FACULTY OF ENGINEERING AND ARCHITECTURE

Cyber-Physical WebAssembly: Interfacing with USB and I2C Hardware

FOSDEM 2025

GHENT UNIVERSITY

imec

Co-funded by The European Union

6GSNS

Dr. ing. Merlijn Sebrechts

merlijn.sebrechts.be/about

Senior researcher @ imec

- Software delivery & trust in clouds and on devices

Lecturer @ Ghent University

- Systems Design
- Computer & Network Security
- Cloud

Open Source & Standardization

- Ubuntu Community Council
- Snapcrafters
- W3C WebAssembly System Interface (WASI)

FACULTY OF ENGINEERING AND ARCHITECTURE

DEPARTMENT OF INFORMATION TECHNOLOGY
IDLab

WebAssembly for IoT Devices

Interfacing with USB and I2C Hardware

GHENT UNIVERSITY

imec

Co-funded by
The European Union

6GSNS

Average lifespan of cars in Europe is **30 years**


Microsoft
Windows 95

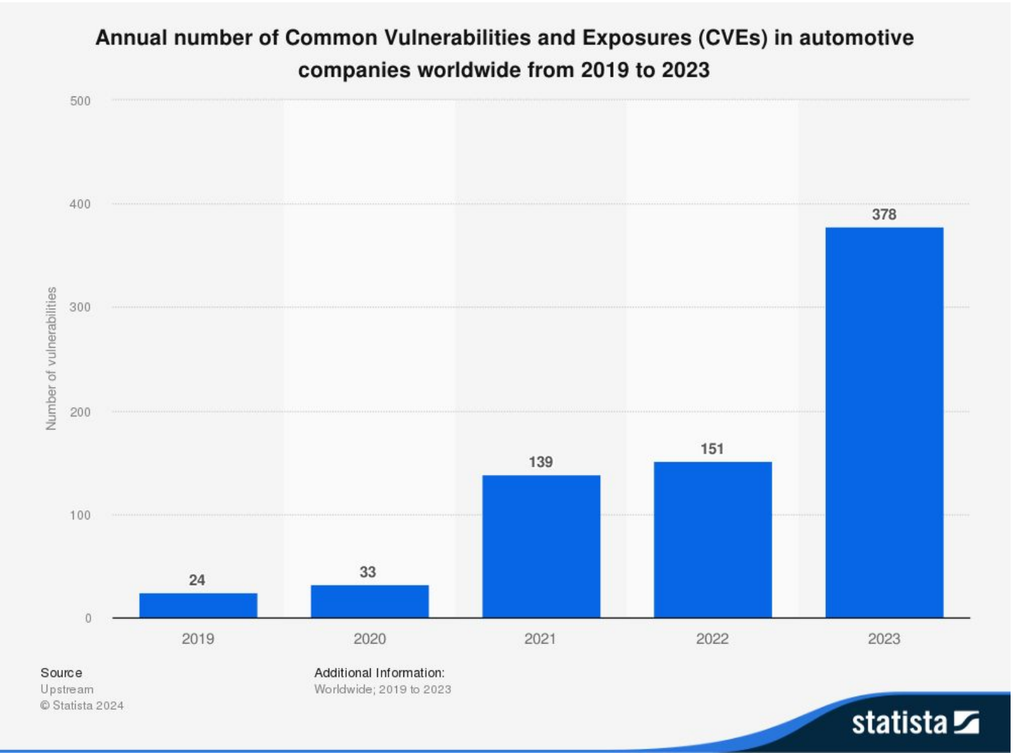
How do you update the software on a car that uses a compiler for Windows 95?

Very slowly..

Most commonly used packages over 10 years old

Those with an asterisk have appeared on this list in prior years as well.

zlib*	libpng*
openssl*	boost
gcc*	qt
sqlite	ncurses*
android_framework_native	curl



Most notorious CVEs still found in automotive technologies in order of frequency and with criticality

vuln_id	Package	Severity
CVE-2018-25032	Zlib	High
CVE-2022-37434	Zlib	Critical
CVE-2023-45853	Zlib	Critical
CVE-2023-4039	gcc	Medium
CVE-2023-3446	OpenSSL	Medium

TECHNOLOGY

The Kia Challenge, explained

How a carmaker's mistake created the ultimate internet challenge.

By Sara Morrison | sara@vox.com | Updated Jun 8, 2023, 4:44pm EDT



Sara Morrison is a senior Vox reporter who has covered data privacy, antitrust, and Big Tech's power over us all for the site since 2019.

It's safe to assume that 17-year-old Markell Hughes wasn't too worried about getting caught for stealing cars last year. After all, he lives in Milwaukee, **where just** 11 percent of reported car thefts resulted in an arrest in 2021 and only 5 percent were prosecuted. But Hughes

WebAssembly and WASI for embedded?

