TinyML as a Service on Multi-Tenant Microcontrollers

Bastien Buil, Chrystel Gaber, Emmanuel Baccelli, Samia Bouzefrane

If you cite this paper, please use the EWSN 2025 reference: B. Buil, C. Gaber, E. Baccelli, S. Bouzefrane. *TinyML as a Service on Multi-Tenant Microcontrollers*. Proceedings of the 22nd International Conference on Embedded Wireless Systems and Networks (EWSN 2025).

TinyML as a Service on Multi-Tenant Microcontrollers

Bastien Buil*
Orange Research
Caen, France
bastien.buil@orange.com

Chrystel Gaber
Orange Research
Caen, France
chrystel.gaber@orange.com

Emmanuel Baccelli
TRiBE
Inria
France
emmanuel.baccelli@inria.fr

Samia Bouzefrane
Cedric
Cnam
Paris, France
samia.bouzefrane@lecnam.net

Abstract

Tiny Machine Learning (TinyML) allows the execution of small machine learning models on low-power devices like microcontrollers. TinyML-as-a-Service (TinyMLaaS) is an architecture to make the usage of TinyML models easier by having a platform that optimizes and compiles machine learning models according to the constraints of target devices, and then deploys the model code on microcontrollers. Within the Cloud-to-IoT continuum, both TinyML and multi-tenant microcontrollers focus on empowering microcontrollers and enabling on-device computing. Multi-tenant microcontrollers are designed to securely execute codes from mutually distrusting actors through the usage of lightweight software containerization solutions, like WebAssembly. In this paper, we propose to integrate TinyMLaaS with multi-tenant microcontrollers by using WebAssembly-based containerization, and we implement a proof-of-concept of the TinyMLaaS architecture based on WebAssembly Micro Runtime (WAMR) and RIOT-ML. In the second part of the paper, to improve the usage of containerized TinyML on microcontrollers, we propose CS4WAMR, a framework to enhance WAMR usage by enabling running simultaneously multiple instances of WAMR to allow better permission and memory consumption control.

CCS Concepts

• **Software and its engineering** → *Runtime environments*; **Virtual machines**; • **Computing methodologies** → *Machine learning*; • **Security and privacy** → **Software and application security**; • **Computer systems organization** → *Embedded systems*.

Keywords

IoT, Microcontrollers, TinyMLOps, WebAssembly, Cloud-to-IoT Continuum, Containers

1 Introduction

The development of containerization on microcontrollers, such as FemtoContainers [35] and Wasmico [28], enables microcontrollers to securely host services coming from multiple mutually distrusting entities. This cohabitation of software on microcontrollers creates multi-tenant microcontrollers.

Tiny Machine Learning (TinyML) is used to describe lightweight machine learning for resource-constrained devices [19], such as microcontrollers. By performing the processing locally, TinyML allows privacy, reduced network consumption, reduced latency, low cost, and enhanced reliability [15, 30]. To simplify the deployment of TinyML, Doyu et al. [14] proposes TinyMLaaS which consists of having a Cloud or Edge platform using model compilers to efficiently compile ML models, and deploy the compiled models on microcontrollers. TinyMLaaS allows to deploy models adapted to the CPU type, RAM and ROM sizes, peripherals, and underlying hardware of the target microcontroller.

On multi-tenant microcontrollers, TinyML can be a service providing machine learning services to other services on the device. To operate on multi-tenant microcontrollers, TinyML code must either be built in the firmware or become a containerized service on these multi-tenant microcontrollers. The former, by requiring the services to be integrated in the device firmware, lets less modularity to the TinyML user in the choice of the TinyML provider. The latter allows TinyML to have more modularity as it allows deploying TinyML code without custom integration with the host system, but TinyML code is constrained by the limitation of containerization such as runtime and memory overhead.

Liu et al. [23] proposes using WebAssembly (Wasm) to enable multi-tenancy on microcontrollers. Wasm provides a standardized format that can be obtained by compiling various programming languages such as C, Rust, and Python. Wasm also offers runtimes, like WAMR [1] and Wasm3 [5], which work on a wide range of devices, including low-end microcontrollers. Among WebAssembly runtimes, WAMR [1] is a runtime that supports low-end devices and is supported by a large consortium, the ByteCode Alliance[1].

This paper proposes to apply Tiny Machine Learning as a Service (TinyMLaaS) architecture to multi-tenant microcontrollers using WebAssembly-based containerization.

*Also with Cnam, Cedric.

[1] https://bytecodealliance.org/

Challenges TinyMLaaS on multi-tenant microcontrollers both creates new challenges and amplifies existing challenges from TinyML and multi-tenant devices.

C1 - Restricted Memory: Machine learning models are memory consuming, but TinyML should work on devices with restricted memory [15, 30]. This problem is amplified on multi-tenant microcontrollers as the memory is shared between multiple actors. Furthermore, some runtimes, like WAMR, do not allow defining memory consumption limits per container, thus TinyML services risk to have insufficient memory to do its computation.

C2 - Weight confidentiality: Model weights can be a business value that must be protected [27]. To maintain the confidentiality of model weights, it is essential to ensure that services remain isolated from each other. WebAssembly-based runtimes provides a first layer of software isolation, but it might be interesting to harden the runtimes using a layer of hardware isolation. Isolation hardening of a runtime for microcontrollers has already been proposed as by Bouffard and Gaspard [10] with the hardening of the Java Card Virtual Machine (JCVM) using MPU. However, we have identified that the memory architecture of WAMR is currently not compatible with this type of isolation.

C3 - Limited Processing Power: One challenge for TinyML is to be efficient with the low processing power of microcontrollers [30]. This problem is amplified by the containerization overhead.

C4 - Fault-tolerance and cyber-resilience of services: TinyML services, that provide TinyML service to other containers on the device, need to be resilient, even in case of corruption, or attack of the container due to the communication with other services. Even if WebAssembly provides isolation, it does not ensure the protection of the integrity of the running service [22].

C5 - Permission restriction for services: The principle of least privilege [29] consists of only granting the privileges needed to reduce the impact of compromised target. TinyML services, like any services on multi-tenant device, must only have the necessary access to the host system to respect the principle of least privilege.

Contribution: By applying Tiny Machine Learning as a Service (TinyMLaaS) architecture to multi-tenant microcontrollers, this paper proposes a TinyMLaaS architecture for multi-tenant microcontrollers using containerized TinyML. This paper then proposes a proof-of-concept to verify the practicability of containerized TinyML on microcontrollers by having a simplified TinyMLaaS architecture that produces TinyML Wasm containers to run model inference on microcontrollers. The proof of concept is based on RIOT-ML [18] for the compilation of machine learning models and WAMR [1] for running containers with Ahead-of-Time compilation.

Then, the paper looks at how to solve some constraints of Tiny Machine Learning as a Service (TinyMLaaS) with WAMR. To do that, we propose CS4WAMR which is a framework that enables multiple instances of WAMR by implementing a swapping mechanism.

By using one instance of WAMR per container, CS4WAMR allows to utilize WAMR's built-in instance-wide configuration to set per-container memory limits and access control permissions. This approach also allows to pre-allocate memory to TinyML containers, preventing memory shortages due to other containers' overuse, and to ensure TinyML services have only the necessary access to the host system.

CS4WAMR adds a snapshot system, which allows to capture and restore snapshots of WAMR instance. By using one instance per container, this allows snapshot of containers. Similarly to the techniques used by Webster et al. [32] to protect containers on servers, this allows to ensure the fault-tolerance of containers by having the possibility to revert compromised containers. This system can be applied to TinyML container to assure their integrity.

