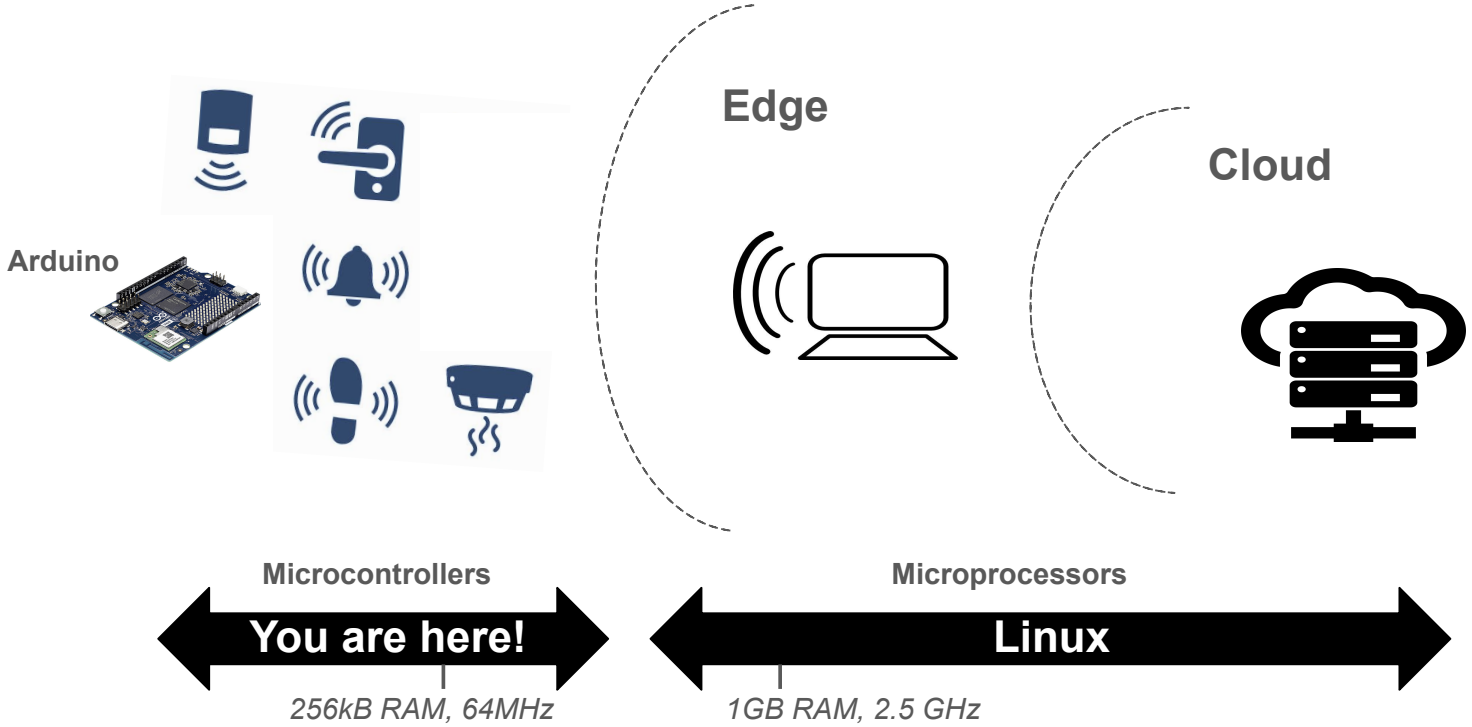
treVM: Tiny Rust Embedded Virtual Machines

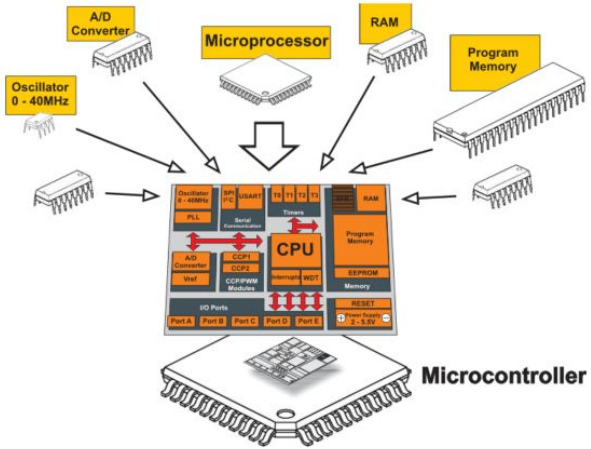
WASM on Variable Resource-Constrained Hardware

Antoine Lavandier, Bastien Buil, Emmanuel Baccelli, Chrystel Gaber

Which embedded are we talking about ?