Current advantages compared to native:

- **Binary device portability** across ISAs (Instruction Set Architectures) and platforms
- Support for **more programming languages** and **language interoperability** on embedded devices
- **Forward compatibility** with newer application toolchains over multiple decades
- Secure and **sandboxed execution of software**, where other solutions like containers do not fit

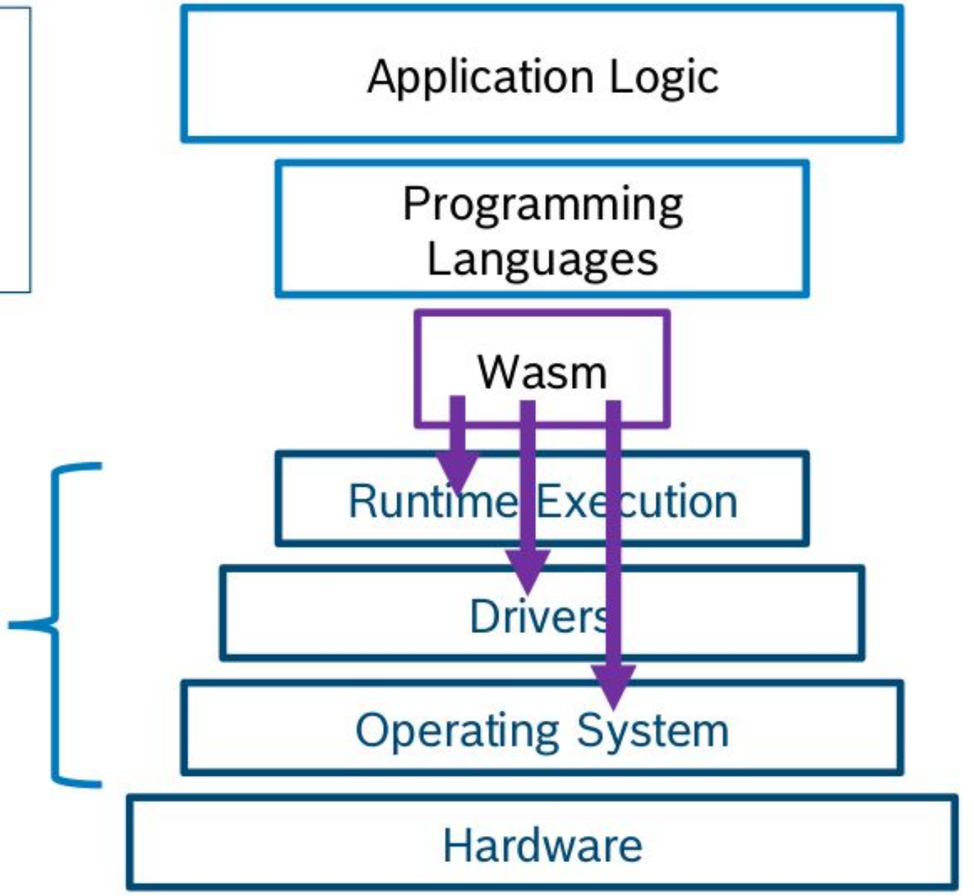
While still ensuring:

- Support for existing (pre-WebAssembly / pre-WASI) software
- Near-native efficiency in execution and compilation

Wasm can act as the “narrow-waist” of automotive software

Opportunity

Use WebAssembly to fully decouple applications and underlying hardware



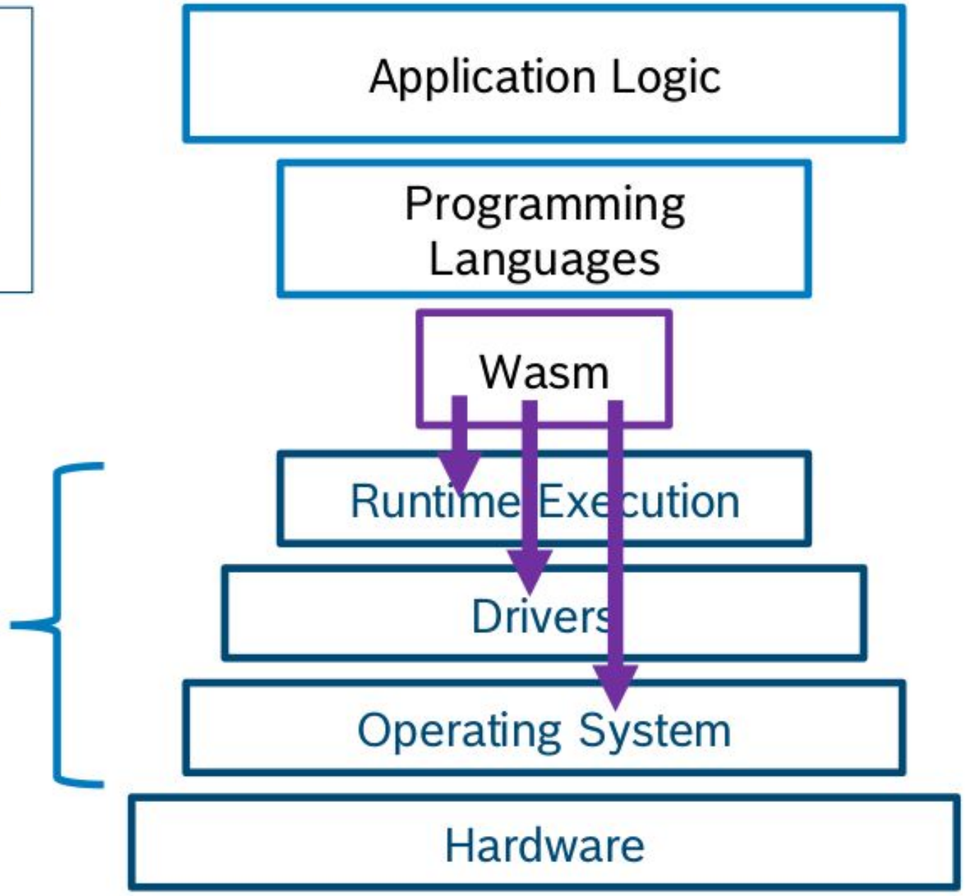
Challenges

- Define unifying interface to underlying software layers
- Maintain safety and real-time guarantees

