrier to scale:

- **No portability means no viable ecosystem**
  every driver and workload is rewritten per vendor.

- **No isolation means no real-world safety**
  a single bug becomes a full compromise.

- **No standard interface means no supply-chain leverage**
  hardware cannot be swapped.

- **No abstraction means no delegation**
  third parties cannot ship code safely and focus on native targets.

Meanwhile, **expectations on devices are rising - not shrinking**. Edge devices are expected to run AI/ML workloads, support secure OTA update, pass compliance, interoperate across vendors, and remain maintainable without physical access. The “flash once and forget” model is no longer viable.

The industry now requires an execution model that is:

- **Portable** across vendors and silicon generations
- **Sandboxed** and capability-constrained
- Small enough for **micro-controllers**, not just Linux edge
- **Updatable over the air** without firmware rebuilds
- Auditable and reviewable **independent of vendor SDKs**

WebAssembly and WASI promise this - but only if implemented in a form that works under real IoT constraints, not just in cloud and browser environments.

- Runtimes do not meet memory and determinism **constraints** of real micro-controllers.
- No standard I/O ABI exists to let WASM touch actual hardware without vendor lock-in.

RIoT Secure’s work begins where the current state seems to have stagnated: the industry has agreed on the right execution model in principle, but no one has carried it across the finish line to embedded reality, hardware I/O, and fleet-scale lifecycle integration.

WebAssembly & WASI - Promise vs Current Reality

WebAssembly (WASM) was not created for IoT - it was created to bring **safe, high-performance execution to the browser**. What made it work there is exactly what IoT needs now: a **compact, deterministic, sandboxed execution format** with no implicit system access and strong isolation guarantees. As tooling matured, WebAssembly left the browser and became viable on servers, edge gateways, and even secure environments. That same transition is now approaching the embedded world.

The industry expectation around WebAssembly in IoT is clear:

- **Write in any language** → compile to WASM binaries
- **Deploy to any device** → without vendor-specific firmware
- **Run in a sandbox** → with only explicit permissions
- **Update the workload independently** → without reflashing base firmware

In parallel the WebAssembly System Interface (WASI) was introduced to extend WASM beyond the browser by defining system-level APIs such as files, clocks, sockets, and randomness. In cloud and server environments, it's hinting as an alternative to containers.

However the current reality does not yet match the promise for IoT:

- **WASI is still unfinished and fragmented**
  Three incompatible previews exist in parallel (P1, P2, P3), each with different design goals and adoption footprints.

- **Toolchains target servers, not micro-controllers**
  The ecosystem assumes megabytes of RAM and large runtimes; virtually nothing is designed for 64KB-class devices.

- **No standard exists for hardware I/O**
  WASI defines files and sockets - but not GPIO, I2C, SPI, UART, or device-level interfaces. Every vendor invents their own.

- **If nothing is published now, fragmentation will harden by default**
  The industry is at the pre-POSIX moment - alignment must happen before platforms diverge permanently.

WebAssembly is on the right trajectory, but it is not yet deployable as a universal execution layer for IoT **without additional engineering**. The gap is not theoretical - it is precisely the gap RIoT Secure chose to work on:

- A WebAssembly runtime **engineered specifically** for MCU constraints
- A portable I/O ABI proposal that avoids **vendor lock-in** at the hardware boundary
- Integration of WASM delivery into an **already-commercial lifecycle platform**

This is the gap RIoT Secure focused on closing and this white paper captures that work and its strategic timing: moving before the standard is decided, not after.

From BRAWL to WASM - Strategic Shift

When RIoT Secure began designing portable execution for IoT back in 2017, WebAssembly was still a browser experiment - there was no WASI, no embedded runtimes, and no ecosystem capable of supporting edge deployment. Under those conditions, building our own byte-code (BRAWL) was the **only viable path** to achieve portability, determinism and sandboxing on constrained devices. For several years, BRAWL served **exactly that purpose**: it proved that firmware could be decomposed into a portable, sandboxed, updatable execution format independent of vendor SDKs.

