RIOT-ML: Toolkit for Over-the-Air Secure Updates and Performance Evaluation of TinyML Models

Zhaolan Huang, Koen Zandberg, Kaspar Schleiser, Emmanuel Baccelli

If you cite this paper, please use the following reference: Z. Huang, K. Zandberg, K. Schleiser and E. Baccelli. RIOT-ML: Toolkit for Over-the-Air Secure Updates and Performance Evaluation of TinyML Models. Annals of Telecommunications, 2024 (to appear).

RIOT-ML: Toolkit for Over-the-Air Secure Updates and Performance Evaluation of TinyML Models

Zhaolan Huang[1*], Koen Zandberg[1], Kaspar Schleiser[1], Emmanuel Baccelli[1, 2]

[1*]Freie Universität Berlin, Germany.
[2]Inria Saclay, France.

*Corresponding author(s). E-mail(s): zhaolan.huang@fu-berlin.de;

Abstract

Practitioners in the field of TinyML lack so far a comprehensive, "batteries-included" toolkit to streamline continuous integration, continuous deployment and performance assessments of executing diverse machine learning models on various low-power IoT hardware. Addressing this gap, our paper introduces RIOT-ML, a versatile toolkit crafted to assist IoT designers and researchers in these tasks. To this end, we designed RIOT-ML based on an integration of an array of functionalities from a low-power embedded OS, a universal model transpiler and compiler, a toolkit for TinyML performance measurement, and a low-power over-the-air secure update framework – all of which usable on an open-access IoT testbed available to the community. Our open source implementation of RIOT-ML and the initial experiments we report on showcase its utility in experimentally evaluating TinyML model performance across fleets of low-power IoT boards under test in the field, featuring a wide spectrum of heterogeneous microcontroller architectures and fleet network connectivity configurations. The existence of an open source toolkit such as RIOT-ML is essential to expedite research combining Artificial Intelligence and IoT, and to foster the full realization of edge computing's potential.

Keywords: AI, IoT, Machine Learning, Low-power, Microcontroller, Benchmarks, Software Update, MLOps

1 Introduction

As Artificial Intelligence (AI) permeates our lives more and more, mechanisms such as deep neural networks [31] are put to use (or their deployment planned) in more and more places in various distributed systems. In particular, wireless sensor network (WSN) can improve its coverage and connectivity, reduce energy and bandwidth usage by deploying AI onto edge nodes [28].

The data pipeline with AI typically requires the creation and the use of a *model*, i.e. a layered structure of complex algorithms (also known as *operators*) which interpret data and make decisions based on that data. This model must first be trained (*learning* phase [31]), before it can be put in production (used for *inference*).

Recent work from the Tiny Machine Learning (TinyML) [23, 25] community forays into optimizing models to fit tinier resource budgets (and to perform efficiently nevertheless) on low-power microcontrollers in the Internet of Things (IoT). As a consequence, both learning and inference placement possibilities are extended to encompass ultra low-power terminals.

However, generic and convenient open source tools lack designers tackling a combination of AI and IoT (AIoT), who are required to:

- evaluate the performance of their models when placed somewhere along the terminal-edge-cloud continuum, especially when including potential placement on different microcontroller-based devices;
- fine-tune their models, and identify performance bottlenecks at model layer granularity, on different microcontrollers;
- select an adequate microcontroller to execute their model, for a targeted task running on a low-power device to-be-designed;
- continuously and securely update pre-provisioned models on fleets of heterogeneous devices-under-test, remotely, over the network;
- monitor the computational performance of deployed models remotely, over the network, e.g. on new data collected in the field.

This paper thus introduces the *RIOT Toolkit for Machine Learning* (RIOT-ML), an open source AIoT toolkit that tightly combines a generic model compiler and a popular low-power IoT operating system. RIOT-ML implements a workflow to automatically compress, flash and evaluate arbitrary models[1] on arbitrary commercial off-the-shelf (COTS) low-power boards[2] based on different popular microcontroller (MCU) architectures. By leveraging a widely applicable low-power network stack combined with secure IoT software mechanisms, RIOT-ML also provides the ability to control, monitor and update ML model on fleets of such devices remotely, over heterogeneous network topologies.

Paper contributions. Our contributions are as follows:

- We designed a universal toolkit for on-board evaluation and remote monitoring of TinyML models (RIOT-ML) on low-power devices. RIOT-ML integrates and extends the toolkit U-TOE [13], providing feasibility checks for the use of arbitrary models in a wide selection of IoT hardware platforms. It allows researchers and developers to locate the performance bottleneck of a given model on a target device. The evaluation results enable co-design with other components at system level, help optimize ML models and configurations for specific use cases, allowing to achieve the best possible performance on target devices.
- We design, implement and evaluate mechanisms for secure over-the-air (OTA) model updates, as well as for continuous deployment and management of arbitrary models on resource-constrained devices over arbitrary network configurations, which may also include low-power network links.
- We released the code[3] of RIOT-ML under an open source licence. This implementation enables compilation, flashing, evaluation, and secure OTA updates of various neural networks (computational graph-based models) from mainstream ML frameworks onto various low-power boards based on popular instruction set architectures (ARM Cortex-M, ESP-32, RISC-V).
- We provide benchmarks and a comparative experimental evaluation using RIOT-ML, reproducible both on an open-access IoT testbed and on personal workstations, which provide insights on inference performance with different models on different low-power hardware and demonstrate how RIOT-ML can be re-used by TinyML experimental researchers and developers to fine-tune IoT configurations.