Low-power Hardware Characteristics



- Dual or single Core
- Single memory space ⇨ no user- vs kernel-space, no MMU
- Baremetal ⇨ No underlying OS, threads or heap allocation

- Low flash memory ⇨ Hundreds of kB
- Low RAM ⇨ Dozens to hundreds of kB
- Low power ⇨ Multi-month battery life

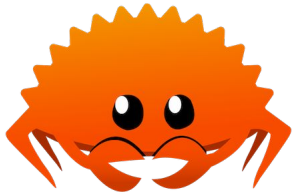
⇨ **Main 32 bit architectures: Cortex-M, Risc-V 32, Xtensa**

⇨ **Popular boards: RPi Pico, Arduino, BBC:microbit, ESP32...**

Table of contents

I - Rust for Embedded

Why Rust is on the Rise



- **Memory safety by design removes some of the most common vulnerabilities (use-after-free, data races, buffer overflows *etc.*)**
- **Lively ecosystem**
- **Modern and standardized tooling with cargo.**

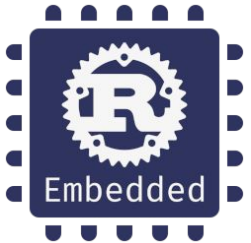
Backed by data:

- Since 2025 Rust is the first language in annual net new LoC in Android [1]
- Rust in the Linux Kernel is no longer experimental since December 2025 [2]

[1] https://blog.google/security/rust-in-android-move-fast-fix-things/
[2] https://lwn.net/Articles/1049831/

The Embedded Rust Ecosystem

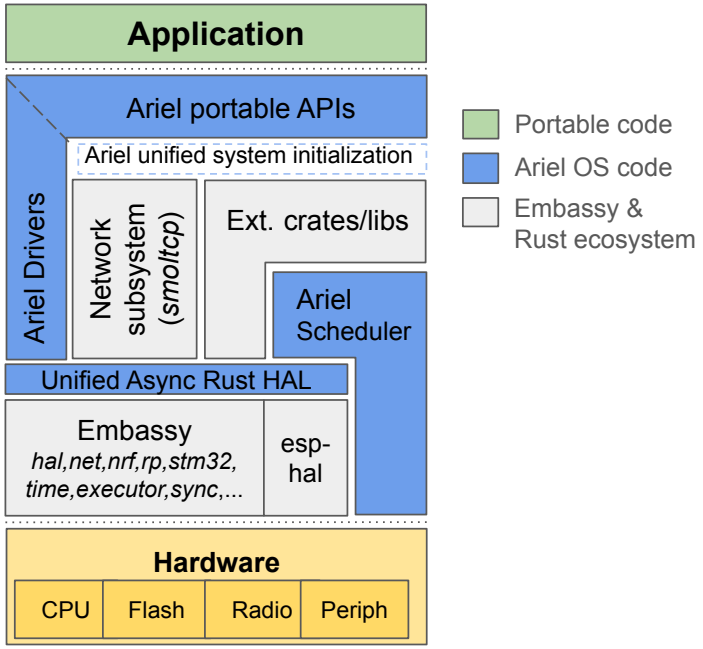
- **Rust libraries: the crates ecosystem (crates.io)**
- **Peripheral access crates (stm32-metapac, *etc.)***
- **Hardware Abstraction Library (Embassy, esp-hal…)**
- **Rust Embedded Working group [3]**
- **Operating systems: Tock OS [4] , Ariel OS [5]**
  - also marginal Rust support in C operating systems: RIOT[18] or Zephyr[6]



[3] https://github.com/rust-embedded/wg
[4] Levy *et al.* Multiprogramming a 64kB Computer Safely and Efficiently. SOSP 2017.
[5] Frank et al (2025). Ariel OS: An Embedded Rust Operating System for Networked Sensors & Multi-Core Microcontrollers. IEEE DCOSS-IoT, 2025.
[6] https://github.com/zephyrproject-rtos/zephyr-lang-rust
[18] Baccelli, E. et al (2018). RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. IEEE Internet of Things Journal, 5(6)

Ariel OS, an Embedded library OS in Rust

- **Library OS leveraging the Rust ecosystem**
  - Asynchronous execution by default
  - Pre-emptive scheduler if needed
- **Cross-hardware unified abstractions**
  - Espressif Systems
  - nRF Semiconductors
  - Raspberry Pi Holdings
  - STMicroelectronics
  - ...