But the environment has changed. WebAssembly has matured from a browser artifact into a **general-purpose execution standard**. WASI evolved enough to validate the direction, even if not yet unified. And standardization momentum now matters more than owning a **proprietary virtual machine**.

- **WASM matured from a browser artifact to a universal runtime target.**
  Compiler support, tooling, and community adoption reached a tipping point - not just from web developers but from systems vendors and cloud platforms.

- **WASI evolved enough to validate the direction, even if not yet unified.**
  The trajectory showed that the industry was aligning on the same model BRAWL was built around - portability through sandboxed interfaces instead of vendor-specific firmware.

- **Standardization momentum now matters more than absolute control.**
  The long-term value is in interoperability and ecosystem reuse, not in owning a bespoke VM forever.

Instead of defending a **proprietary instruction set**, RIoT Secure made the deliberate decision to transition BRAWL to a **web-standard execution foundation** - inheriting the WebAssembly core while **preserving the original mission**: micro-controller-first portability with deterministic execution and strict sandboxing.

The transition did not reset our work - it accelerated it:

- Everything BRAWL enabled **(portability, isolation, MCU suitability)** is preserved.
- Now built on a **standardized execution core** instead of a proprietary one.
- Developers gain access to **existing toolchains and tooling** - instantly expanding who can develop for our platform.

We are not **abandoning control or discarding** what we built - we are aligning it with the standard the industry is already moving toward. By shifting from a proprietary VM to WebAssembly, we **preserve our architecture,** our learnings, and our advantages, while ensuring they operate within the same execution model the ecosystem will eventually converge on. This allows our work to **scale with the market** rather than stand outside it, increasing both its adoption potential and its strategic value.

Engineering WebAssembly for micro-controllers

Running WebAssembly on a browser or Linux server is **straightforward** - the environment provides ample memory, system calls, filesystems and schedulers and mature host runtimes. None of that exists on a bare-metal micro-controller. A viable MCU-class WebAssembly runtime **must meet constraints that typical WASM engines never consider**: kilobytes of RAM, static memory layouts, no operating system, no dynamic loader, no allocator assumptions, and deterministic timing.

When we set out to implement our runtime, we made three engineering commitments:

- **Full compliance with the WebAssembly MVP (1.0), not a subset**
  Many runtimes "optimize" for embedded devices by **removing instruction classes**, **skipping validation,** or i**ntroducing custom opcodes**. That creates fragmentation immediately. We went the opposite direction: implement the **full WebAssembly MVP specification** exactly, then optimize under those rules.

- **Absolute control of memory - nothing implicit, nothing hidden**
  The runtime must not assume the availability of:

  - unlimited memory (heap and stack)
  - a file system
  - threads or timers
  - OS services of any kind

  All memory usage inside the runtime is **explicit and predictable**. The runtime and the module cannot allocate beyond declared limits - a hard requirement for safety-critical and resource-isolated environments.

- **Determinism over convenience**
  IoT devices often operate where non-determinism is a security liability, not a feature. We rejected JIT compilation, background threads, host callbacks, or async suspension. Every instruction path is **inspectable, bounded, and reproducible** - essential for both certification and remote certification.

Implementation Result (BRAWL WebAssembly Runtime)

- ~12,000 lines of C - fully readable, test-driven, and hardware-portable
- ~46 KB flash overhead - small enough for low-end micro-controllers
- ~14 KB RAM - configurable, bounded, and stable
- 100% compliance with MVP against ~12,000 official WebAssembly test vectors

This is not a conceptual or partial port - it is a **fully conformant execution engine** built specifically for the class of devices the industry actually ships, by an embedded team with more than 30 years of experience with resource-constrained environments, ensuring the implementation reflects real-world constraints rather than academic assumptions.

Demonstrated Feasibility - Proven on Real Hardware with Measurable Results

We validated this on both an Arduino UNO R4 WiFi with a Cortex-M4F MCU (32 KB RAM / 256 KB flash) and an ESP32-S3 WROOM board (520 KB RAM / 16 MB flash), running standard WebAssembly modules and interacting with hardware only via imports - proving WebAssembly is viable on micro-controllers today, not just “embedded Linux”.