2 Related work

Recent work has surveyed [24] the scope of ML frameworks, tasks, metrics, including a comprehensive review of TinyML stack and deployment pipeline. A number of challenges need to be met in order to fit the tiny resource budgets typical of microcontrollers (kiloBytes of memory, power consumption in milliWatt, central processing unit (CPU) frequency in MegaHertz) while maintaining performance at an acceptable level and retaining portability to extremely polymorphic hardware in this category.

Embedded IoT Software Platforms – Various open source IoT operating systems are used to provide hardware abstraction, and resource-sharing primitives as well as convenient peripheral access (e.g. sensor/actuator, network subsystem) on heterogeneous low-power microcontrollers. Prior work such as [11] surveys such operating systems, among which prominent examples

[1] output of TensorFlow, PyTorch...
[2] such as BBC:microbit, nrf52840dk, Arduino Zero, HiFive...
[3] see https://github.com/TinyPART/RIOT-ML

include RIOT [2] and Zephyr[4]. So far however, such software platforms offer no support for ML framework – or if they do, this support is very limited. Moreover, the most advanced supports so far are typically hardware- or vendor-specific e.g. with libraries provided by STM32CubeMx or ARM CMSIS-NN.

Low-power IoT Testbeds – Various testbeds offer remote access to fleets of reprogrammable microcontroller-based devices. Prior work such as [10, 18] survey such testbeds, among which prominent examples include the open access facility IoT-lab [1], which offers remote bare-metal access (serial over Transmission Control Protocol (TCP)) to a fleet composed of hundreds of popular low-power boards of various kind.

Benchmarking Suites for TinyML – Benchmarking ML on low-power hardware entails a number of challenges [4]. Prior work such as MLPerf Tiny [3] provides a standard benchmark suite (a fixed set of representative ML tasks) for evaluating the performance of given hardware, and an online platform for manufacturers to publish their comparative benchmark results. In contrast, RIOT-ML offers a more powerful and more customizable toolkit for performing feasibility checks of user-defined machine learning models on low-power devices, with a greater degree of flexibility and customization.

TinyML Benchmarks – Prior work such as [21] focuses on performance comparison of different machine learning frameworks on two COTS low-power boards (Arduino Nano BLE 33 and STM32 NUCLEO-F401RE). In particular, it benchmarked two TinyML frameworks, Tensorflow Lite for Microcontrollers (TFLM) and X-CUBE-AI over gesture recognition and wake word spotting. Other work such as [30] tested TFLM models on several microcontroller-based boards. While such papers provide a performance comparison of specific frameworks on specific boards for specific tasks, RIOT-ML offers greater flexibility and generality, allowing developers to evaluate a wider range of (user-specified) models on a larger variety of low-power devices, and to dive into the execution details of ML models.

TinyML Model Transpiler & Compilers – Compilers such as Tensor Virtual Machine (TVM) [8] can be used to automate the transpilation and compilation of models provided by major ML frameworks (TFML, Pytorch, etc.) so as to expose low-level routines and optimize them for execution on specific processing unit characteristics (CPU, graphics processing unit (GPU) etc.). An extension of TVM called uTVM was recently introduced, adding smaller hardware targets including a variety of MCUs .

TinyML Model Profilers – ML-EXray [22] enables TinyML developers to gain visibility into the layer-level details of ML execution and diagnose cloud-to-edge deployment issues. Developers can analyze and debug edge deployment pipelines with high usability, using less than 15 lines of code for a full examination. However, the reliance on Tensorflow Lite restricts the capability to accommodate models from further ML frameworks and deploy them on low-power devices. Major ML frameworks (TFML, Pytorch and MXNet, etc.) provide internal profiler [33]. Such tools allow developers to measure the performance of their models. They can be used to collect metrics such as inference time and memory usage, which can then be analyzed to optimize the model's performance. Though it can provide us execution details at layer level, it still lacks the support for on-device deployment and evaluation on various IoT devices, while RIOT-ML is a more general-purpose toolkit that provides a comprehensive solution on a wide range of low-power devices.

Tiny Machine Learning Operations (TinyMLOps) – MLOps is a paradigm adapted from software engineering (Development Operations (DevOps)) and aims to deliver and maintain ML models efficiently and reliably in the field. It requires continuous end-to-end deployment of model and monitoring of its performance. TinyMLOps extends MLOps to support continuous integration and continuous deployment (CI/CD) workflows of ML model on resource-constrained devices. Several frameworks [9, 16, 17] provide MLOps components for conventional servers or large-scale clusters, but typically do not encompass TinyML devices due to the unique challenges these incur, in particular: extreme polymorphism in hardware architectures and network stacks. As a result, TinyMLOps (and to a large extent MLOps itself) is a paradigm that is still in its infancy.

[4] see https://www.zephyrproject.org/

Table 1 Comparison of AIoT frameworks and toolkits.