This work has been done as part of a project working on hardware-assisted memory isolation of microcontrollers. We tried to reinforce the confidentiality of model weights by adding hardware-assisted memory isolation between containers to WAMR, similar to the proposition of Bouffard and Gaspard [10] on the Java Card Virtual Machine using the Memory Protection Unit (MPU) to isolate applets. However, we have identified that WAMR do not have delimited memory regions for each container, which makes WAMR not compatible with this type of isolation. To solve this problem, the framework segmentizes the memory used by WAMR, and allows to have delimited memory region for each container. This is a first step to have hardware isolation to strengthen the isolation between containers using hardware-assisted isolation.

In summary, the contributions of this paper are as follows:

(1) an architecture for Tiny Machine Learning as a Service (TinyMLaaS) on multi-tenant microcontrollers,
(2) a proof-of-concept to run containerized TinyML code on microcontrollers to verify the practicability of TinyMLaaS on multi-tenant microcontrollers using WebAssembly-based containerization,
(3) CS4WAMR which is a framework to use WAMR with multiples instances to tackle some limitations of the runtime for TinyMLaaS on multi-tenant microcontrollers.

Outline: Section 2 defines Tiny Machine Learning as a Service (TinyMLaaS) and WebAssembly-based multi-tenancy on microcontrollers. Section 3 presents our proposition of architecture for TinyMLaaS on multi-tenant microcontrollers, tests the feasibility of the architecture with WebAssembly Micro Runtime with a proof of concept, and highlights some limits of WAMR for TinyMLaaS. Section 4 proposes CS4WAMR framework that fixes those limits by allowing multiple instances of WAMR. Section 5 evaluates the performances of the proof-of-concept, as well as the performance overhead of the framework. Section 6 discusses the pros and cons of containerizing TinyML with WAMR and using CS4WAMR. Section 7 presents works on TinyMLOps and multi-tenancy on microcontrollers. Section 8 concludes with the next steps for TinyMLaaS on multi-tenant devices.

2 Background

2.1 TinyML

TinyML consists of running optimized ML models on ultra-low-power device [15] such as microcontrollers. Pre-trained Machine Learning models are definable in model files using different formats depending on the machine learning frameworks such as TensorFlow Lite Micro (TFLM), PyTorch, ONNX. These pre-trained models files can then directly be compiled to code runnable on microcontrollers using model compilers like Apache TVM [11], uTensor [4], Edge Impulse [25]. To make TinyML usage easier, the RIOT-ML toolbox

If you cite this paper, please use the EWSN 2025 reference: B. Buil, C. Gaber, E. Baccelli, S. Bouzefrane. *TinyML as a Service on Multi-Tenant Microcontrollers*. Proceedings of the 22nd International Conference on Embedded Wireless Systems and Networks (EWSN 2025).

[18] manages the deployment of model updates on devices and provides ready-to-use code generation using Apache TVM.

2.2 TinyMLaaS architecture

Doyu et al. [14] defines TinyML as a Service (TinyMLaaS) as a Cloud or Edge platform that simplifies the deployment of the TinyML model. TinyMLaaS platform works by first gathering information on the inference model to process and the target device, such as CPU type, RAM and ROM size, available peripherals, and underlying software. Then, according to the chosen model and the device constraints, the TinyMLaaS platform automatically chooses the most appropriate model compiler and compiler configurations. Finally, the platform compiles the model and provides ready-to-use images to run TinyML inference on microcontrollers with a standardized interface to interact with the model.

2.3 WebAssembly

WebAssembly (Wasm) is a binary instruction format designed for a sandboxed execution on WebAssembly virtual machines.

WebAssembly compilation. WebAssembly (Wasm) is designed to be compiled from a wide range of programming languages, such as C, Rust, Go and Python. The code obtained after compiling a source code to WebAssembly is called a module. A module uses three main types of memory. The runtime operand stack allows loading values on the stack and executing instructions that consume stack values. The linear memory, which is a continuous memory buffer, is used both for static values and stack allocation. The last type of memory is module global variables, which most often are only used by modules to share information with runtimes on the structure of linear memory.

WebAssembly on constrained devices. When running Wasm modules on constrained microcontrollers, usual runtimes are not appropriate because of the required computational resources. This is notably due to the specification of WebAssembly which defines the linear memory of WebAssembly modules as composed of memory page of 64 KB. This makes the tiniest linear memory possible as 64 KB of RAM memory, which is too much for low-end constrained devices. Some runtimes, such as WebAssembly Micro Runtime (WAMR) and Wasm3, address this issue by allowing to use less than a page of linear. Another alternative, which is proposed to be added to the WebAssembly specification and is implemented in runtimes such as Wasm3, is to allow custom page sizes[2].

2.4 WebAssembly Micro Runtime

This paper is focused on the WebAssembly Micro Runtime (WAMR). WAMR is a WebAssembly runtime dedicated to have a small footprint and work across a wide range of devices from microcontrollers up to the Cloud.

2.4.1 Memory of WebAssembly Micro Runtime. WAMR uses three types of memory to function: constant, static variables, and dynamic allocations.

We define the notion of WAMR instance as all memory used by WAMR to store its state and configuration, it is equivalent to the static variables of WAMR and any dynamic memory allocations. Due to the usage of static variables, WAMR is designed to have only one instance at a time.

[2]https://github.com/WebAssembly/custom-page-sizes

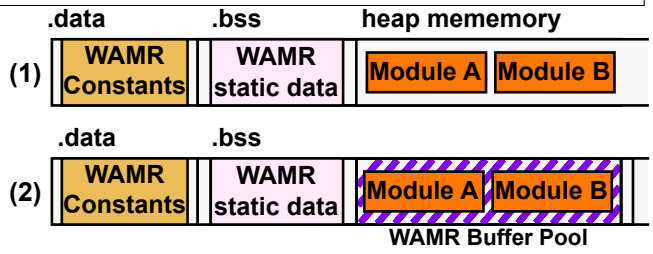


Figure 1: WAMR memory modes and memory usage of the runtimes. (1): *Alloc With System Allocator* memory mode. (2): *Alloc With Pool* memory mode.

To execute a WebAssembly module, WAMR dynamically allocates structures containing information about the modules and the execution. WAMR contains three memory modes that change how dynamic allocations are performed. The *Alloc With System Allocator* mode uses the malloc and free allocator functions provided by the system. The *Alloc With Pool* mode uses the WAMR built-in allocator functions and allocates from a buffer provided by the users. The *Alloc With Allocator* mode uses malloc and free functions given by the user. The organization of RAM memory of WAMR in the *Alloc With System Allocator* and *Alloc With Pool* modes is represented in the Figure 1.

While WAMR allows controlling its dynamic allocation, it does not allow configuring the allocation differently between modules. As a result, achieving a distinct separation between module dynamic allocations is challenging, since memory allocations from various modules can become entangled.

2.4.2 Acceding host system and inter-module communication in WAMR. When running WebAssembly modules in WebAssembly runtimes, modules are executed in an isolated environment and do not have access to the host environment. To solve this problem, virtual machines can give modules access to native functions running in the host environment. The native functions can be provided either by the runtime or by the user of the runtime. One limitation of the WAMR implementation for native functions is that the native functions are defined instance-wide, permitting unrestricted access to all the native functions for all the modules within that instance. Another functionality in some WebAssembly runtimes is the possibility to give some modules access to the exported functions of some other modules. WAMR implements the concept using registered modules, which are modules accessible by all the other modules running in the instance of the runtime.