For investors, acquirers, and strategic partners, the question is not whether WebAssembly could work on micro-controllers - but whether it already has been proven under real constraints. We have crossed that threshold.

The Binary Size Reality - High-Level Languages compiled to WebAssembly

The promise of WebAssembly is often summarized as: **“write in any language, compile to WebAssembly, run anywhere.”** That statement is directionally true - but in IoT, it is not universally true. On micro-controllers, the constraint is not the instruction set - it is the **binary size and runtime baggage** that each toolchain brings with it.

To validate feasibility, we **prepared and compiled two trivial programs** in high-level programming languages, that every developers knows - (“Hello World” and Fibonacci) across multiple toolchains into WebAssembly (WASM). The results are unambiguous:

Language / Toolchain	Typical WASM Size
C (Emscripten)	1-11 KB
C (WASI SDK)	20–100 KB
TinyGo	21–110 KB
Rust (wasip1/wasip2)	50–93 KB
JavaScript (Javy)	1 MB
Go (standard toolchain)	2-3 MB
Python (py2wasm)	25 MB

The conclusion is not that some languages simply “don’t work” and not feasible - it is that their runtime assumptions make them unsuitable for micro-controller-class devices as-is.

Why the sizes explode

Across high-level languages, the bulk does not come from the application and business logic - it comes from:

- **Bundled runtimes** (garbage collectors, schedulers, allocators, interpreters)
- **Standard library payloads** pulled in even when unused
- **Abstraction layers** for WASI and host integration
- **Compiler safety nets** such as panic handlers and debug metadata

In cloud and server environments, this overhead is trivial. In IoT, it determines feasibility.

What actually works for embedded WebAssembly

For devices in the kilobyte-to-megabyte class, only three approaches consistently produce viable binaries:

- **Handwritten WAT**
  smallest possible artifact, but not practical for most projects

- **C / Rust / TinyGo**
  with aggressive linker flags - realistic and maintainable

- **Custom-host-provided functionality**
  via imports instead of bundling libraries - offload complexity to host

This directly influences the future WebAssembly ecosystem for IoT: the languages that survive are those that can **minimize or externalize their runtime burden**.

Strategic implication

This is where RIoT Secure's runtime and device I/O ABI work matters: by defining stable, minimal host imports for device I/O, higher-level languages can remain lean - without embedding full hardware access logic inside the module itself. Instead of every toolchain carrying drivers and hardware logic, the runtime supplies it once, portably.

Binary size is not an academic detail - it is the gatekeeper for adoption on real devices.

The Missing Layer - Why a Universal Device I/O ABI Is Required

Even if WASM runtimes become small enough for micro-controllers, portability still fails without a **standard way** for modules to **access real hardware**. Today, there is no consensus - every WASM-on-device solution implements its own private API surface for GPIO, SPI, I2C, UART, displays, sensors, etc. The result is deterministic fragmentation:

- A sensor driver written for Runtime A **does not run** on Runtime B
- The same WASM module **cannot run across two boards** without recompilation
- OTA deployments still **require per-hardware variants**
- **Ecosystem reuse becomes impossible** before it even gets established

Without a standard device ABI, WebAssembly in IoT would reproduce the exact failure pattern that UNIX faced pre-POSIX: **many runtimes, zero portability.** The same dynamic later re-appeared with Java and .NET, where "write once" portability collapsed into **parallel, incompatible ecosystems** that forced vendors to **maintain separate code paths for each runtime**.

Our Perspective When Designing a Proposed ABI

We did not design an **idealized or "future-WASI" system**. We constrained the problem deliberately: the question we solved was **minimalist by intent**:

> *"What is the smallest stable ABI that allows WASM to talk to real device I/O - without requiring threads, async, post-MVP features or OS services?"*

The ABI therefore obeys strict constraints:

- **No traps, no exceptions, no host callbacks**
  all operations return numeric codes
- **All data passed through linear memory**
  the only universal primitive WebAssembly guarantees
- **Symmetric semantics**
  same ABI works if the bus is hardware or emulated in software
- **No dependency on Preview 2/3, WIT or language tooling**
  ABI works under WebAssembly MVP (1.0), regardless of target environment
- **Language-agnostic and vendor-neutral**
  one C header, one WAT import block, one Rust FFI