II - WebAssembly in Embedded

WebAssembly: virtualization suitable for embedded!

- Portable, easy to decode and small(-ish) bytecode
- Strong memory isolation properties by design
- Simple memory model based on 64 kiB or 1 B pages
- Compile- and run-time type-checking

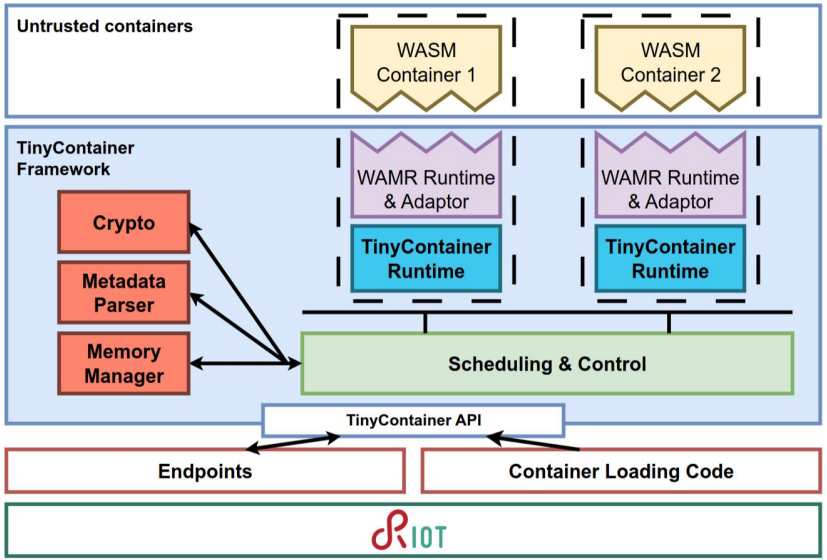
Studies [7, 8] show WASM is faster and less memory intensive than other small VMs such as MicroPython, Lua or JavaScript

[7] K. Moron and S. Wallentowitz, “Benchmarking webassembly for embedded systems,” ACM Trans. Archit. Code Optim., vol. 22, no. 3, Sep. 2025. [Online]. Available: https://doi.org/10.1145/3736169
[8] K. Zandberg et al., “Femto-Containers: Lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers,” in ACM/IFIP Middleware, 2022.

WebAssembly: virtualization suitable for embedded!

Prior work [9, 10] uses WASM (WAMR) for updatable application containers atop RIOT

⇨ hosting untrusted 3rd party business logic



[9] Buil et al. TinyML as a Service on Multi-Tenant Microcontrollers. *EWSN 2025.*
[10] Buil et al. TinyContainer: Container Runtime Middleware Enabling Multi-tenant Microcontrollers with Built-in Security. ACM *WiSec, 2026.*

The million dollar question(s)

We surveyed maintained, open-source & embedded capable runtimes :

- WAMR [11], Wasmtime [12], Wasm-interpreter [13], Wasefire [14], Wasmi [15]...

  Q1: Which runtime should you actually use?

Aiming for a Rust embedded platform to host/update WASM 3rd party logic

Q2: What prototype can we come up with ?

[11]: https://github.com/bytecodealliance/wasm-micro-runtime
[12]: https://github.com/bytecodealliance/wasmtime
[13]: https://github.com/DLR-FT/wasm-interpreter
[14]: https://github.com/google/wasefire
[15]: https://github.com/wasmi-labs/wasmi

III - WASM Runtime Benchmarks

Benchmarking Setup (February 2026)

- **Running on top of Ariel OS for portability and timing drivers**
- **Two benchmark suites: Embench 1.0 [16] and CoreMark [17]**
- **4 MCUs from 3 manufacturers and covering 3 ISAs :**
  - RP2350 (ARM Cortex-M33 ,Raspberry Pi Holdings)
  - nRF52840 (ARM Cortex-M4, Nordic Semiconductors)
  - ESP32-C6 (RISCV 32 Core, Espressif Systems)
  - ESP-WROOM32 (Dual LX6 Xtensa cores, Espressif Systems)
- **7 runtimes configurations:**
  - WAMR fast and regular interpreters (v 2.3.1)
  - Wasmtime's regular and No SIMD Pulley interpreters (v 38.0.3)
  - Wasmi (v 0.51)
  - Wasm-interpreter (v 0.1)
  - Wasefire (v 0.5)