Wasm can act as the “narrow-waist” of automotive software

Opportunity

Use WebAssembly to fully decouple applications and underlying hardware



Challenges

- Define unifying interface to underlying software layers
- Maintain safety and real-time guarantees

BOSCH

FACULTY OF ENGINEERING AND ARCHITECTURE

Cyber-physical WebAssembly

Connecting WebAssembly applications to hardware

GHENT UNIVERSITY

imec

Co-funded by The European Union

6GSNS

Goals

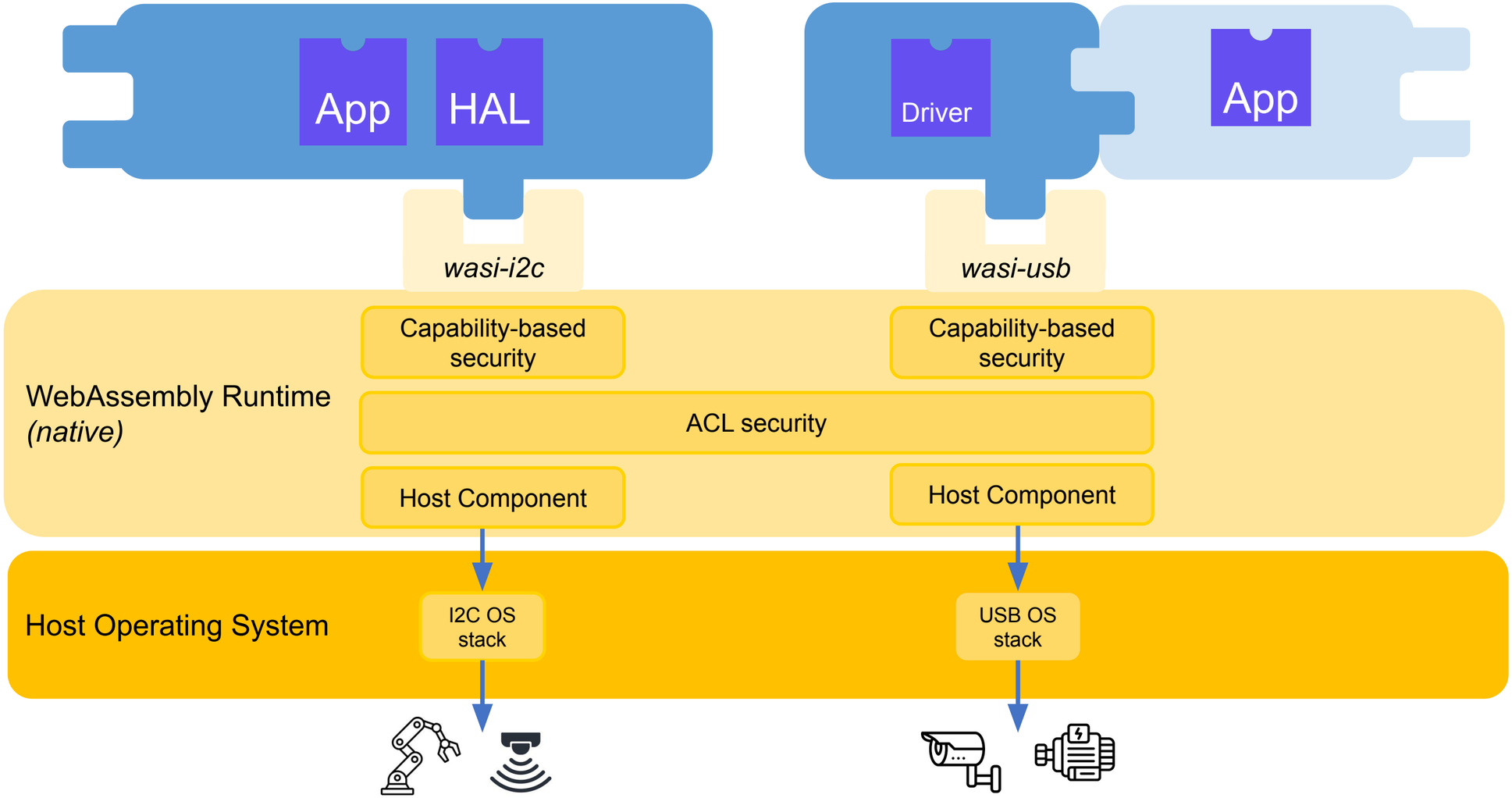
- Making it possible to use WebAssembly for IoT

- Secure drivers: only access exactly what they need
  - Defend against supply-chain attacks by sandboxing third-party drivers
  - Higher reliability and robustness by sandboxing components