Five Device Domains Covered First

Rather than attempt a full catalog, we standardized the primitives that underpin almost all embedded deployments at a hardware level:

- **GPIO** - the foundation of all device control
- **I2C** - the dominant bus for sensors and configuration peripherals
- **1-Wire** - widely used in instrumentation and identity chips
- **SPI** - the backbone for high-throughput peripherals and flash
- **UART** - the universal debug/bridge/protocol escape hatch

If these are not portable, nothing layered above them (sensors, actuators) can be.

Why This Matters Now - Not Later

Standardization does not happen once products ship - it happens before they **harden around incompatible defaults**. If the industry simply waits for WASI to eventually include device I/O after vendors embed proprietary ABIs in shipped devices, the window for portability closes for a decade - a process that is already eight years deep.

This white paper therefore publishes the ABI not as **"the one true standard"** - but as a concrete starting position before fragmentation becomes irreversible.

Demonstrated Feasibility - Zero-Bloat Demos & Cross-Hardware Execution

A portability standard is only credible if it is **implemented and proven on real hardware** - not merely defined on paper. To validate the practicality of a WebAssembly based execution layer on micro-controllers, we built and executed a series of **zero-bloat demonstrations** using the same logic expressed in multiple forms: handwritten WebAssembly Text (WAT), C, Rust, and TinyGo compiled to WASM.

These demos were not simulations; they ran on **physical boards and interacted with actual device hardware** using only host-provided imports - not vendor SDK code inside the module.

The Proof Case: LED Matrix “Bouncing Balls” Demo

We selected a visual demo intentionally - to eliminate ambiguity and speculation. The same WebAssembly module was executed on:

- **Arduino UNO R4 WiFi**
- **ESP32-S3 WROOM** with larger LED matrix attached

Target	MCU	RAM	Flash	Result
Arduino UNO R4 WiFi	Cortex-M4F @48 MHz	32 KB	256 KB	✅ worked
ESP32-S3 WROOM	Xtensa dual-core @240 Mhz	520 KB	16 MB	✅ worked

In both cases:

- The **module was unchanged** - no vendor-specific rebuilds
- The **runtime resolved imports** to the correct hardware interface
- The **behavior and timing were identical** across boards

This confirms the core claim: portability is not hypothetical - it is achievable today. No rebuild, no modification, no board-specific divergence. The difference in hardware was absorbed entirely by the runtime and device I/O ABI - not by the application module.

Memory Footprint - Fits Real IoT Budgets

Component	Flash	RAM
Runtime (full MVP)	~46 KB	~14 KB
Application module	1–50 KB	inside same linear page
Total	**< 70 KB**	**fits within 256 KB MCU**

This fits inside the class of devices deployed in real IoT fleets today.

Zero-Bloat Builds from Modern Languages

Using the same logic and the same ABI imports:

- **C and Rust builds** reproduced behavior with minimal overhead
- **TinyGo builds** remained in viable size ranges for micro-controllers
- **No embedded runtime logic** for displays or timing was bundled - all via host ABI
- **WAT baseline** served as the canonical reference for correctness and footprint

The goal was not to write a demo - the goal was to prove that:

- Modern languages can target WASM without pulling MBs of runtime baggage
- Portable logic can ship as WASM while hardware access stays in the host
- MCU-class deployment is possible without modifying the module per device

Why this matters beyond the demo

Even though the demo is simple, what it proves is foundational:

- If an animation can run identically on two boards, so can a sensor driver
- If imports abstract LEDs, they can abstract radios, storage, and modems
- If WASM modules run today on an Arduino, they can run anywhere lighter or bigger

We have not just described portability - we have demonstrated it under constraint.

Strategic Implication

Because the technology has already been proven on physical hardware, the remaining work is execution - **commercial, not technical**. The next phase is about **scaling**: integrating WebAssembly modules as a **first-class payload into the existing lifecycle platform** and operationalizing delivery at fleet scale. The risk is already removed on the technical side - the hard parts are solved - which means the remaining effort is **productization and go-to-market, not invention**.