Framework	MCU	Model Type	Remote Eval./Up.	Granu.	Model to Board Sol.
MLPerf	Yes	Specified	No	Model	No
ML-EXray	No	TFLite	No	Layer	No
TFLite	Yes	TFLite	No	Layer	No
Pytorch	No	Torch	No	Layer	No
uTVM	Yes	Universal	No	Operator	No
U-TOE	Yes	Universal	No	Operator	Yes
RIOT-ML	Yes	Universal	Yes	Operator	Yes

Eval. – Evaluation, Sol. – Solution, Up. – Update, Granu. – Granularity

Software Updates for Low-Power IoT – Software updates in the context of IoT are crucial for resolving security vulnerabilities, extending functionalities, and improving performance of low-power devices during their lifetime. Research studies such as [5, 20, 34] focused on secure update mechanisms, including covering techniques for authentication, integrity checks and encryption, aiming to mitigate attacks coming from potential adversaries with various levels of computing power, up to quantum computing power levels. Efficiency and reliability aspects are explored, along with strategies for network load minimization and power management, aiming to optimize the OTA update process in IoT environments with limited network and battery resources [6]. Standardization efforts at the Internet Engineering Task Force (IETF), such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) and Constrained Application Protocol (CoAP) [7, 29] have shrunk on-board memory footprint and network transfer costs of Internet Protocol version 6 (IPv6) and Hypertext Transfer Protocol (HTTP)-like interaction over the network, while more recent standards such as Software Updates for Internet of Things (SUIT) [19] aim further at specifying generic architectures, data models and metadata for low-power secure IoT software updates over transfer protocols such as CoAP.

The above is summarized in Table 1.

3 Background on TinyML Performance Analysis

On the one hand, as the most immediate limiting resource budget on microcontrollers concerns memory limitations, typically in the order of kiloBytes, TinyML performance evaluation typically focuses primarily on metrics measuring memory consumption – while keeping an eye on execution speed – as described below. On the other hand, TinyML performance analysis can be tackled at different granularity levels: at the global model level, or at the operator level, for finer granularity, as described in the following.

3.1 Performance Metrics

The considered metrics offer insights into the feasibility, efficiency and resource utilization of offloading model inference burden to low-power devices. By analyzing these metrics, users can make initial decisions regarding model selection, optimization techniques, and hardware configurations to maximize performance and minimize the resource footprint on low-power devices.

Memory (RAM) Consumption – This metric measures the amount of dynamic memory space (primary random access memory (RAM)) consumed by the model during inference. It reflects the memory footprint of the model activation and is important for low-power devices that have limited memory resources. Efficient memory utilization allows for the deployment of larger and more complex models on such devices.

Storage (Flash memory) Consumption – This metric quantifies the amount of storage space, typically in terms of Flash memory region, required to store the compute instruction and associated parameters. It reflects the model's storage footprint on the low-power device. Minimizing storage consumption allows for accommodating multiple models on the device or orchestrating with other essential applications.

Computational Latency – This metric measures the time consumption of performing inference for each input sample, either at the model level or at the level of individual operators within the model. It reflects the inference speed of the model on the low-power device and plays a crucial role in real-time or latency-sensitive applications. Core clock frequency, cache strategies and communication latency between memory and working core have great impact on this indicator.

SoC Price – This metric considers the cost of the System-on-a-Chip (SoC) used in the low-power device. The price of the SoC affects the

overall affordability and feasibility of deploying model in large-scale distributed systems. Lower-cost SoCs can make the deployment more accessible and cost-effective.

3.2 Measurement Granularity

As for performance analysis of machine learning in other domains, TinyML performance can be measured at different granularity levels:

Per-Model Evaluation – At this coarse level, one measures the performance of the model as a whole, i.e. the resource footprint incurred by the execution of the model including all its layers and operators. For example, this allows for evaluating the resource consumption for inference with the production-ready code, on a particular industrial hardware setup.

Per-Operator Evaluation – At this level, one measures separately the performance of one or more operators (i.e. one or more components of the model). This per-operator measurement can help identify specific operators that contribute to performance or inefficiencies, in optimizing the model's efficiency and spotting potential bottlenecks.

4 Background on TinyML Model Update

A model deployed and running on a device can be updated at different levels. Depending on this level, flexibility and network traffic costs can vary.

Firmware Updates – As surprising as it may be, the dominant approach in the domain of low-power IoT software updates on microcontroller-based devices is to perform a firmware update, i.e. transferring, verifying and (re)installing the totality of the software running on the device (except a minimal bootloader, in some cases). A simplistic approach to update a model remotely over the network is thus to roll out and deploy a firmware update containing the new model. This level requires transmitting the largest amount of data to the device can bloat network load and buffer memory requirements.

Full Model Updates – A more refined approach to updating software remotely over the network is to roll out and deploy fractional software updates instead of firmware updates. With this approach, a user updates only model binary code, i.e. its operators and related parameters (weights, bias, etc.). Network transfer is decreased, while flexibility remains high, as a user can renew the operators with optimized execution flows, or even replace a model with totally different architecture with better performance in accuracy or resource utilization.

Partial Model Updates – An even more fine-grained approach is to update only a subset of specified parameters of an already-deployed model. For instance, a user can specify a single layer (or even: a single parameter) to be updated, which can minimize network traffic and related power consumption. Such fine-grained level may suit low-power IoT nodes with the same model exchanging parameters (federated/distributed learning), or receiving newer parameters from central server. However, note that flexibility is decreased, as such an update cannot significantly alter a model's architecture.

5 Threat Model & Security of TinyML Model Updates

Users trust machine learning models to provide accurate and unbiased predictions. To maintain model trustworthiness, at the very least, integrity checks and authorization during the model update process are necessary for ensuring the reliability of TinyML systems and to mitigate attacks such as model poisoning, where attackers attempt to manipulate training data or inject malicious bits into the model parameters.