2.5 Multi-tenant microcontrollers

Multi-tenancy describes a single system shared to multiple entities, known as tenants. Multi-tenant microcontrollers are shared microcontrollers that simultaneously execute applications from different entities on one device. Entities deploying services on microcontrollers are the Service Providers. Typical scenarios involve emergence of microservice-hosting platforms on microcontrollers [6]

and marketplaces for services deployable on microcontrollers. Service Providers are untrusted and may provide malicious applications trying to escape sandbox isolation, either to spy other applications, to interfere with their execution, or to attack the host system. Thus, a key requirement for multi-tenancy is maintaining service isolation [6, 35]. This is achievable via software containerization which uses lightweight virtualization or hardware methods to isolate software units, known as containers. A typical architecture for multi-tenant microcontrollers involves one or multiple microcontrollers with a container runtime and a container management agent, and one platform managing the deployment of containers on the microcontrollers.

Several commercial solutions such as MicroEJ [24] and Atym [7] propose containerization solutions for microcontrollers. Specifically, Atym promotes containers for ease of debugging and development, and for enabling isolation between applications for security. Atym notably proposes solution using containerized Edge AI. MicroEJ proposes a multi-application runtime to run applications in sandboxed environments and notably emphasizes servitization which consists of enabling third-party to develop services for the microcontrollers of another entity, for example through an app store. While enabling the personalization of a product through third-party applications allows a custom experience for the user, and thus increasing devices value, it also allows device manifacturers to monetize its devices through premium features. MicroEJ highlights its work in areas like smart homes, smart grids, industrial applications, and medical devices.

3 Architecture for TinyMLaaS on multi-tenant microcontrollers

This section introduces the Tiny Machine Learning as a Service (TinyMLaaS) on multi-tenant microcontrollers architecture. The first subsection details this architecture, while the second provides a proof of concept to evaluate WebAssembly containerization's feasibility for TinyMLaaS on these multi-tenant microcontrollers.

3.1 A new architecture of TinyMLaaS on multi-tenant microcontrollers

We propose to integrate TinyMLaaS (Tiny Machine Learning as a Service) in the multi-tenant microcontrollers architecture. In this architecture, the TinyMLaaS provider deploys optimized TinyML containers with a unified interface to interact with the machine learning model. The TinyML container can then provide TinyML services to other containers on the microcontroller.

The TinyMLaaS for multi-tenant microcontroller architecture, represented in Figure 2, works in five steps. In step ①, a service container or a service provider requests TinyMLaaS services for a container on the microcontroller and specifies the model type and device characteristics. Then, in ②, the TinyMLaaS provider requests to model providers a model that matches the requested needs, and then the selected model is given to the TinyML build service. In ③, the TinyML build service selects a model compiler, then optimizes and compiles the model to produce code to run the model. Then in ④, the code produced is transformed into a container that is given to the container management platform to be deployed on the microcontrollers. Finally in ⑤, the container

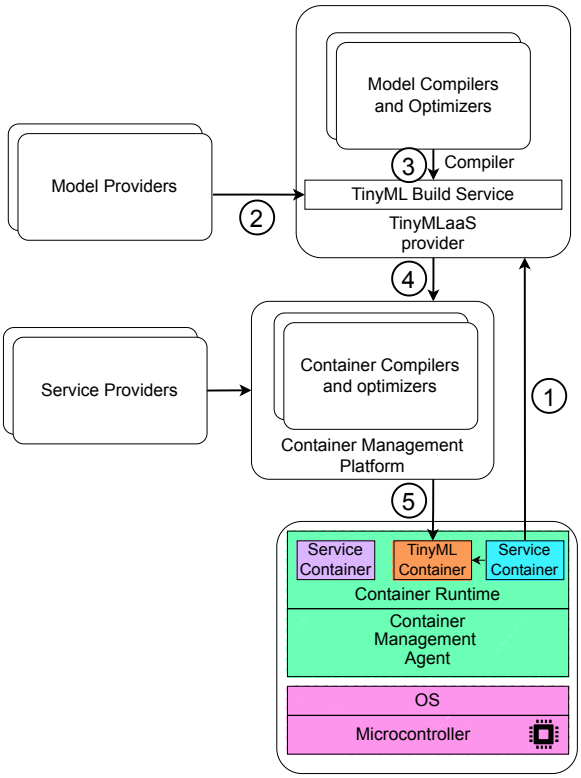


Figure 2: Architecture of TinyMLaaS on multi-tenant microcontrollers

management platform can either keep the container file as received or optimized and transform it into a format optimized for the device, as with Ahead of Time compilation.

This architecture allows to deploy as-a-service TinyML models that can be consumed by services on the microcontrollers. The architecture also allows the isolation and interoperability for the deployed TinyML services, while having a reasonable memory overhead.

3.2 Proof of Concept with WAMR and RIOT-ML

In the subsection, we propose a simplified proof of concept of TinyMLaaS on multi-tenant microcontrollers. This proof of concept allows testing the feasibility of containerized TinyML by having a build pipeline to create ready-to-be-run TinyML WebAssembly containers. The build pipeline uses the RIOT-ML toolkit for compiling the ML model, and WAMR Ahead of Time (AoT) compilation for transforming the WebAssembly container into optimized machine code. This subsection first presents model compilation with the RIOT-ML toolkit, then WebAssembly container compilation to machine code with WAMR Ahead of Time (AoT) compilation. Finally, the section presents the architecture of the proof of concept.

3.2.1 Model compilation with RIOT-ML toolkit. RIOT-ML[18] provides a toolbox for deploying, running, and benchmarking TinyML models. One RIOT-ML tool is a model compiler that leverages Apache TVM to produce a ready-to-be-compiled C code to perform

If you cite this paper, please use the EWSN 2025 reference: B. Buil, C. Gaber, E. Baccelli, S. Bouzefrane. *TinyML as a Service on Multi-Tenant Microcontrollers*. Proceedings of the 22nd International Conference on Embedded Wireless Systems and Networks (EWSN 2025).

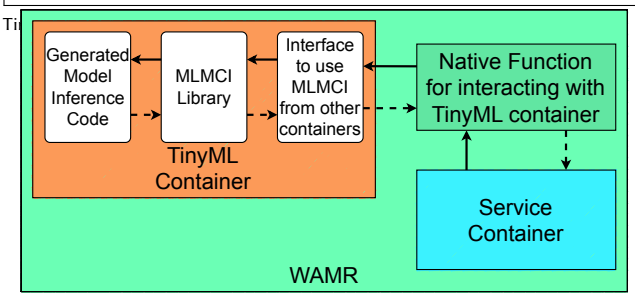


Figure 3: Example of a container using a TinyML container generated with our proof-of-concept

a machine learning inference with a model on microcontrollers. The generated code uses the RIOT-OS to produce the firmware. However, by modifying the build script and providing functions missing in WebAssembly such as some math functions, the generated code can be compiled to WebAssembly. One advantage of RIOT-ML is that the generated code uses the mlmci library, which provides an interface to interact with the underlying machine learning model.

3.2.2 WebAssembly Ahead of Time Compilation. The interpreter running mode of WAMR executes Wasm bytecode with some important performance overhead. To remedy this problem, while keeping the isolation advantages of WebAssembly, WAMR proposes the Ahead of Time (AoT) compilation running mode to have near native performances. Ahead of Time (AoT) compilation consists of compiling the Wasm bytecode to machine code dependent on the target hardware. In practice, AoT compilation consists in transforming .wasm file into .aot file which can be executed by WAMR in the AoT running mode. AoT gives performance gains but has an memory overhead both for the file of the container and the memory consumption of the runtime.

The memory overhead can be partially compensated by the Execution in Place (XIP) of AoT code with WAMR. When the WAMR interpreter executes WebAssembly bytecode, it modifies the bytecode for some optimizations. This forces the WebAssembly bytecode to be loaded in RAM when being executed. WAMR AoT compiler has an Execute In Place (XIP) mode to generate code that can be executed in place (XIP), i.e. without any modification. This allows to store the compiled Wasm module in flash, as it will not be modified. This allows to avoid having the model weights appear twice in RAM, with first the bytecode of the Wasm module and then the linear memory of the instance of the module.