- Portable drivers: any architecture, any platform
  - Support newer hardware on older platforms
  - “Write a driver once, run anywhere”

Cyber-physical WebAssembly

Hardware WASI interfaces & Componentized drivers



wasi-usb interface (phase 1) https://github.com/WebAssembly/wasi-usb/blob/main/wit/device.wit

Based on libusb (instead of WebUSB)

- Close to hardware
- Powerful
- Compatible*

```
1   package component:usb@0.2.0;
2
3   interface usb {
4       use types.{device-handle-error};
5       use descriptors.{configuration-descriptor, device-descriptor};
6
7       type duration = u64;
8
9   >   resource usb-device {…
20      }
21
22  >   resource device-handle {…
53      }
54  }
```

```
42      read-interrupt: func(endpoint: u8, timeout: duration) -> result<tuple<u64, list<u8>>, device-handle-error>;
43      write-interrupt: func(endpoint: u8, data: list<u8>, timeout: duration) -> result<u64, device-handle-error>;
44
45      read-bulk: func(endpoint: u8, max-size: u64, timeout: duration) -> result<tuple<u64, list<u8>>, device-handle-er
46      write-bulk: func(endpoint: u8, data: list<u8>, timeout: duration) -> result<u64, device-handle-error>;
47
48      read-isochronous: func(endpoint: u8, timeout: duration) -> result<tuple<u64, list<u8>>, device-handle-error>;
49      write-isochronous: func(endpoint: u8, data: list<u8>, timeout: duration) -> result<u64, device-handle-error>;
50
51      read-control: func(request-type: u8, request: u8, value: u16, index: u16, max-size: u16, timeout: duration) -> re
52      write-control: func(request-type: u8, request: u8, value: u16, index: u16, buf: list<u8>, timeout: duration) ->
```

wasi-i2c interface (phase 2)

https://github.com/WebAssembly/wasi-i2c/blob/main/wit/i2c.wit

Based on embedded-hal

- Close to hardware
- Cross-platform (even Zephyr RTOS)
- wasi-embedded-hal crate

```
61    resource i2c {
62        /// Execute the provided `operation`s on the I²C bus.
63        transaction: func(
64            address: address,
65            operations: list<operation>
66        ) -> result<list<list<u8>>, error-code>;
67
68        /// Reads `len` bytes from address `address`.
69        read: func(address: address, len: u64) -> result<list<u8>, error-code>;
70
71        /// Writes bytes to target with address `address`.
72        write: func(address: address, data: list<u8>) -> result<_, error-code>;
73
74        /// Writes bytes to address `address` and then reads `read-len` bytes
75        /// in a single transaction.
76        write-read: func(address: address, write: list<u8>, read-len: u64) -> result<list<u8>, error-code>;
77    }
```

Preprint (submitted to IEEE/IFIP NOMS 2025)

https://doi.org/10.48550/arXiv.2410.22919

Cyber-physical WebAssembly: Secure Hardware Interfaces and Pluggable Drivers

Michiel Van Kenhove*, Maximilian Seidler†,
Friedrich Vandenberghe*, Warre Dujardin*, Wouter Hennen*,
Arne Vogel†, Merlijn Sebrechts*, Tom Goethals*,
Filip De Turck* and Bruno Volckaert*