[16]: D. Patterson et al., "Embench: Recruiting for the long overdue and deserved demise of dhrystone as a benchmark for embedded computing," Computer Architecture Today Blog, Jun, vol. 11, 2019.
[17]: S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," The Embedded Microprocessor Benchmark Consortium, vol. 6, no. 23, p. 87, 2012.

Benchmark results: CoreMark

- **Wasefire and Wasm-interpreter score 10 to 100 times less**
- **Wasmi and Wasmtime-No-SIMD share the top stops**

- **Wasmi is quite bloated:**
  - 60% of total Flash on the nRF52840DK
  - 1.6x times the next highest footprint
- **Both of WAMR's interpreters and Wasefire are comparatively small**
  - Roughly half as big as the next options

Table I: CoreMark scores

	RP2350	ESP-WROOM-32	nRF52840	ESP32-C6
WAMR	6.6	6.7	1.8	5.3
WAMR Fast	10.8	11.0	3.8	8.0
Wasmi	15.5	17.3	4.8	14.5
Wasmtime	13.2	11.6	3.5	9.9
Wamstime No SIMD	18.6	11.7	5.1	12.3
Wasm Interpreter	0.6	0.5	0.2	0.6
Wasefire	0.2	0.3	0.06	0.3

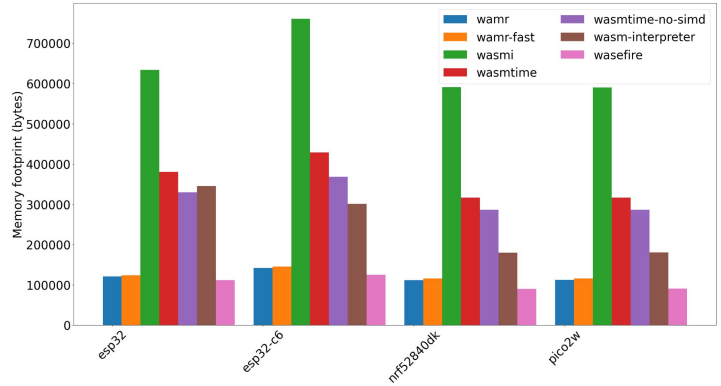


Fig. 1: Flash memory footprint when running CoreMark

Benchmark results: Embench

- Wasmi, Wasmtime-no-SIMD and WAMR-fast are consistently top 3

- Wasmi and WAMR-fast show unacceptably high peak RAM usage
- Wasmtime-no-SIMD shows consistent peak RAM across snippets