In this paper, we do not consider sophisticated defenses against model poisoning. Rather, we consider five fundamental attack vectors during model update process:

Tampered Model Update – An adversary holds the model repository and tries to upload flawed models, which will be fetched by IoT devices for model update.

Tampered Parameter Update – same as the above, but at the granularity of (malicious) parameter tilting.

Unauthorized Model Update – An unauthorized adversary attempts to request the IoT device to fetch and deploy modified models.

Unauthorized Parameter Update – same as the above, but at the granularity of (unauthorized) parameter changes.

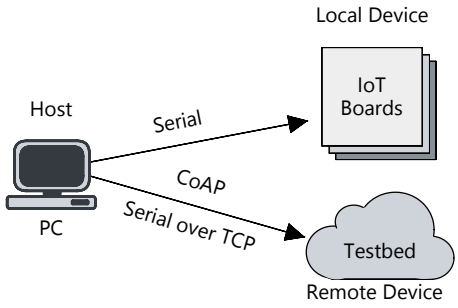

Fig. 1 Hardware setup of RIOT-ML. Users can connect local boards with host PC via serial, or use remote board service on IoT-Testbed.

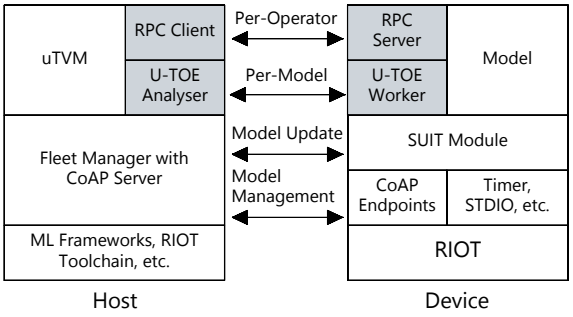

Fig. 2 Software architecture and components of the RIOT-ML framework. In grey: U-TOE toolkit components.

Break of Confidentiality – The updates to models or parameters should be encrypted such that only the authorized maintainer and the device have access to the decrypted model updates.

Note that this threat model has limits: it does not cover the case where the root of trust, i.e. the authorized model maintainer, is itself compromised. For instance, the authorized maintainer himself could go rogue, or be otherwise tricked into lacing the model updates with malware, or into malicious tilting of model parameters. Nevertheless, any threat model must cover the above vectors, and in this sense the security mechanisms we design are fundamental. They should allow extensions towards mitigating more sophisticated attacks.

6 RIOT-ML Framework

RIOT-ML integrates uTVM and RIOT to perform model compilation, optimization and flashing, secure model update, model management, and utilizes U-TOE for evaluation of arbitrary models produced by mainstream ML frameworks onto various low-power boards.

As depicted in Fig. 1, users can use a host personal computer (PC) (combining Linux and the RIOT-ML toolchain) to deploy, evaluate and update their models on local IoT devices (e.g. connected via Universal Serial Bus (USB) to their PC), or on remote IoT devices via a testbed offering bare-metal access to various types of MCUs over the network.

6.1 Architectural Design

As depicted in Fig. 2, the toolkit is composed of the following key components:

- **Model Compiler**. RIOT-ML leverages the uTVM compiler to convert arbitrary neural network models into efficient C code. This compiler takes as input the output of typical ML frameworks (such as PyTorch, TensorFlow...), then enhances the efficiency of the models for microcontrollers and enables them to be run on low-power devices.
- **OS Environment & Hardware Support**. RIOT is a general-purpose OS for low-power IoT devices, which was chosen to provide a lightweight runtime environment for model execution and evaluation on microcontrollers. This base provides extensibility and wide-spectrum support for heterogeneous low-power boards.
- **Model Evaluation Module**. This component integrates the U-TOE toolkit [13] to perform on-board model evaluation. Section 9 describes its architecture and measurement procedures in detail.
- **Secure Model Update Module**. This module integrates an implementation of Request for Comments (RFC) 9019 (SUIT [34]) to provide a secure, low-power software updates on IoT devices. It utilizes SUIT manifests, which contain model version information, payloads' hashes, and digital signatures from maintainers, to ensure the integrity and authenticity during update process. We adapted and integrated it into RIOT-ML to support secure OTA model update as described in Section 6.2.

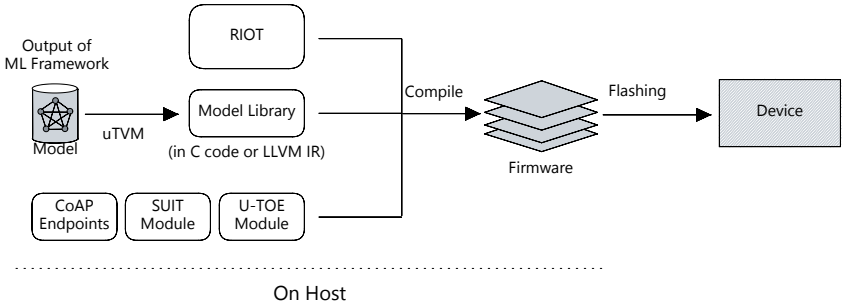

Fig. 3 Compilation and deployment workflow of RIOT-ML. uTVM optimizes and translates model from mainstream ML framework into model library, which is co-compiled and flashed with RIOT and U-TOE components onto target boards.