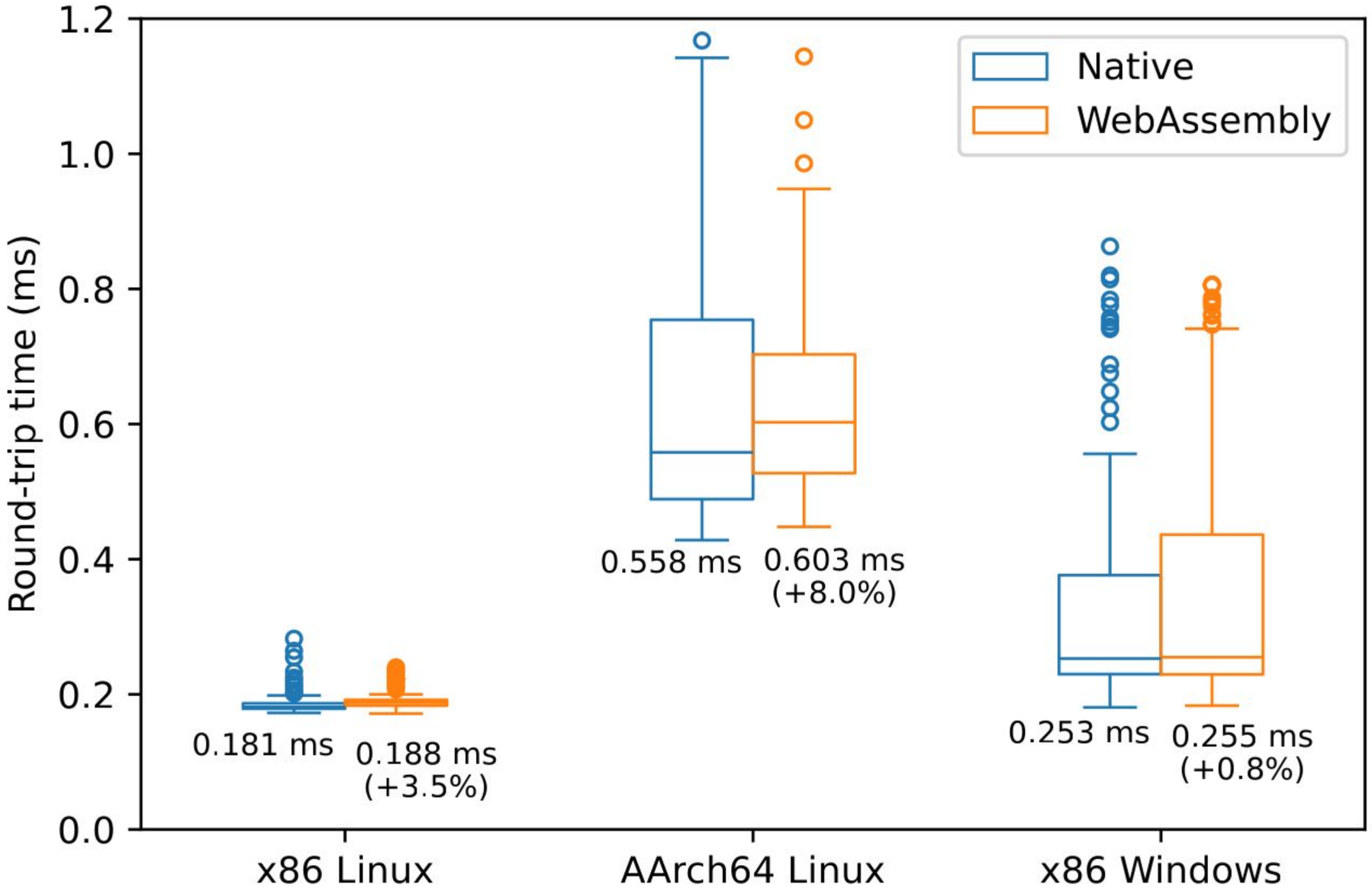
*IDLab, Department of Information Technology
Ghent University - imec, Ghent, Belgium
michiel.vankenhove@ugent.be

†System Software Group, Department of Computer Science
Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany
maximilian.seidler@fau.de

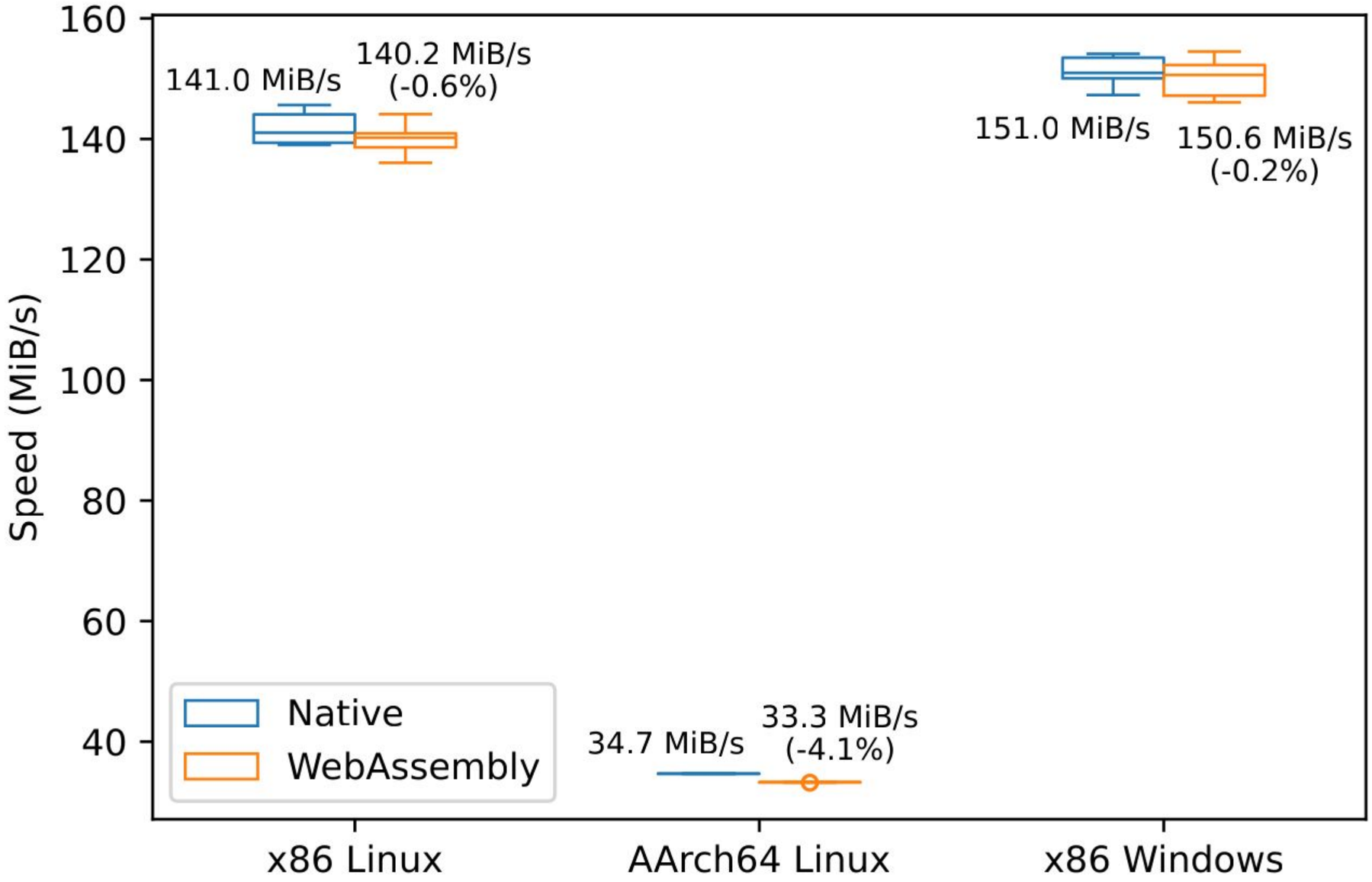
***Abstract*—The rapid expansion of Internet of Things (IoT), edge, and embedded devices in the past decade has introduced numerous challenges in terms of security and configuration management. Simultaneously, advances in cloud-native development practices have greatly enhanced the development experience and facilitated quicker updates, thereby enhancing application security. However, applying these advances to IoT, edge, and embedded devices remains a complex task, primarily due to the heterogeneous environments and the need to support devices with extended lifespans. WebAssembly and the WebAssembly System Interface (WASI) has emerged as a promising technology to bridge this gap. As WebAssembly becomes more popular on**