- **Scaling**
  Integrating the WASM module payloads into the lifecycle platform

- **De-risked technology:**
  Hard parts are solved - remaining work is productization

- **First-mover timing:**
  Standard not yet locked - window still open to make a massive influence

- **Acquisition leverage:**
  Whoever owns runtime and delivery owns the device execution layer

The **timing advantage** is significant: the standard is not yet locked, which means there is still a window to shape the space and become the reference implementation instead of adapting to someone else's. This is not early-stage research - it is a validated execution model positioned to be **commercialized ahead of industry consolidation**.

Bottom Line

This is **not a hypothesis or a conceptual design** - it is a **real implementation** that has already been built, executed on physical micro-controllers, **validated against compliance suites**, and measured under realistic constraints. The work is **technically de-risked**: the runtime exists, the feasibility has been demonstrated on production-class hardware, and the **supporting ABI is defined**. What remains is not **proving that it works**, but **bringing it to market** at the right moment - before the **industry settles on a de-facto standard** and before vendor-specific solutions harden into place. In other words, this is a **timing advantage**, not a technical gamble, and it is **positioned to be commercialized ahead of consolidation**.

Integration Path: WebAssembly a First-Class Deployment Format in an Already-Commercial Platform

RIoT Secure is entering the WebAssembly era from a **position of proven execution**, not early experimentation. Unlike cloud-first platforms that assume Linux resources, our system was **built for devices** with **limited RAM, intermittent connectivity**, and long **service lifetimes.**

Today: A Sandboxed Architecture via Separation of Concerns

Our current design already enforces isolation at the hardware level: a **dedicated micro-controller** is allocated strictly for security, communications, and policy enforcement, independent from the application MCU. This separation acts as a **hardware-enforced sandbox**, preventing uncontrolled execution on the application side and ensuring secure lifecycle management. This provided a hardware sandbox even before WebAssembly.

This means RIoT Secure already provides:

- **Isolated execution domain** for controlled workloads
- **Secure OTA firmware** delivery & rollback
- Cryptographic **integrity & identity**
- Fleet-wide **state & policy management**

WebAssembly does not introduce sandboxing from scratch - it introduces a standardized, portable, language-agnostic sandbox format on top of an already-enforced architecture.

Tomorrow: WebAssembly as a Parallel Execution Channel

We are not replacing native firmware support - we are extending the existing pipeline so WASM modules can be delivered in parallel using the same lifecycle controls:

Capability	Native Firmware	WASM Modules
Delivery channel	✅ Already commercial	✅ Added in parallel
Isolation model	✅ Hardware-based MCU split	✅ Software sandbox in-app MCU
Portability	❌ Recompiled per hardware	✅ Single binary, multiple boards
3rd party logic	Risky to embed into firmware	Safe, revocable, runtime-bound
OTA granularity	Full firmware image	Fine-grained module swaps

This is an additive evolution, not a migration gamble.

Why This Matters Commercially and Strategically

Because the lifecycle infrastructure already exists:

- **Technical risk** is low - infrastructure is proven
- **Time to adoption is short** - adding format, not rebuilding platform
- **Acquisition relevance is high** - WebAssembly drops into a mature system

We are not selling a runtime - we are **commercializing an execution layer** inside an already deployed lifecycle product. The transition from **native, vendor-tied firmware to portable, sandboxed byte-code** is not just a technical change - it alters **who controls the IoT stack** and where value accumulates.

Strategic Implications for the Industry: Why This Changes Control, Economics, and Ecosystem Power

The shift from native, vendor-specific firmware to **portable, sandboxed byte-code** is not just a technical improvement - it changes the economic and strategic structure of IoT. Whoever **controls the execution layer controls the ecosystem above it**: the developer tooling, the security model, the update channel, and ultimately the business models built on top.

WebAssembly introduces a new equilibrium point in IoT similar to what POSIX did for UNIX and what the JVM did for enterprise software - but this time at the device layer.