- **Network Management Endpoints**. RIOT-ML exposes CoAP endpoints on IoT devices running the framework, serving as interfaces for model management. Users can query model metadata and status (name, version, evaluation results, etc.), control model behaviors or trigger model update, as described in Section 6.2.
- **Fleet Management Module**. RIOT-ML implements fleet management on the host for remote updates of models, automating a build pipeline to support CI/CD of machine learning models on device. This model builds update payloads, generates and signs SUIT manifests, pushes them to a repository (CoAP registry) and notifies the managed IoT devices about model updates availability. It also provides a user-friendly interface.

Additionally, RIOT-ML provides a *connector* for cloud-based IoT testbed which enables seamless interaction with remote boards using serial over TCP.

Workflow. We depict in Fig. 3 the high-level view of the typical workflow with RIOT-ML. In a preliminary step, RIOT-ML first gathers the specification of target device to decide the compilation options for uTVM and RIOT. Then uTVM generates a non-optimized model "library". Some static optimization strategies are then applied in this stage, according to the target device type, in particular to determine an appropriate scheduling[5]. This uTVM-generated library is then jointly compiled with RIOT, an RPC server and a measurement worker into an executable firmware, which is then automatically flashed on the device (via USB, or remotely via the network).

[5]A schedule specifies low-level optimization for loop execution, enhancing cache hit and memory access. The optimal schedule is co-determined by model and device specification, and identified by heuristic search algorithms based on measurements on device [8].

6.2 Model Over-The-Air Update and Management

We provide interfaces and modules both on the IoT devices and on the host (fleet manager) to enable secure OTA update and management.

Architecture – On the one hand, software components embarked on IoT devices which can answer to management commands and act upon update notifications were implemented on top of RIOT's low-power IPv6 network stack and a SUIT implementation. Update and management interfaces were implemented as separate RIOT modules.The host on the other hand, as fleet manager, is responsible for building and coordinating updates, sending commands and notifications, and maintaining an update repository (a server with a CoAP registry) which stores update payloads.

Model Management – To remotely control, maintain and monitor ML model operation and performance efficiently and continuously, RIOT-ML exposes several CoAP endpoints on managed IoT devices, as shown in Table 2. These end-points can be accessed by the authorized maintainer via the fleet manager module.

Model Update Procedure – The procedure consists of two steps. In the first step depicted Fig. 4 an authorized maintainer identified with the public/private key pair (P_k, S_k) produces and pushes the update binary and the associated metadata secured as per SUIT (the SUIT manifest). In a second step depicted Fig. 5, the maintainer (using the fleet manager module) to notify the

Table 2 CoAP endpoints for model management on device.

CoAP Endpoints	Remarks
(For Monitor)	
/model/status	Get working status of model
/model/name	Get model name
/model/params/info	Get info of parameters
/model/eval_result	Fetch last evaluation results (compute latency, loss etc.)
(For Control)	
/model/stop	Stop the model
/model/run	Run the model
/model/run_eval	Evaluate the model
/model/params/update	Trigger partial update
(For SUIT)	
/suit/slot/(in)active	Get number of active or inactive firmware slot
/suit/trigger	Trigger firmware update
/suit/version	Get version of the current firmware

managed IoT devices that a new update should be fetched, verified and installed, which then ensues.

Note that, as per the SUIT specification, different schemes for digital signatures, hashing and encryption can be used (performance comparisons are available in prior work such as [5]). In the below, we assume the use of Secure Hash Algorithm 256-bit (SHA256) [12] for hashing, and Ed25519 algorithm [14] for digital signatures.

The following describes the update procedure in more detail:

1. (*Preliminary*: pre-provision the device with the authorized maintainer's public key P_k);
2. The fleet manager encapsulates the model binary (or the selected model parameters to be updated) into a SUIT payload format and computes the corresponding SHA256 digest as payload checksum;
3. The fleet manager generates and signs the SUIT manifest using an Ed25519 signature scheme, which contains the uniform resource identifier (URI) and checksum of the payload;
4. The fleet manager pushes the update binary and the SUIT manifest to the CoAP server (matching the URI).
5. Through a dedicated exposed CoAP endpoint, the device is notified with the URI of the new SUIT manifest;
6. The device fetches and verifies (using P_k) the SUIT manifest. If the verification fails, the update aborts and reports an error code;

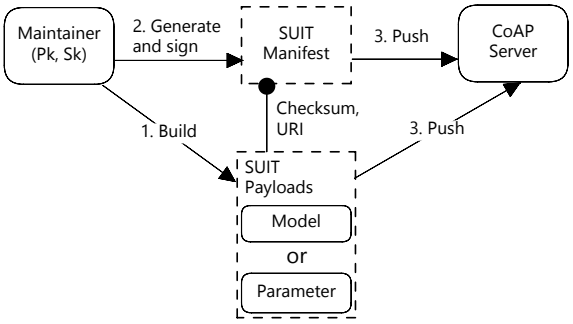

Fig. 4 Generation & pushing of SUIT manifest & payloads for secure model update with RIOT-ML. The key pair (P_k, S_k) authenticates the authorized maintainer. The CoAP server has no knowledge of secret key S_k and only serves as artifacts repository.