3.2.3 Proof-of-Concept of the architecture. We have implemented a simplified proof of concept of TinyMLaaS on multi-tenant microcontrollers using RIOT-ML [18] as the model compiler and the WAMR compiler as the container compiler.

Our proof of concept proposes a building system that generates AoT-compiled WebAssembly containers from machine learning model files to perform machine learning inference of ML models. These containers are tailored to the device specifications and prepared for immediate use by the microcontroller using WAMR. First, the system generates WebAssembly containers from TinyML file by generating C code using RIOT-ML and modifying the build scripts. Then, the system compiles the WebAssembly container according to the device characteristics and produces a file that can be run using WAMR's Ahead-of-Time running mode.

Figure 3 illustrates a practical use case of a TinyML container produced with our proof of concept. The produced TinyML container is composed of the inference code generated by Apache TVM, the configured mlmci library generated by RIOT-ML to interact with the model and our code to use the mlmci library from other containers. Other containers on the device have the capability to perform inference with the produced TinyML container by using a native function tailored for interaction with TinyML containers. The native function facilitates the copy on the TinyML container memory of input values for the model and subsequently returns the model's prediction output values. All computations occur within the Wasm container.

3.2.4 Limits of WAMR for TinyMLaaS on multi-tenant microcontrollers. When implementing our proof of concept, we have identified limits in WAMR for using it for TinyMLaaS on multi-tenant microcontrollers.

First, WAMR built-in permission system to interact with the host system, which is native functions, is only configurable instance-wide, granting identical permissions to all the containers. This grants excessive privileges to containers, while others, such as our TinyML container, operate without needing any permissions.

Additionally, WAMR does not allow controlling the memory consumption of modules. Thus, a module can consume a lot of memory and prevent the subsequent initialization of other modules. This is especially problematic with TinyML containers that require a lot of memory.

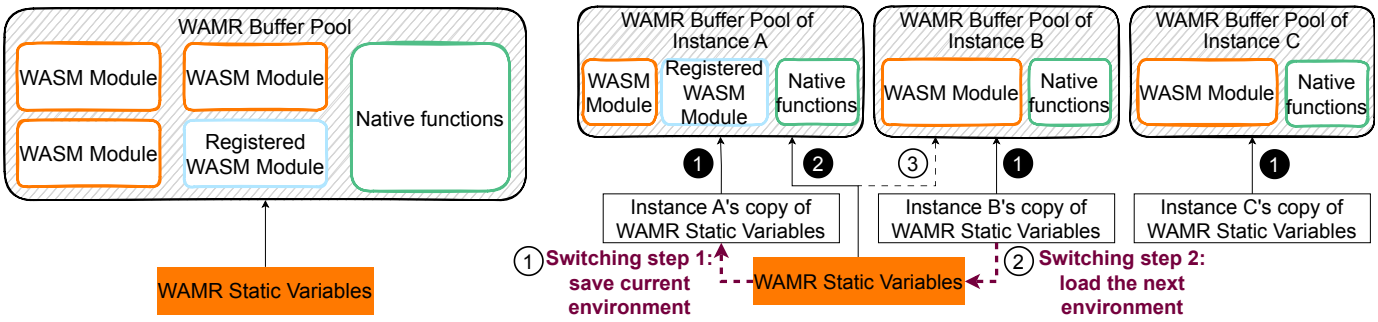
Furthermore, when implementing our proof of concept, we have identified a possible attack. When a TinyML container provides machine learning service to multiple containers, if the inter-module communication code is not correctly implemented, a container can corrupt the TinyML, like by providing too long values, and do either a Denied of Service (DoS) to other containers or corrupt the model so that it returns invalid values.

As proposed by Bouffard and Gaspard [10], hardware isolation can be used to reinforce the isolation between code on multi-tenant devices. This additional isolation is critical to reinforce the protection the model weights, which are the model providers business value, especially in case of an attacker escaping the isolation provided by WAMR. The WAMR memory structure, which assigns variables to the heap or pool allocator, complicates establishing distinct memory regions for each module because the memory a module uses is mixed with other WAMR allocations.

4 CS4WAMR: A Framework for enhancing WAMR for containerized TinyML through a swapping mechanism

Given the different limitations of WAMR for TinyMLaaS on microcontrollers, we propose CS4WAMR[3] (Context Switching for WAMR) which is a framework integrating an instance swapping mechanism to allow multiple instances of WAMR to run simultaneously on single-core systems. By separating the execution of Wasm modules

[3]The framework is available at https://github.com/Orange-OpenSource/CS4WAMR



(a) Original: WAMR organization in the *Alloc with Pool* memory mode

(b) Target: multiple WAMR instances each containing a module ● is for the initial state. ○ is for the switching steps.

Figure 4: Overview of WAMR memory organization with and without the swapping mechanism

in distinct instances of WAMR, this allows to have more control on the memory consumption and permission of each module. Also, by having dedicated memory space in RAM, this mechanism opens up to the implementation of snapshot mechanism and hardware isolation.

The first subsection presents the context switching mechanism used by the framework. The second subsection presents the framework structure with its static variable detection tools and its context-switching library.

4.1 Configuration-switching mechanism for WAMR

To create multiple WAMR instances at the same time, CS4WAMR implements an instance swapping mechanism and utilizes the *Alloc With Pool* memory mode of WAMR. By copying statically allocated variables, the swapping mechanism changes the current WAMR instance.

As represented in the Figure 4a, the original functioning of WAMR in *Alloc with Pool* memory mode is to have the state and the configuration of WAMR stored in the static variables and the pool buffer pointed by one of the static variables. It corresponds to having a single instance of WAMR.

The Figure 4b is a representation of the memory of our solution presenting the steps of the context switching to swap from instance A to instance B. After initializing the instance, each instance's copy of WAMR variables points to their respective buffer (❶), and the WAMR static variables point to the buffer of the currently loaded instance (❷). The swapping works in two steps. First, the static variables of the current instance are saved (①), then, the values of the static variables of WAMR are replaced by the backup of these variables associated with the new instance to switch to (②). By copying the static variables, the WAMR static variables now point to the buffer of the newly swapped instance, the buffer of instance B (③). As pointers in WAMR buffer pool and pointers managed by the user are instance-specific, they already point to the variables in the correct buffer. This means that other variables, such as dynamically allocated variables, do not have to be updated.

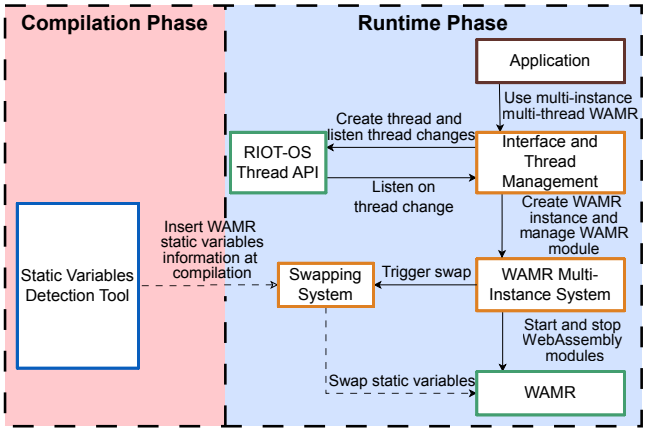


Figure 5: CS4WAMR functional structure

4.2 Framework structure

The framework mechanism is designed to be implemented without modifying the WAMR library and limiting memory overhead to be compatible with resource-constrained microcontrollers.