Vendor Lock-In Collapses

Today, hardware **vendors control ecosystems by controlling firmware APIs and toolchains**. If WebAssembly becomes the unified execution format:

- Code becomes **portable across vendors**
- Hardware becomes **swappable without rewrite**
- Integrators can choose based on cost, supply, or regulation - not SDK compatibility

This shifts leverage from chip vendors to execution-platform owners.

Over-The-Air Updates Stop Being Firmware Events

Today, updating functionality means reflashing firmware - a high-risk, high-cost, slow process. With WebAssembly:

- Updates can target **individual modules**, not full firmware images
- **Risk, downtime, and certification impact** are all reduced
- Regulatory **re-certification burden drops** from firmware to isolated logic units

This changes the economics of maintaining fleets over 10–20 year lifetimes.

Security Review and Compliance Move Upstream

With native firmware, security review is opaque: every vendor implements its own stack. WebAssembly **shifts this dynamic**:

- A single portable artifact can be **audited once and deployed fleet-wide**
- **Capability-based sandboxing** constrains attack surface by default
- Compliance frameworks can validate **the module**, not the full firmware image

This opens the door to standardized security certification at the workload level.

Third-Party Logic Becomes Economically Viable on Devices

Today, deploying third-party logic to devices is **dangerous and expensive** - requiring trust in native firmware injection. WebAssembly changes this:

- Third parties can ship code as **sandboxed artifacts**
- Code can be versioned, revoked, certified, and isolated
- A **marketplace of device workloads** becomes possible

This is how ecosystems - not just products - are created.

Timing Matters - Standards Are Not Set Yet

The IoT ecosystem is at a pre-standardization inflection point:

- WebAssembly is **mature enough to deploy** - but not yet fragmented by incumbents
- WASI is advancing - but has **not yet defined hardware I/O**
- No **dominant runtime vendor has captured** the embedded layer

This is the rare phase in a platform cycle where **technical alignment can create structural control**. This is precisely the window in which **technical leadership becomes structural advantage.** Whoever establishes the execution device I/O layer before consolidation becomes the reference point others must follow.

Strategic Summary

WebAssembly on micro-controllers is not merely an efficiency gain - it changes who controls IoT software, how long devices remain economically serviceable, **who captures ecosystem value, and which companies become acquisition targets** as consolidation begins.

RIoT Secure is positioned on the correct side of that change - early enough to shape it, late enough to execute it against reality, not theory.

RIoT Secure: Proven Lifecycle Platform Built for the Class of Devices Everyone Else Ignores

RIoT Secure is **not a research group or a cloud-first startup retrofitting ideas** downward into hardware. For more than five years, our lifecycle management platform has been running in **production** - on real micro-controllers - delivering secure firmware updates and enforcing device policy at scale.

Where most IoT platforms assume Linux-class resources, we built for the class of devices that define the **real IoT market**: 32–256 KB MCUs, long-lived deployments, intermittent connectivity, and zero-trust operating environments.

A Platform Already Proven in the Field

The existing RIoT Secure lifecycle stack provides:

- **Secure OTA delivery** to low-resource devices
- Cryptographic **integrity & certification** of updates
- Fleet-wide **version management** & rollback
- Device **on-boarding, identity, and policy enforcement**
- **Operational longevity** - built for 10+ year lifecycles

This is not theory - it is already commercially deployed and revenue-validated.

Engineered for Resource-Constrained Environments - Not Cloud Convenience

Most platforms assume threads, filesystems, MBs of memory, and a container host. Our platform was explicitly built for environments where none of those exist:

- Works on **bare-metal micro-controllers**
- No **OS dependency**
- **Deterministic memory & runtime behavior**
- Communication and security isolated on a **dedicated control MCU**
- Application MCU treated as an **untrusted sandbox** domain from day one

That architectural decision - made before WebAssembly entered the conversation - gives us a structural advantage now.