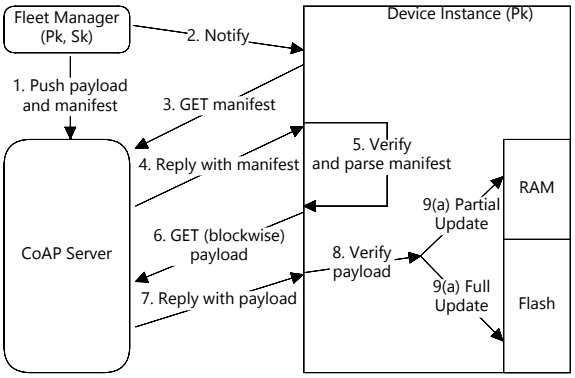

Fig. 5 Notification, fetching, verification and installation of model updates with RIOT-ML, securely over the network.

7. The device fetches the update binary (payload) designated in the SUIT manifest and performs integrity check. If the verification fails, the update aborts and reports an error code;
8. The device installs the new model in RAM or in Flash memory, depending on the part of the model that is updated.

6.3 Security Guarantees with RIOT-ML Over-the-Air Updates

To counter the potential attacks using model update as vector, RIOT-ML uses SUIT to provide the following guarantees:

Model Update Integrity – RIOT-ML inspects the checksums of the update payloads and the metadata specified by SUIT, signed by the

model maintainer. This ensures the integrity and prevents tampered update payload.

Model Update Authenticity – By using the combination of a digital signature mechanism and a cryptographically secure digest specified by SUIT, the RIOT-ML framework guarantees that only the users with the valid private keys have the authority to update the model.

Model Update Confidentiality – The IETF also specifies (optional) encryption mechanisms for SUIT payload binaries (see [32]). For this, a symmetric content encryption key is used, this key being either pre-shared, or established on-the-fly for instance via an Ephemeral-Static Diffie-Hellman exchange (EDHOC) [27]. If used, this mechanism provides confidentiality for the updates of models.

7 U-TOE Toolkit

U-TOE is a toolkit we designed and implemented, integrating RIOT and uTVM to perform model compilation and evaluation, on-board a microcontroller-based device.

7.1 Architectural Design

The toolkit is composed of the following key components:

- **RPC Module**. To evaluate the resource consumption at operator level, U-TOE utilizes the Remote Procedure Call (RPC) mechanism of uTVM. The RPC mechanism enables to upload and launch functions onto IoT boards over serial. This is useful for remote testing and profiling, enabling U-TOE to wrap the model operators for measuring the computational latency and memory usage. It is composed of a client on the host and a server on the target device, receiving commands and executable instructions from the host.
- **Evaluation Module**. It contains two units: measurement worker and analyser. As shown in Fig. 2, the measurement worker is deployed on the MCU for acquiring performance metrics at model or operator level. Besides carrying out the measurement of resource footprint, it is in charge of the randomization of model input, and is responsible for reporting metrics data to host device. The analyser runs on the host, statisticizes the uploaded metrics from device and provides a human-readable frontend for users.

Evaluation Workflow with U-TOE – After deploying the program on the target device, a bidirectional channel is set up between host and device, as depicted in Fig. 2. The measurement worker starts collecting performance metrics at user-specified level and uploads metrics data to analyser. Eventually, users can obtain the overall statistics of model metrics, or catch the performance bottleneck with execution details of each operator. All the raw metrics data are saved in a log file for further, user-customized analysis.

7.2 Measurement Procedure

We designed two measurement procedures to support evaluation at different granularity. The procedures run across multiple components of the toolkit and most workloads are primarily on the target board. The following steps describe the measurement routine after compilation of executable program. The steps marked with **bold number** are executed on the target board.

Per-Model Evaluation – This mode focuses on the model performance in actual production environment. Here is the corresponding measurement routine:

1. Calculate model memory and storage consumption based on Executable and Linkable Format (ELF) file. We disable dynamic memory allocation to enable static analysis of memory footprint.
2. Deploy executable program to local or remote IoT board.
3. Repeat model inference based on the user-specific number of trials with randomized input on uniform distribution.
4. Record computational latency of each trial.
5. Upload records to host device for further analysis and archive.

At the end of evaluation, results with statistics (e.g. 95% confidence interval, median, maximum and minimum) are presented on the host device, including computational latency and consumption of memory and storage.

Per-Operator Evaluation – In contrast to per-model evaluation, this mode focuses on the efficiency and resource footprint of each operator,

enabling to discover the performance bottleneck inside models. Thanks to the time evaluator inside the uTVM's RPC mechanism, we can measure computational latency at operator level. The high abstraction of timer and serial in RIOT allows us to unify the implementation of time measurement and RPC communication on arbitrary IoT boards. Here is the corresponding measurement routine:

1. Analysis memory footprint at operator level utilizing internal API of TVM.
2. Deploy executable program to local or remote IoT board.
3. Start RPC Server on IoT board.
4. Launch RPC client to benchmark and record execution performance of each operator.

It is noted that the operator structure constructed by uTVM is usually inconsistent with the hand-crafted version in ML framework. That's because uTVM as model compiler applies model optimization (i.e., operator fusion) and inserts execution details (i.e., quantization arithmetic) during conversion and compilation, which potentially merges multiple operators into a single one or inserts additional operators. Nevertheless, we annotate the operators from uTVM with model parameters (weights, biases, etc.), so that users can associate a specific operator to the corresponding layer.

8 Experiments With RIOT-ML

We next demonstrate the capabilities of RIOT-ML. We report on two categories of experiments that were carried out. In this section we first describe the experimental setup for each category of experiments, then Section 9 will present and analyze the results we gathered.