The CS4WAMR framework is structured in two parts, a tool and a library. The tool is executed at the end of the compilation to automatize the setup of the library by detecting in the device firmware the statically allocated variables used by WAMR and updating the firmware with information on the position of the variables. The library is a wrapper library to use WAMR with multiple instances and to perform the swapping to dynamically change to the appropriate WAMR instance. The library is composed of three components: the swapping component, the WAMR multi-instance component, and the thread and snapshot management component.

The flow of the functional flow of the framework is represented in the Figure 5.

Static variable detection tool. The static variable detection tool is used after compilation to analyze the device firmware to collect the addresses and sizes of WAMR static variables, and then inject this information into a specific variable in the firmware used by the swapping systems.

Swapping System. The swapping system is responsible for copying the WAMR static variables to change the current WAMR instance. The swapping system uses the information provided by the static variable detection tool to know the position of the variables to copy. The swapping system stores the default values of the WAMR static variables to re-initialize WAMR, thereby facilitating the creation of a new WAMR instance.

WAMR Multi-Instance System. The WAMR Multi-Instance System provides an interface to use WAMR with multiple instances while guaranteeing that instances are used correctly. The system works by forcing the user of the system to specify the used instance, verifying that the used modules are loaded in the specified instance, and using the swapping system to switch WAMR to the appropriate instance. This prevents any misusage of the multi-instance system such as running a module in an instance different from which the module has been loaded.

Thread and Snapshot Management System. The thread and snapshot management system uses the WAMR Multi-Instance System and RIOT-OS thread API to run multiple instance of WAMR with one thread by instance. The library also adds container management functionalities such as containers pause and resume, as well as snapshots to restore containers to a previous state. By listening to thread changes, the system automatically swaps to the appropriate instance when the thread scheduler changes the current thread.

5 Evaluation

5.1 Methodology

The evaluation has been done on the release 2025.01 of RIOT-OS. The tests and benchmarks have been run on two boards. The first one is the Arduino nano 33 BLE ABX00030 (rev1) board containing a nRF52840 CPU based on ARM Cortex-M4 and clocked at 64MHz, 256 kilobytes of SRAM, and 1 megabyte of flash. The second one is the nRF9160-DK containing a nRF9160 CPU based on ARM Cortex-M33 and clocked at 64 MHz, with 256 kilobytes of SRAM, and 1 megabyte of flash. All WebAssembly modules have been compiled using clang version 18 with optimization, and without statically linking any library such as libc or WASI related libraries. In order to optimize the modules to the memory constrained of microcontrollers, the size of the Wasm modules has been reduced by using linker flags setting the stack size in linear memory to 1024 and the global base, setting at which address starts the data in the linear memory, to 16. We have also used for all the compilation and linking the optimization flags *O3* and link-time optimization. We have used RIOT-ML 82b1b5c commit version and Apache TVM v0.18.0. Any timing measurements in this paper have been done using the benchmark module from RIOT-OS which uses hardware timer.

5.2 Performances overhead of containerization for TinyML

In this section, we evaluate the performance and memory impact of containerizing TinyML inference code using WebAsembly. We compare direct interpretation of WebAssembly bytecode, execution of WebAssembly code that has been Ahead-of-Time (AoT) compiled to machine code, and native code. To evaluate the impact, we have compared the execution of the inference of the DS-CNN Small INT8 model[4] [36]. To generate the interpreted WebAssembly module and the AoT-compiled WebAssembly module, we have used the proof of concept presented in subsection 3.2. For the native inference code, we have used the RIOT-ML toolkit [18].

Table 1: Model inference time comparison with DS-CNN Small model [36] on Arduino Nano 33 BLE and nRF9160-DK boards

Runtimes	Inference time (in ms)	
	Arduino Nano 33 BLE	nRF9160-DK
WAMR AoT	450.96	396.56
Native execution	488.53	495.79
WAMR Interpreter	46 868.12	48 139.92

Table 2: ROM and RAM usage in bytes of TinyML code, WAMR, bootstrap code for running DS-CNN Small model on Arduino Nano 33 BLE

Runtimes	ROM	RAM
WAMR Interpreter	**89 328**	**138 974**
Wasm module	39 389	39 389
WAMR library	48 849	963
Additional Memory Usage	1 138	98 622
WAMR AoT	**98 716**	**129 705**
Wasm module	61 442	0
WAMR library	35 904	3 083
Additional Memory Usage	1 166	126 622
Native execution	**39 225**	**63 190**

5.2.1 Execution Speed. The means inference time over twenty executions for the three implementations is presented in Table 1. We can see that WebAssembly bytecode interpretation overhead is very high, with an inference time more than 95 times slower than native execution. Surprisingly, AoT compilation gives better performance than native execution. Similar phenomena have been observed by Wen and Weber [33], which implements its own WebAssembly AoT compiler and OS kernel layer to run WebAssembly. Some possible reasons for performance improvement are the optimizations of the WebAssembly code or the optimization of the WAMR Ahead-of-Time compiler which is configured for the specific CPU. We have tested if this was caused by the compatibility code used in our WebAssembly containers to replace unavailable mathematical functions by using the reimplemented functions in the native code. We have seen that this was not the source of better performance as it only caused some performance overhead to the native execution.

In terms of execution speed, due to the major execution overhead, WebAssembly interpretation with WAMR is not usable for TinyML in practice, but WAMR Ahead-Of-Time compilation seems to be totally usable, as it has given us native performances.

[4]Available at https://github.com/ARM-software/ML-zoo/tree/master/models/keyword_spotting/ds_cnn_small/model_package_tf/model_archive/TFLite/tflite_int8

Table 3: ROM and RAM usage of WAMR multi-instance and WAMR on a minimal runnable WebAssembly modules. WAMR has been compiled in AoT mode

	For 1 container		Per additional container	
Application	ROM	RAM	ROM	RAM
WAMR-based	**36424**	**5925**	**+600**	**+2484**
WAMR Library	35756	3083	+0	+0
WAMR Buffer Pool	0	2710	+0	+2480
Additional Memory	668	132	+600	+4
CS4WAMR-based	**37848**	**8099**	**+640**	**+3984**
WAMR Library	35770	3083	+0	+0
CS4WAMR Library	1406	1032	+0	+0
Buffer Pools	0	3980	+0	+3980
Additional Memory	672	4	+640	+4

The cost per additional container for the WAMR-based section corresponds to the cost to add a container to WAMR, while the cost for the CS4WAMR-based section is the cost to create a new WAMR instance and add a new container to it using CS4WAMR.

5.2.2 Memory Overhead. The size occupied in memory by the TinyML code, the WebAssembly Micro Runtime (WAMR) and the bootstrap code to launch WAMR and the TinyML code is presented in Table 2. From the memory usage measure, we can see that the usage of WebAssembly for TinyML requires two to three times more ROM and more than two times more RAM than with the native execution.

While Ahead-of-Time (AoT) compilation uses slightly less RAM than bytecode interpretation, it consumes more ROM. This is notably due to AoT compilation creating larger containers, leading to increased ROM consumption. However, these containers can be accessed directly from flash using execute in place (XIP) mode, which allows execution without using RAM to store containers. Also, AoT consumes more RAM for running, thus requires a larger pool buffer, present in the table in the additional memory usage, but this is compensated by XIP mode.

The memory usage overhead of using WebAssembly is substantial. While the overhead still allows to run the smaller model, it is a major obstacle for running slightly larger models with WebAssembly on microcontrollers.

5.3 Performances overhead of the CS4WAMR framework

In this section, we evaluate the performance and memory impact of the CS4WAMR framework. We first compare the memory impact by comparing the overhead when running tiny WebAssembly modules. Then, we measure the cost of the swapping mechanism and the instance snapshot mechanism.