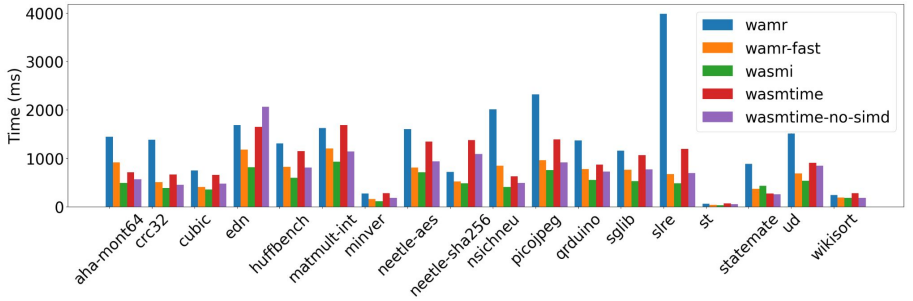


Fig. 2: Embench execution times on the RP2350

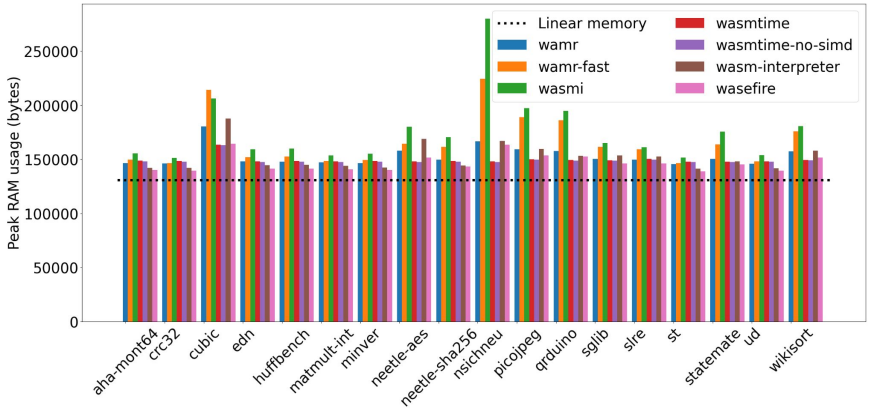


Fig. 3: Embench peak RAM usage on the RP2350

Benchmark conclusions

For our purpose we chose Wasmtime-no-SIMD:

- Good compromise between speed and flash footprint
- Predictably good RAM requirements compared to WAMR-fast

Also:

- Written in Rust ⇨ Cleaner integration on top of Ariel OS
- Developed by the Bytecode Alliance, a leader in WebAssembly standardization efforts

What changed since February ?

New versions :

- Wasmtime v38 -> v48
- Wasmi v 0.51 -> v 1.1 (with a beta v2)
- Wasm-interpreter -> v0.1 to v0.2. Now named dlr-wasm-interpreter

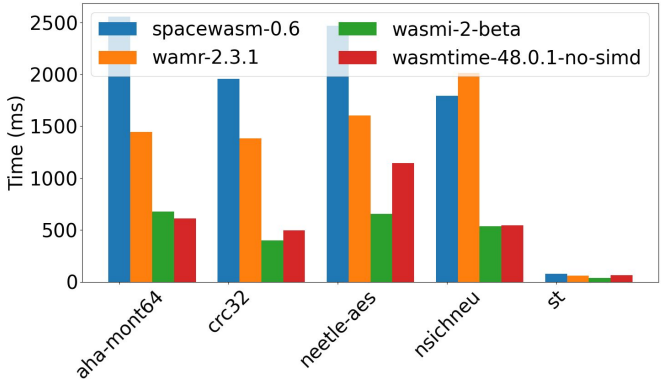
New runtime:

- SpaceWasm (https://github.com/nasa/spacewasm) from Nasa JPL

This warrants an update !

What changed since February ?

- **Wasmtime:** Bigger **(+15%)** while keeping the same performances.
- **Wasmi: v2** is significantly smaller **(-23%)** but still **above 450 KB.** Only works on ARM.
- **Wasm-interpreter v0.2.0** asks for **1.5 MB** of RAM.
- **SpaceWasm:** Worse than WAMR in every way (but the source code language).
  No support for most of the post WASM 1.0 specs

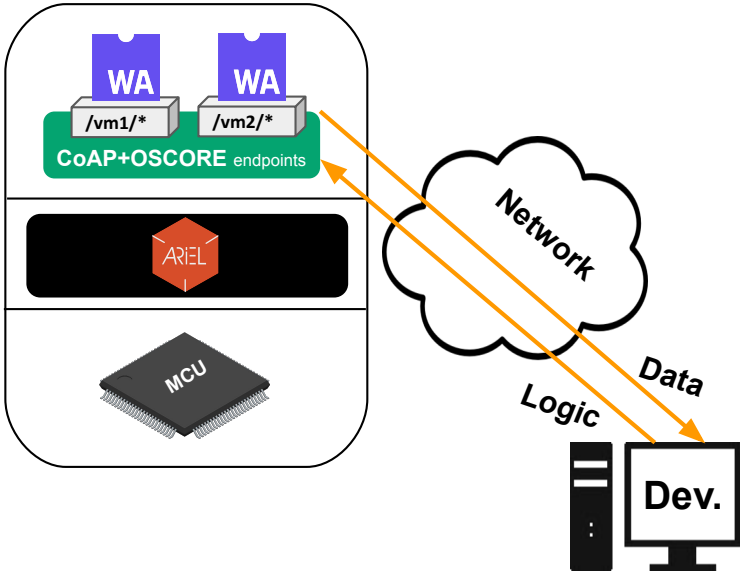


No reasons to change runtimes for now !