8.1 Model Performance Evaluation

We conducted experiments to validate the functionality and compatibility of RIOT-ML on model side (universal support for model structure and ML frameworks) and on device side (wide-spectrum support for IoT devices). Hence, we dived into two orthogonal directions: For device support, we evaluated a quantized LeNet-5 on various IoT boards; For model compatibility, we evaluated multiple models on a local STM32F746G discovery board.

Model Selection – We selected pre-trained, quantized models from open source repositories[6], which target on typical TinyML tasks (Image Classification, Key Word Spotting, Visual Word Wake, Noise Suppression and Abnormal Detection). The weights and activations of the model were quantized to 8-bit integer, yet the inputs and outputs remain in IEEE 754 single-precision floating-point [15] format.

Model Optimization – We only used built-in, rule-based optimization in uTVM. Thus, all heuristic optimization strategies like model scheduling were disabled.

MCU Configuration – We disabled data and instruction cache to observe the "memory wall" effect in ML model. The core clock frequency was pre-set by CPU initialization code in RIOT and is presented with experiment results in Section 9.

Hybrid Deployment – The experiments were conducted both on local and remote IoT boards provided by FIT IoT-LAB.

It is noted that for each evaluation we preset the number of trials to ten in order to address random error.

8.2 Secure Model OTA Updates

We conducted experiments updating models running on a nRF52840dk Development Kit board in the FIT IoT-LAB testbed, and measured the resource consumption of significant components and the network transfer overhead. Our measurements used the LeNet-5 model as use case. We consider different granularity for the update: either updating the firmware or, instead, a partial update concerning only the weights of the final layer (serving as classifier) of the quantized LeNet-5 model.

Network Setup – The nRF52840 board is connected to the network through a low-power IEEE 802.15.4 radio access link (using the Generic Network Stack (GNRC) in RIOT) on which

[6] see https://github.com/ARM-software/ML-zoo and https://github.com/mlcommons/tiny

6LoWPAN, IPv6 traffic can flow. Via an intermediate IPv6 border router, CoAP messages can transit to/from the device and reach the fleet manager hosted on the user's PC connected to the Internet. By using this approach (6LoWPAN and IPv6) we ensure that the approach can run end-to-end over the Internet over an arbitrary set of links which can include not only typical Ethernet and WiFi links, but also some low-power radio links (e.g. Bluetooth Low Energy (BLE), Long Range (LoRa), IEEE 802.15.4).

Crypto Configuration – We produced public and secret keys using Ed25519 algorithm before commissioning the model update modules. CBOR Object Signing and Encryption (COSE) [26] was used as specified by SUIT to sign the manifest and CoAP payloads. The crypto libraries we used in practice are libcose[7] and c25519[8], which target on low-memory systems, and implement COSE and Ed25519, respectively.

9 Analysing RIOT-ML Measurements

In this section, we present the results of the experiments we describe in Section 8.

9.1 Model Performance Evaluation

Per-Model Evaluation – Table 3 presents the resource consumption of LeNet-5 model on various IoT boards, generated by Per-Model evaluation. MCUs are grouped by family and arranged in ascending order of clock frequency within each group. ARM Cortex-M series MCUs showed no significant difference in memory and storage usage, and the computational latency declined as the core frequency increased. One **outlier** is the RP2040-based rpi-pico board, but the extra 16KB RAM is in fact reserved for the debugger. Benefits from full support of Digital Signal Processing (DSP) and Thumb-2 instruction set, Cortex-M3, -M4 MCUs perform better than Cortex-M0+ with the same core clock frequency. Another **outlier** was discovered on SiFive RISC-V MCU. With the highest core clock frequency, this won the least favorable ranking on computational latency and memory usage. This MCU uses an external, Serial Peripheral Interface (SPI) NOR flash for data and program storage, causing a huge performance regression while we disabled the cache.

Table 4 presents the results of various ML models on representative TinyML tasks on individual IoT boards, showcasing the universal support for various ML frameworks and model structures. Except for LeNet-5 trained on a local host device with Pytorch, all the others came from open source model zoos. Memory and storage columns refer to their resource consumption.

Unsurprisingly we observe how storage consumption decreases proportionally to decreasing the number of model parameters. One apparent outlier is *DS-CNN Small*, which consumes more storage space than *LeNet-5*, although it has nearly twice the number of model parameters. Further analysis revealed however that *DS-CNN Small* contains almost three times more scaling factors (in floating-point) compared to *LeNet-5*. Those scaling factors are not counted as model parameters but occupy a large amount of storage.

As expected, the most complex model, *MobileNetV1*, consumed the most memory (RAM) and computation resources. However, as shown in Table 4, the execution overhead, memory consumption and computational latency, cannot be reliably predicted solely from the amount of model parameters in general. The model structure and its associated computational pattern, as well as the effects of compression also impact intensely on the execution overhead (hence the usefulness of RIOT-ML as an experimental toolkit for benchmarking!).

Per-Operator Evaluation – We here used a tiny model with only three layers from TFlite as an example to avoid unnecessary complexity in demonstration, with output results presented in Table 5. The computational bottlenecks are located in operator *add_nn_relu* and *add_nn_relu_1*, and with the highest memory and storage consumption as well. We can trace down the corresponding layers of the original model with the hints of associated parameters, which are the weights, bias or other trainable parameters of the model, making it possible to apply optimization strategies on well-targeted layers.