Memory Overhead. The flash and RAM memory usage for executing the minimal AoT-compiled WebAssembly containers using WAMR and CS4WAMR is shown in Table 3. In our evaluation with minimal containers, CS4WAMR has a memory overhead of less than 2 kilobytes of ROM, and an overhead of more than 2 kilobytes of RAM.

Swap time cost. The main cause of execution overhead with CS4WAMR is swapping to another WAMR instance. This can occur when calling CS4WAMR functions performing operations on a specific WAMR instance or when the thread scheduler changes the current thread for a thread in which WAMR runs. We have measured the swapping time by alternatively swapping between three instances. The average swapping time over 2000 swaps is 6.9us on Arduino Nano 33 BLE and 6.2us on nRF9160-DK with some highest swapping time without memory caching of 9us on both devices. When using a preemptive scheduler, the impact of swapping on the execution depends on the timeout for thread context switching, as it will influence the number of switching per second.

Snapshot mechanism performances. The snapshot mechanism stores the buffer pool of the instance to save, so it requires the same size to store. Although with the minimal container, the size to store is only a bit less than eight kilobytes, it can cost more than a few hundred kilobytes for TinyML containers. The performance overhead to snapshot a ten kilobytes instance buffer pool is 159us on the Arduino Nano 33 BLE and 100us on the nRF9160dk. The same time is required to restore the instance. The overhead is mainly provoked by the copy of the buffer pool and by the thread pause and resume.

6 Discussion

6.1 Benefits and challenges of containerization of TinyML

6.1.1 Benefits of containerization of TinyML. By leveraging TinyML containerization, entities deploying their services on microcontrollers gain autonomy in selecting the models they use and are not confined to pre-existing service on the device. Furthermore, containerization lets device owners safely authorize ML providers to deploy models on a device without requirering the device owner to completely trust the TinyML providers. In addition, the isolation and permissions access control provided by containerization for both applications and TinyML service limit the impact of attacks by bad actors and enhance the confidentiality of model weights by limiting the number of actors who can access them.

6.1.2 Challenges of WebAssembly for TinyML.

Constrained of WebAssembly linear memory with TinyML. WAMR loads the entire linear memory of WebAssembly modules into the RAM to execute them. As the weights of the model are stored inside the WebAssembly linear memory, in our proposition, the entire model is loaded in RAM as soon as the module is instantiated, even if the model is not currently in use. This creates an issue with larger models that cannot be entirely loaded to RAM, making their usage from containers impossible with our technique. One possible solution would be to use weights directly from WebAssembly bytecode in flash. This would require the support for multiple memories in one module and the possibility to define constant linear memories in WebAssembly. The former is currently a proposal for WebAssembly[5] and is starting to be supported by major runtimes, but is not yet supported by the majority of compilation toolchains.

[5] https://github.com/WebAssembly/multi-memory/

If you cite this paper, please use the EWSN 2025 reference: B. Buil, C. Gaber, E. Baccelli, S. Bouzefrane. *TinyML as a Service on Multi-Tenant Microcontrollers*. Proceedings of the 22nd International Conference on Embedded Wireless Systems and Networks (EWSN 2025).

Integration with underlying hardware. Currently our proposition cannot use special hardware, such as an AI chip, to accelerate machine learning. This is due to the isolation of WebAssembly and the absence of a secure interface to interact with the peripheral. Some WebAssembly runtimes, such as Aerogel [23], provide secure native functions for accessing microcontroller peripherals. Another solution could be an extension of the Webassembly System Interface (WASI) [17], which defines interfaces to interact with the host system, to support microcontroller peripherals.

6.2 Benefits and limitation of containerization of CS4WAMR

6.2.1 Benefits of the swapping mechanism. The CS4WAMR framework improves WAMR in different aspects.

Memory exhaustion protection. Memory for each WAMR instance is now allocated in separate buffers managed by the WAMR buffer allocator. Having each WAMR instance its own allocation buffer prevents a module from impacting modules of other instances, as the exhaustion of memory by one module would only impact the buffer of its own instance. This allows pre-allocating dedicated space for TinyML container.

Snapshot mechanism. The snapshot mechanism allows resetting environments to a state where the environment is already initialized and is known to be in a non-corrupted state. This allows to use the mechanism against any attacks hiding malware in the container's memory. With TinyMLaaS, this allows to reinforce TinyML service stability and security, especially when interacting with multiple containers.

Granular permissions through multiple instances of WAMR and instance-wide configuration. The context switching mechanism allows for the creation of multiple WAMR instances. By assigning one WAMR instance per module, each instance can be configured to the specific permissions of its corresponding module, enabling a permission system which checks modules permissions during the loading of the module when linking the native functions. Thereby, CS4WAMR allows to reinforce the granularity of permission management in WAMR by having per-container permissions instead of permissions shared by all the containers of the device.

A first step for hardware isolation between Wasm module. The CS4WAMR framework allocates distinct memory spaces for each WAMR instance, guaranteeing separated and clearly delimited memory regions. Assigning one module per WAMR instance allows to establish distinct memory areas for each Wasm module. This segmentation of containers' memory regions is a preliminary step for achieving MPU isolation between modules. With CS4WAMR, hardware-level isolation between modules is now a low-hanging fruit using tools like PIP-MPU [13].

6.2.2 Limitations of the swapping mechanism. We have identified multiple limits of the configuration-switching mechanism notably due to inherent issues in WAMR.

Runtime overhead of the swapping mechanism. After loading the container, the only performance overhead is the swapping between WAMR instance. Thus, since the microcontrollers studied are single core, the performance overhead depends on the number of switches between WAMR instances. The number of switches depends on the thread scheduler of the OS and its configuration. For example, for a round-robin scheduler using a timeout system, the overhead will depend on the timeout value: a shorter timeout yields more switches and, thus, more overhead. Thus, the total overhead of CS4WAMR is mainly dependent on the underlying system.

Inter-module communication. When using the WAMR's built-in inter-module communication through registered modules, shared modules must share the same WAMR instance as the calling modules. In addition, when a module is registered, it is shared with all modules in the instance. This might be an issue in the situation where a module wants to communicate with two modules, as the two modules would have access to each other. Also, since the modules shared the same WAMR instance, it means that all the modules would have access to the same native functions. Moreover, to share buffers between modules using the built-in inter-module communication in WAMR, modules must either use function arguments to share few bytes at a time or share their memory with the other module. The first option is slow and ineffective, as it requires a lot of calls to native functions. The second option is difficult to implement as C toolchains do not support multiple linear memory, thus modules should share the same memory while not conflicting each other. That is why we think it is better to have one WAMR instance per module and an inter-container communication system, similarly to the one used in our proof of concept to communicate with our TinyML container or like the one used by Akkermans et al. [6].

Fix memory position of WAMR instances. One limitation of the switching mechanism is that WAMR uses pointers with absolute memory addresses, making WAMR instance position-dependent in memory. Thus, an instance should be restored at the same location from where it was saved. This makes difficult the use of the snapshot mechanism for dynamic-linking as it requires guaranteeing that the location from where an instance has been saved and unloaded will be free when the instance needs to be reloaded. This limits the opportunity to use the snapshot system for dynamically loading ML models when needed, like by saving to flash and unloading the WAMR instance of the model and reloading the instance when needed. However, if multiple ML models should be loaded and used, but only one model should be loaded at the same time, the mechanism can still be used by dedicating a location for ML models and switching from flash and to flash the currently loaded model. Moreover, doing inter-device instance migration with the snapshot mechanism requires that instances are loaded at the same address in the memory of the two devices, and the buffer available for the instance should have the same size. Having the same firmware on the two devices is one solution. Another better solution for fix memory mapping of instances would be to define the buffer in a user-defined section, and use a link script to link the section at a fix address. This would guarantee that the section and its variables are always loaded at the same place under the condition that instance buffers have the same size on the different devices.