Now Extending to Become the WASM Delivery Standard

The WASM runtime and device I/O ABI work described in this paper is not an **independent experiment** - it is the next logical stage of the same platform:

- WebAssembly becomes a **second artifact type** delivered via the same OTA pipeline
- Hardware sandboxing remains (if needed), WebAssembly adds a **software sandbox**
- Modules can be updated, revoked, and audited without reflashing firmware
- The same lifecycle controls apply to **both native firmware and WASM payloads**

This positions RIoT Secure uniquely in the market:

> *Our work is not about producing another WASM runtime - it is about making WASM deployable at scale in IoT by embedding it into an existing lifecycle platform that has already been proven in production on constrained devices.*

Commercial and Strategic Implication

Because the lifecycle infrastructure, security model, and field deployment already exist:

- **The technology is de-risked** - only the integration stage remains
- **The market timing is optimal** - before WebAssembly standardization hardens
- **The acquisition value is high** - a ready-made platform into which WASM drops

The runtime makes us early. Our platform makes us credible. The convergence makes us strategically valuable.

Conclusion

IoT is entering a decade in which devices will not only need to run for long lifetimes - they will need to run new logic, from new parties, under new security expectations, without replacing firmware and **without trusting vendors blindly.** The existing model of vendor-

locked firmware, opaque code bases, and per-hardware rebuilds is **fundamentally incompatible with that future.**

WebAssembly provides the missing execution model - **portable, sandboxed, and language-agnostic** - but by itself it is not ready for IoT. Two gaps had to be closed before WebAssembly could become the foundation of device computing: a runtime that fits micro-controllers, and a portable device I/O layer that breaks hardware lock-in. **We have demonstrated both.**

At the same time, RIoT Secure is **not entering this space from a standing start.** Our lifecycle management platform is already deployed, already trusted, and already operating under the exact constraints and realities where WebAssembly will matter most. Integrating WebAssembly as **a first-class deployment format** is not a speculative bet - it is a leverage play on top of proven infrastructure.

IoT devices are expected to operate for a decade or more, under untrusted conditions, with evolving requirements - yet the industry still relies on static, vendor-specific firmware models that cannot scale. WebAssembly introduces a portable, auditable, sandboxed alternative that aligns with the future of IoT - but only if implemented in a form suitable for micro-controllers and standardized before fragmentation hardens.

The inflection point is now - before proprietary ABIs become entrenched, before toolchains diverge, before the standard is set by incumbents.

This white paper establishes three conclusions:

- **WebAssembly on micro-controllers is not theoretical - it works today.**
  Proven on real hardware, with real numbers, under real constraints.

- **The opportunity window is now, before fragmentation becomes permanent.**
  Early standard alignment determines who becomes a reference point.

- **RIoT Secure is positioned to lead, not follow, this transition.**
  With a deployed platform, a working runtime, and a published ABI, we are ahead of the consolidation curve - not waiting for it.

The Call to Alignment

This is not a solitary proposition - but a collective inflection for the industry. We invite:

- **Module vendors & silicon providers** - integrate before standards harden elsewhere
- **Platform operators & integrators -** deploy WASM without rewriting toolchains
- **Investors & strategic buyers** - participate before this layer consolidates
- **Standards collaborators** - build on a concrete, working proposal, not abstractions

The future of IoT will **not be won by whoever ships the next board - but by whoever defines what runs on them, safely, portably, and at scale**. This is not about adding another VM to IoT - it is about deciding what will run on devices for the next decade and who controls that layer.

References

https://riotsecure.se/
https://riotsecure.se/brawl

Blog Posts

For additional technical background, implementation details, and supporting analysis behind the concepts presented in this whitepaper, see the following RIoT Secure engineering blog posts:

https://riotsecure.se/blog/brawl_meets_webassembly_standardization_for_a_smarter_iot
https://riotsecure.se/blog/riot_secure_webassembly_on_arduino_uno_r4_wifi
https://riotsecure.se/blog/wasi_current_state_and_roadmap
https://riotsecure.se/blog/wasm_binary_size_in_high_Level_languages
https://riotsecure.se/blog/wasm_zero_bloat_demos_in_c_rust_and_tinygo
https://riotsecure.se/blog/wasi_universal_device_io_abi

The Next IoT Cycle - WebAssembly at the Bottom of the Stack

August 2025
Author: Aaron Ardiri

RIoT Secure AB
www.riotsecure.se