the support period of the product, vulnerabilities are handled effectively and that security updates should be available to users for at least the time the product is expected to be in use. Moreover, the automotive industry is increasingly adopting the practice of wirelessly distributing software updates to vehicles, known as over-the-air updates, where security is of critical importance [6], [7]. In parallel, Industrial Internet of Things systems often feature devices with operational lifespans of more than 30 years [8], [9]. To ensure forward compatibility of these systems, they must be able to integrate with newer hard-

Latency to USB device: +0.007ms



USB throughput: -0.6%



Ongoing work at imec & Ghent University

- I2C WASI proposal: Phase 2
  - Proposal: https://github.com/WebAssembly/wasi-i2c
  - Implementation: https://github.com/idlab-discover/i2c-wasm-components
  - Collaboration with Siemens

- USB WASI proposal: Phase 1
  - Proposal: https://github.com/WebAssembly/wasi-usb
  - Implementation:
    - https://github.com/idlab-discover/usb-wasm
    - https://github.com/Wouter01/USB_WASI

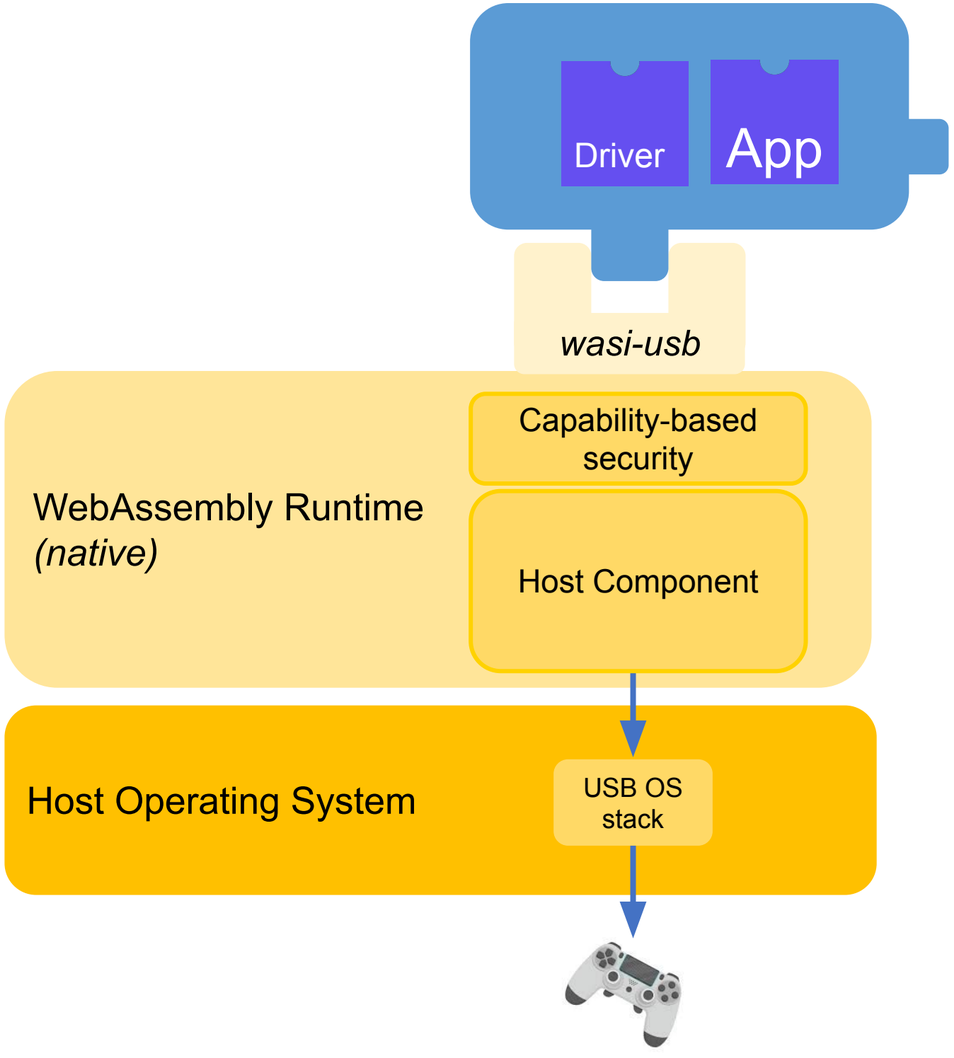
- GPIO WASI proposal (in development)
  - Proposal: https://github.com/WebAssembly/wasi-digital-io
  - Implementation: https://github.com/emielvanseveren/gpio-wasm-components