Snapshot and OS timers. The snapshot mechanism affects container execution by causing a side effect. That is, during the loading

or saving of a container, any thread that was paused due to the sleep function of the timer library of the OS will bypass the sleep and continue running immediately. Potential issues may arise if the container uses a waiting mechanism, such as when a thread sleeps and should be awakened by an event.

6.3 Generalization

Generalization of the swapping mechanism. The swapping mechanism can be applied to other libraries. We have identified two requirements to apply this technique. The first requirement is that the library should store its state in some statically defined variable. The second requirement is that the library must not use any dynamic allocations, or the dynamic allocation should be controllable. For example, with WAMR, it is possible to do allocations from a given buffer. The swapping mechanism can directly be applied on any devices as long as they are single-core, only the multi-thread and snapshot integration is dependent on RIOT-OS. The swapping mechanism cannot be applied to systems with multiple cores, as different WAMR instances would run simultaneously on different cores, but only one WAMR instance can be loaded at the same time.

Portability to other microcontrollers/boards. By design, CS4WAMR offers portability on various devices, as we have built it on top of a set of tools and platforms that provide out-of-the-box portability to a wide range of similar systems based on various 32-bit microcontrollers (Arm Cortex-M, ESP32, RISC-V). Namely: we use RIOT-OS for microcontroller hardware abstraction and peripheral abstraction. We use uTVM (as integrated in RIOT-ML) to allow arbitrary neural network model compilation on various microcontrollers. Last but not least, we used WebAssembly Micro Runtime which offers a similar level of portability for our container runtime, across a wide variety of hardware.

7 Related Work

7.1 Multi-tenancy on microcontrollers

Multi-tenancy is possible on microcontrollers through a compartmentalization system providing sandbox isolation. Compartmentalization systems allow setting policies to separate programs into two or more protection domains, called compartments, and enforcing isolation policies at runtime Lefeuvre et al. [21]. Sandbox isolation is a type of compartmentalization that consists of restricting the privileges of an application to protect the rest of the system and other applications.

One common type of sandbox on microcontrollers is runtimes. VeloxVM [31], Polyglot Cerberos [6] and FemtoContainers [35] propose runtimes that interpret bytecode for the secure execution of third-party code on multi-tenant microcontrollers and provide a permission system to restrict access to host resources.

One bytecode used by numerous runtimes on microcontrollers is WebAssembly [1, 5, 23, 28]. While some runtimes like WAMR [1] only allows setting global permission to host functions for all containers of the runtimes, some runtimes like Aerogel [23] integrate a fine-grained access control system to perform runtime checks based on permission and energy consumption to control access to sensors and actuators without having to pre-inspect the application. WebAssembly is easily associable with containers as it is based on Wasm modules which are runnable units of software, and runtimes like Wasmico [28] propose to manage WebAssembly module as containers with features like pause and resume, memory consumption control.

Table 4: Feature comparison between runtimes from state of the art and CS4WAMR

Runtimes	Bytecode	Permission management per container	AoT compilation	Memory consumption limitation	Container snapshot
Femto-Containers [35]	ebpf-based	✓	✗	✓	✗
VeloxVM [31]	Custom	✓	✗	✓	✗
Polyglot Cerberos [6]	Custom	✓	✗	✓	✗
Aerogel [23]	Wasm	✓	✗	✓	✗
WAMR [1]	Wasm	✗	✓	✗	✗
Wasmico [28]	Wasm	✓	✗	✓	✗
CS4WAMR	Wasm	✓	✓	✓	✓

Another way to execute WebAssembly is through compilation of WebAssembly to machine code, called Ahead-of-Time (AoT) compilation. Multiples solutions allow WebAssembly AoT compilation [1, 9, 33] with different features like formally verified compiler [9] and OS for kernel call compatibility [33].

We compare CS4WAMR with some runtimes from state of the art in Table 4.

7.2 Machine learning on microcontrollers

TinyMLOps. TinyML consists of running machine learning on microcontrollers. TinyML faces multiple challenges such as energy efficiency, security and privacy, latency, limited memory, model update [16]. TinyMLOps [20], like MLOps, focuses on streamlining the deployment of production ML models using optimal tools and practices, from data acquisition to model inference on an embedded device. TinyMLOps can be seen as a pipeline for the production and execution of TinyML. Oliveira et al. [26] describe TinyMLOps as seven phases such as ETL (Extraction, Transformation, and Loading) to create a data set, model training, model validation, model optimization for constrained devices, model conversion in runnable code, sending the model over-the-air, and inference from the device. On low-end microcontrollers, the framework most used for the implementation of TinyML models is TensorFlow Lite Micro (TFLM) [12], but other frameworks exist, such as Open Neural Network Exchange (ONNX) [2] and Pytorch [3]. To perform model inference, the previous frameworks propose their ML engine, another alternative is model compilers like Apache TVM [11] or uTensor [4], which transforms models into machine code. Another alternative is Edge Impulse [8] which is a TinyMLOps platform that performs training, optimization, and prepares the code for deployment on the target device. For the last steps of the TinyMLOps pipeline, RIOT-ML [18] is an ML framework that manages model compilation, model deployment and updates, and model evaluation.

TinyMLaaS. Doyu et al. [14] defines TinyMLaaS (TinyML as a Service) as a Cloud or Edge platform that simplifies ML model deployment by managing their compilation and offering code to

use the model with a hardware and software-agnostic high-level interface to use the model. Zaidi et al. [34] defines TinyMLaaS (TM-LaaS) differently as the exposure of machine learning services on gateways and end devices that can be consumed by other services. The on-device model is for quick-and-coarse inference, while the gateway is for fine grained inference.

8 Conclusion

We have presented a new infrastructure for Tiny Machine Learning as a Service (TinyMLaaS) on multi-tenant microcontrollers based on TinyML containers compiled at the edge or the cloud. We have implemented a proof of concept of this architecture through the use of RIOT-ML [18] model compiler generating efficient code to run machine learning models, and WAMR Ahead-of-Time (AOT) compilation to optimize the execution of WebAssembly containers and permit in-place execution (XIP).

Through the isolation and easy deployment of WebAssembly modules, this architecture allows untrusted third parties to provide TinyML services, running on the microcontrollers with near-native performance, for the applications running on the device. However, WebAssembly for TinyML on microcontrollers still has major limitations, such as the need to load model weights in RAM as they are located in WebAssembly linear memory which cause massive RAM overhead. We have also identified some lacks of features in the WebAssembly Micro Runtime that would be desirable for the TinyMLaaS use case. The lacking features are module-wide permission to respect the least-privilege principle, controls of module memory consumption, hardware isolation between containers, and snapshot mechanism to manage container corruption.

As a first step against the WAMR limitations, we have proposed CS4WAMR which is a framework to extend capabilities of WAMR by enabling multiple instances of WAMR running simultaneously on single core systems. This framework enables container snapshots, module permissions, and guarantee a segmentation of the memory use by the modules.

Some forthcoming challenges for TinyMLaaS on multi-tenant microcontrollers involve the implementation of a constant WebAssembly linear memory in flash to store model weights and the implementation of Memory Protection Unit (MPU) isolation to guarantee confidentiality of model weights.

Acknowledgments

The research leading to these results is funded by ANRT Convention Cifre n°2024/0426. This paper reflects only the authors' views.