IV - treVM prototype

treVM Prototype Design

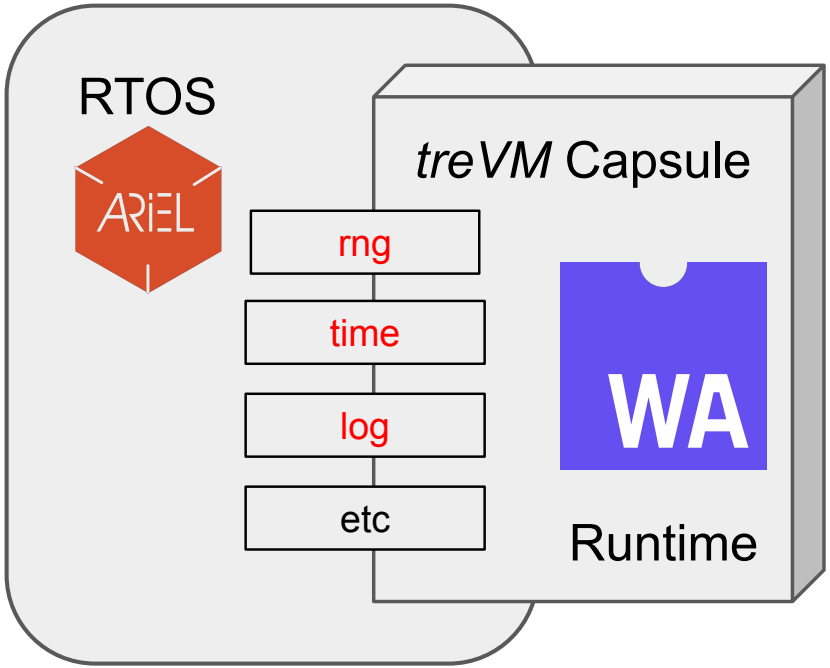
- **Sandboxed WebAssembly capsules running on Wasmtime**
- **Communication using CoAP + OSCORE [18] for secure and authenticated request and updates**
- **Two types of capsules:**
  - Ephemeral FaaS type of capsules
  - Persistent capsules acting like GET-able CoAP resources



[18] OSCORE, RFC8613, https://www.rfc-editor.org/rfc/rfc8613.html

treVM Capsule OS Integration

- Use of WebAssembly component model to segment functionality
- Per-API permissions using WIT descriptions enforced at instantiation time
- Portable bindings (from the OS)



treVM Prototype Performance

Measurements on an nRF52840 dev kit board (Cortex-M microcontroller)

Table II: Memory footprint of treVM (in kilobytes) using 32 KiB linear memory capsules

Capsule Type	Capsule Size	Flash *treVM*	RAM *treVM*	Flash total	RAM total
Ephemeral w/o bind.	9.6	≈208	≈40	560	60
Ephemeral	28.8	≈228	≈60	608	94
Persistent w/o bind.	34.5	≈212	≈70	600	117
Persistent	43.2	≈230	≈80	626	130
Minimal CoAP Server	N/A	N/A	N/A	208	29

treVM Flash and RAM footprint are substantial, mostly because of the runtime.

Example usage

-

IV - Conclusion

Summary & Conclusions

- **Our benchmarks of various WebAssembly runtimes using diverse microcontroller hardware highlight significant performance disparity**
  - We derive recommendations on which WASM runtime to use

- **We designed treVM to enable Rust-based hosting and OTA updates of sandboxed 3rd party WASM logic on heterogeneous microcontroller hardware**
  - treVM functionalities & characteristics can be useful in contexts such as eSIM

- **We published our implementation as open source**
  - Reproducible benchmark: full open source code available online at **https://github.com/anlavandier/ariel-wasm-runtime-comparisons**
  - treVM prototype implementation: full open source code available online at **https://github.com/ariel-os/trevm**

Perspectives & Future Work

- Updating benchmarks with newer runtimes and newer versions of existing runtimes (still)
- Adding more bindings to the treVM prototype, in conjunction with Ariel OS development
- Explore optimization opportunities to reduce the treVM related memory footprint.

SUIT with treVM is now possible thanks to @max-dau and @jhirschler

Thanks, Any Questions ?

Contacts:

antoine.lavandier@inria.fr

emmanuel.baccelli@inria.fr

https://arxiv.org/pdf/2604.27570

https://github.com/ariel-os/trevm