- SPI WASI proposal (in development)
  - Proposal: https://github.com/WebAssembly/wasi-spi

Demo time!

Cyber-physical WebAssembly

Xbox controller usb driver + pacman in wasm with wasi-usb



Q & A

Thanks to

Michiel Van Kenhove, Maximilian Seidler, Friedrich Vandenberghe, Warre Dujardin, Wouter Hennen, Arne Vogel, Merlijn Sebrechts, Tom Goethals, Filip De Turck, Bruno Volckaert

Valentin Olpp, Dan Gohman, Emiel Van Severen

Bytecode Alliance & W3C WASI subgroup

EU ELASTIC project (101139067) from Horizon Europe SNS JU

Contact: merlijn.sebrechts@ugent.be
Follow: https://www.linkedin.com/in/merlijn-sebrechts/

GHENT UNIVERSITY

imec

Co-funded by The European Union

6GSNS

FAQ: WebAssembly vs Java runtime?

Many similarities both in design and use-cases

- “Write once, run anywhere”
- Architecture-independent bytecode
- JVM on microcontrollers: Johnson Controls heat pumps
- JVM in browser: Java Applets (vSphere web UI)

But ultimately, JVM failed in most of these fields. It remains a single-vendor runtime for a single language family.

Why WebAssembly instead of JVM

WebAssembly learned from the 20+ years of JVM experience.

- Compilation target **for all languages**, standardized by W3C
  - JVM is too reliant on a single vendor and too focussed on a single language family
  - JVM is too opinionated about languages: e.g. requires classes and garbage collector
- **Sandboxed by default** with capability-based access to outside world
  - JVM apps have too much access to underlying OS, resulting in security and portability nightmares.
  - JVM assumes all code is trusted
- **Software delivery baked-in:** Streamability, Hotplugging
  - JVM is too focussed on traditional applications consisting of a single, static, monolithic app already on the user's machine.

Gotcha #1: Language support

Language support varies, but improving rapidly

- Best supported
  - Rust
  - C, C++
- Some functionality doesn’t work
  - Python
  - Java
  - C#
- Important stdlib functionality doesn’t work
  - Go

Gotcha #2: Changing landscape of system interfaces

Which system interface is your compiler targeting?

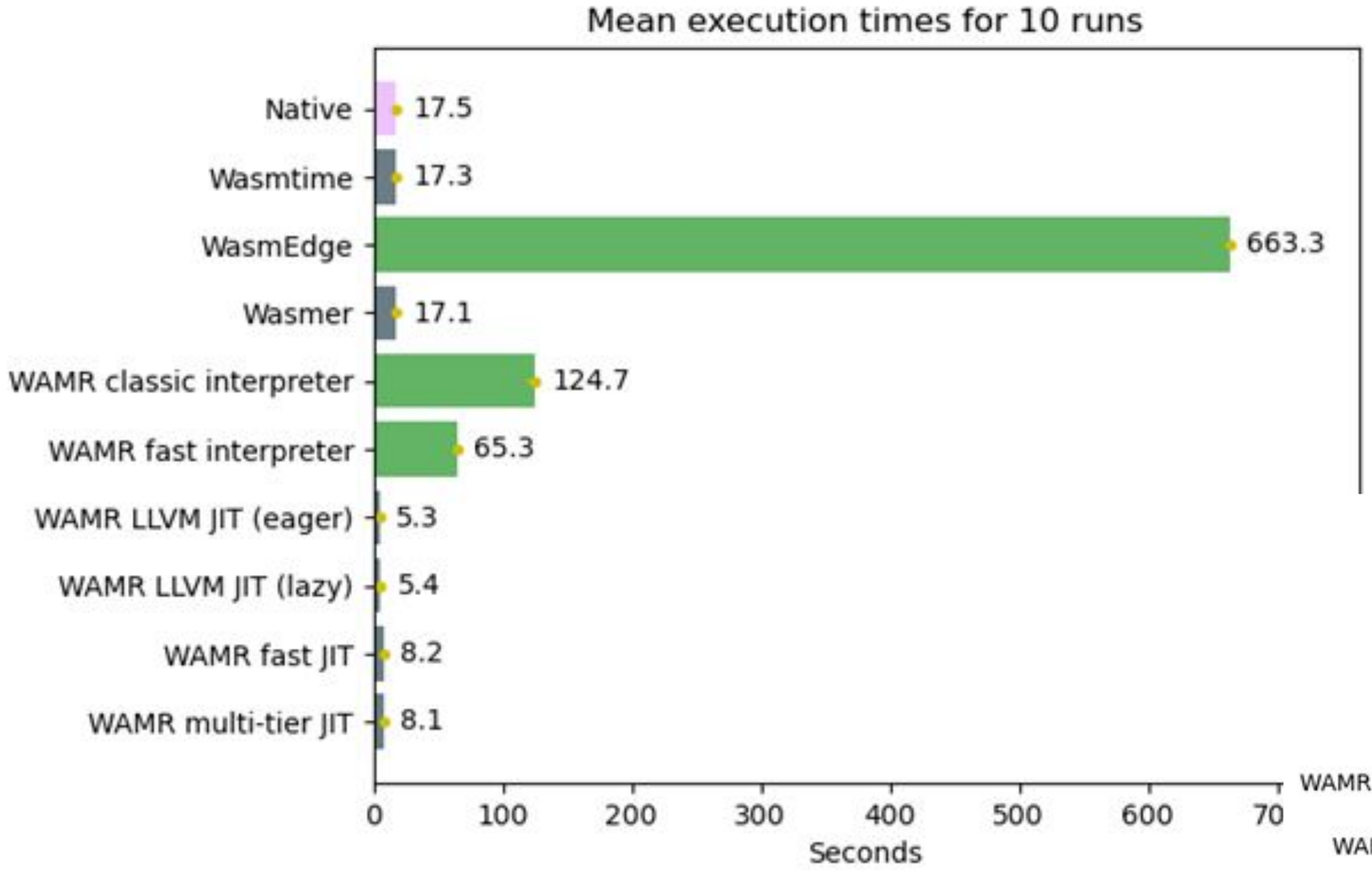
- emscriptem -> browser
- wasip1 -> legacy WASI
  - many non-standard “dialects” like wasmer
- wasip2 -> component model