References

[1] [n. d.]. Bytecodealliance/Wasm-Micro-Runtime: WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime.
[2] [n. d.]. Open Neural Network Exchange (ONNX). https://onnx.ai/.
[3] [n. d.]. PyTorch. https://pytorch.org/.
[4] [n. d.]. uTensor: TinyML AI Inference Library. https://github.com/uTensor/uTensor.
[5] [n. d.]. Wasm3/Wasm3. Wasm3 Labs.
[6] Sven Akkermans, Bruno Crispo, Wouter Joosen, and Danny Hughes. 2018. Polyglot CerberOS: Resource Security, Interoperability and Multi-Tenancy for IoT Services on a Multilingual Platform. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ACM, New York NY USA, 59–68. https://doi.org/10.1145/3286978.3286997
[7] Atym. [n. d.]. Atym. https://www.atym.io.
[8] Colby Banbury, Vijay Janapa Reddi, Alexander Elium, Shawn Hymel, David Tischler, Daniel Situnayake, Carl Ward, Louis Moreau, Jenny Plunkett, Matthew Kelcey, Mathijs Baaijens, Alessandro Grande, Dmitry Maslov, Arthur Beavis, Jan Jongboom, and Jessica Quaye. 2023. Edge Impulse: An MLOps Platform for Tiny Machine Learning. *Proceedings of Machine Learning and Systems* 5 (March 2023), 254–268.
[9] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing Using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*. 1975–1992.
[10] Guillaume Bouffard and Léo Gaspard. 2018. Hardening a Java Card Virtual Machine Implementation with the MPU. In *Symposium Sur La Sécurité Des Technologies de l'information et Des Communications (SSTIC)*. Rennes, France.
[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 579–594.
[12] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. *Proceedings of Machine Learning and Systems* 3 (March 2021), 800–811.
[13] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. 2023. Pip-MPU: Formal Verification of an MPU-Based Separationkernel for Constrained Devices. *International Journal of Embedded Systems and Applications* 13, 02 (June 2023), 1–21. https://doi.org/10.5121/ijesa.2023.13201
[14] Hiroshi Doyu, Roberto Morabito, and Martina Brachmann. 2021. A TinyMLaaS Ecosystem for Machine Learning in IoT: Overview and Research Challenges. In *2021 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 1–5. https://doi.org/10.1109/VLSI-DAT52063.2021.9427352
[15] Dr. Lachit Dutta and Swapna Bharali. 2021. TinyML Meets IoT: A Comprehensive Survey. *Internet of Things* 16 (Dec. 2021), 100461. https://doi.org/10.1016/j.iot.2021.100461
[16] Thommas K. S. Flores, Ivanovitch Silva, Mariana B. Azevedo, Thaís de A. de Medeiros, Morsinaldo de A. Medeiros, Daniel G. Costa, Paolo Ferrari, and Emiliano Sisinni. 2025. Advancing Tiny Machine Learning Operations: Robust Model Updates in the Internet of Intelligent Vehicles. *IEEE Micro* 45, 1 (Jan. 2025), 76–86. https://doi.org/10.1109/MM.2024.3354323
[17] Dan Gohman, Lin Clark, Alex Crichton, Andrew Brown, Sam Clegg, Pat Hickey, Yosh, Dave Bakker, Mendy Berger, Colin Ihrig, Peter Huene, Piotr Sikora, Jakub Konka, Bailey Hayes, Chris Dickinson, Mike Frysinger, Robin Brown, YAMAMOTO Takashi, Syrus Akbary, Sergey Rubanov, Josh Triplett, George Kulakowski, Eric Crosson, Denis Vasilik, Christian Clauss, Mark Christian, MaulingMonkey, Merlijn Sebrechts, Michiel Van Kenhove, and Nathaniel McCallum. 2025. WebAssembly/WASI: V0.2.4. Zenodo. https://doi.org/10.5281/ZENODO.14826680
[18] Zhaolan Huang, Koen Zandberg, Kaspar Schleiser, and Emmanuel Baccelli. 2024. RIOT-ML: Toolkit for over-the-Air Secure Updates and Performance Evaluation of TinyML Models. *Annals of Telecommunications* (May 2024). https://doi.org/10.1007/s12243-024-01041-5
[19] Riku Immonen and Timo Hämäläinen. 2022. Tiny Machine Learning for Resource-Constrained Microcontrollers. *Journal of Sensors* 2022, 1 (2022), 7437023. https://doi.org/10.1155/2022/7437023
[20] Minh Tri Lê and Julyan Arbel. 2023. TinyMLOps for Real-Time Ultra-Low Power MCUs Applied to Frame-Based Event Classification. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*. ACM, Rome Italy, 148–153. https://doi.org/10.1145/3578356.3592586
[21] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2024. SoK: Software Compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 75–75. https://doi.org/10.1109/SP61157.2025.00075
[22] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old Is New Again: Binary Security of {WebAssembly}. In *29th USENIX Security Symposium (USENIX Security 20)*. 217–234.
[23] Renju Liu, Luis Garcia, and Mani Srivastava. 2021. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. 94–105. https://doi.org/10.1145/3453142.3491282
[24] microEJ. [n. d.]. MicroEJ Embrace IoT-Enabled Servitization. https://www.microej.com.
[25] Irene Niyonambaza Mihigo, Marco Zennaro, Alfred Uwitonze, James Rwigema, and Marcelo Rovai. 2022. On-Device IoT-Based Predictive Maintenance Analytics Model: Comparing TinyLSTM and TinyModel from Edge Impulse. *Sensors* 22, 14 (Jan. 2022), 5174. https://doi.org/10.3390/s22145174
[26] Franklin Oliveira, Daniel G. Costa, Flávio Assis, and Ivanovitch Silva. 2024. Internet of Intelligent Things: A Convergence of Embedded Systems, Edge Computing and Machine Learning. *Internet of Things* 26 (July 2024), 101153.

https://doi.org/10.1016/j.iot.2024.101153

[27] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. SoK: Security and Privacy in Machine Learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 399–414. https://doi.org/10.1109/EuroSP.2018.00035

[28] Eduardo Ribeiro, André Restivo, Hugo Sereno Ferreira, and João Pedro Dias. 2024. WASMICO: Micro-containers in Microcontrollers with WebAssembly. *Journal of Systems and Software* 214 (Aug. 2024), 112081. https://doi.org/10.1016/j.jss.2024.112081

[29] J.H. Saltzer and M.D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (Sept. 1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[30] Nikolaos Schizas, Aristeidis Karras, Christos Karras, and Spyros Sioutas. 2022. TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review. *Future Internet* 14, 12 (Dec. 2022), 363. https://doi.org/10.3390/fi14120363

[31] Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A Safe Execution Environment for Resource-Constrained IoT Applications. *Journal of Network and Computer Applications* 118 (Sept. 2018), 61–73. https://doi.org/10.1016/j.jnca.2018.06.001

[32] Ashton Webster, Ryan Eckenrod, and James Purtilo. 2018. Fast and Service-preserving Recovery from Malware Infections Using {CRIU}. In *27th USENIX Security Symposium (USENIX Security 18)*. 1199–1211.

[33] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 1–4. https://doi.org/10.1109/PerComWorkshops48775.2020.9156135

[34] Syed Ali Raza Zaidi, Ali M. Hayajneh, Maryam Hafeez, and Q. Z. Ahmed. 2022. Unlocking Edge Intelligence Through Tiny Machine Learning (TinyML). *IEEE Access* 10 (2022), 100867–100877. https://doi.org/10.1109/ACCESS.2022.3207200

[35] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2022. Femto-Containers: Lightweight Virtualization and Fault Isolation for Small Software Functions on Low-Power IoT Microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference (Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 161–173. https://doi.org/10.1145/3528535.3565242

[36] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2018. Hello Edge: Keyword Spotting on Microcontrollers. https://doi.org/10.48550/arXiv.1711.07128 arXiv:1711.07128 [cs]