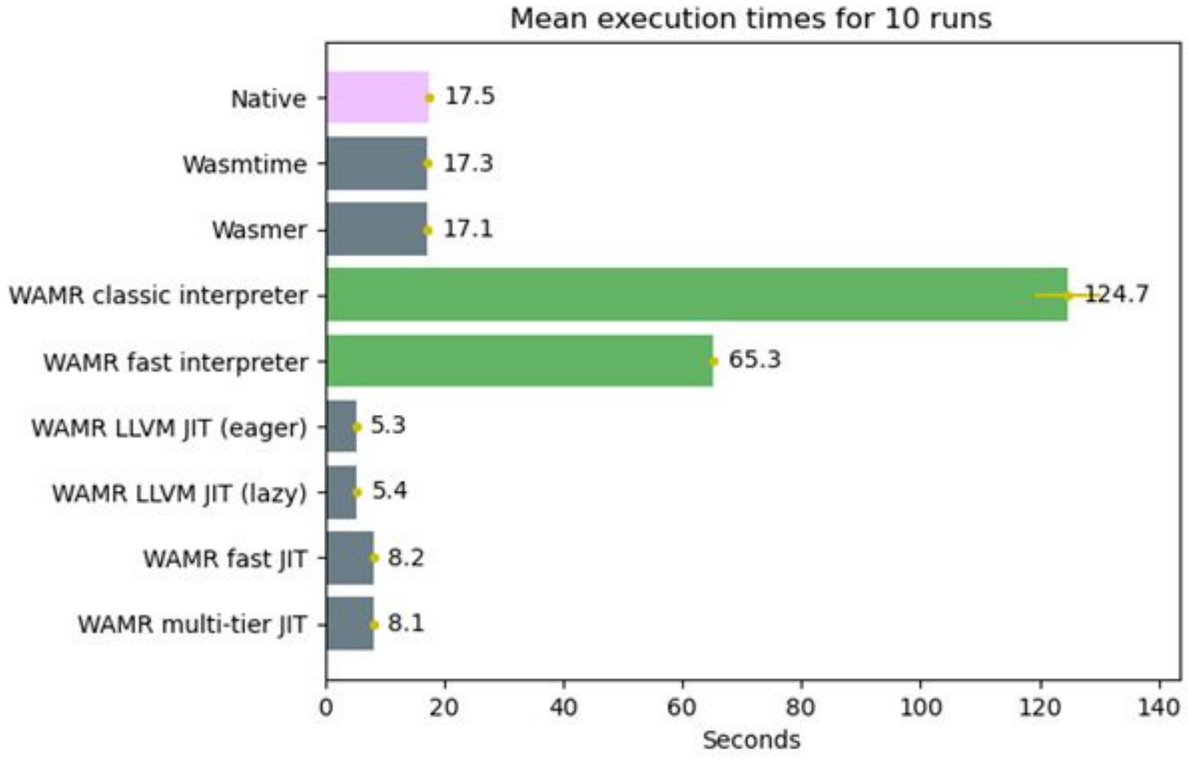
Gotcha #3: Varying support of runtimes

Does your runtime support the system interfaces?

- Recommended
  - Wasmtime: wasip2 & Component Model
  - WAMR: wasip1
- Not recommended
  - Wasmer: non-standard toolchains & WASI
  - Wasmedge: super slow

Gotcha #4: Runtime performance





GHENT UNIVERSITY

imec

Co-funded by
The European Union

6G