[7] see https://github.com/bergzand/libcose
[8] see https://www.dlbeer.co.nz/oss/c25519.html

Table 3 Evaluation of LeNet-5 model on various IoT boards.

Board / MCU	Core	Memory (KB)	Storage (KB)	Computational Latency (ms)			
				95%-CI	Median	Min.	Max.
b-l072z-lrwan1 / STM32L072CZ	M0+ @ 32 MHz	11.288	64.340	[261.829, 262.249]	262.187	261.350	262.216
samr30-xpro / ATSAMR30G18A	M0+ @ 48 MHz	11.208	65.168	[176.936, 176.965]	176.958	176.924	176.975
arduino-zero / ATSAMD21G18	M0+ @ 48 MHz	11.292	64.940	[182.061, 182.082]	182.068	182.051	182.098
rpi-pico / RP2040	M0+ @ 125 MHz	28.704	65.172	[70.108, 70.130]	70.117	70.091	70.151
openmote-b / CC2538SF53	M3 @ 32 MHz	11.100	66.080	[200.337, 200.384]	200.367	200.323	200.404
IoT-LAB M3 / STM32F103REY	M3 @ 72 MHz	11.296	62.260	[97.740, 97.757]	97.751	97.733	97.764
nucleo-wl55jc / STM32WL55JC	M4 @ 48 MHz	11.288	63.180	[98.649, 98.668]	98.661	98.637	98.679
nrf52840dk / nRF52840	M4 @ 64 MHz	11.348	61.332	[66.078, 66.112]	66.088	66.087	66.163
b-l475e-iot01a / STM32L475VG	M4 @ 80 MHz	11.288	61.604	[52.900, 52.901]	52.901	52.900	52.902
stm32f746g-disco / STM32F746NG	M7 @ 216 MHz	11.076	64.712	[39.600, 39.602]	39.601	39.599	39.604
esp32-wroom-32 / ESP32-D0WDQ6	ESP32 @ 80 MHz	115.958	157.719	[85.580, 85.583]	85.582	85.576	85.584
esp32c3-devkit / ESP32-C3FN4	RISC-V @ 80 MHz	258.874	222.272	[54.947, 54.957]	54.953	54.938	54.961
sipeed-longan-nano / GD32VF103CBT6	RISC-V @ 108 MHz	103.108	106.422	[37.783, 37.789]	37.789	37.779	37.791
hifive1b / SiFive FE310-G002	RISC-V @ 320 MHz	60.884	66.492	[153.621, 154.166]	153.747	153.717	154.938

Table 4 Evaluation of various quantized models on stm32f746-disco board. The numbers of model parameters are presented alongside model names.

Model (#Parameters)	Task	Memory (KB)	Storage (KB)	Computational Latency (ms)			
				95%-CI	Median	Min.	Max.
MobileNetV1-0.25x[1] (500K)	Visual Wake Words	185.352	491.668	[1435.937, 1435.938]	1435.938	1435.938	1435.939
Deep AutoEncoder[1] (264K)	Anomaly Detection	6.532	292.696	[35.637, 35.638]	35.638	35.638	35.639
RNNoise[2] (87K)	Noise Suppression	4.688	119.652	[12.151, 12.157]	12.154	12.148	12.160
LeNet-5 (40K)	Image Classification	12.068	65.851	[39.599, 39.603]	39.601	39.598	39.605
DS-CNN Small[2] (22K)	Keyword Spotting	68.992	71.796	[461.395, 461.396]	461.396	461.396	461.397

[1]These models originate from MLPerf Tiny Benchmarks repository on https://github.com/mlcommons/tiny.

[2]These models originate from ARM Model Zoo on https://github.com/ARM-software/ML-zoo.

All models were pre-trained and quantized in INT8 by TFLite, except LeNet-5 was by Pytorch.

Table 5 Per-Operator Evaluation Output of TFlite sinus model on stm32f746-disco board.

Operators	Time (us)	Time (%)	Params	Memory	Storage
add_nn_relu	8.856	15.22%	p0, p1	0.128	0.128
add_nn_relu_1	46.682	80.23%	p2, p3	0.128	1.088
add	2.646	4.54%	p4, p5	0.068	0.068

The uTVM auto-generated prefix *tvmgen_default_fused_nn_dense_* of operator name is not presented for the purpose of clarity. Memory and storage consumption are presented in KB.

9.2 Secure Model OTA Updates

We show in Table 6 a breakdown of resource consumption. Since full- and partial-updates share the same software stack, their consumption remains identical. We observe that, due to its large number of constant model parameters (weights and bias) the ML model is the component which uses the most storage (Flash memory). Concerning (RAM) memory usage, the network stack and SUIT module use relatively more memory because of the large buffer requirements for network packets and SUIT payloads handling. We also observe that the cryptographic implementations employed by SUIT (namely libcose and c25519) claim minimal memory and storage consumption.

Table 6 Storage consumption and transfer costs of model updates on nrf52840dk board with deployed LeNet-5 model.

Entity	Memory	%	Storage (Transfer)	%
MCU	256.000	-	1024.000	-
Firmware	43.864	100.00	126.374	100.00
ML Model	12.420	28.31	49.858	39.45
Network Stack	13.714	31.26	35.121	27.79
SUIT	13.933	31.76	15.351	12.14
-Crypto	1.340	3.05	5.826	4.61
SUIT Manifest	-	-	0.471	0.37
Final Layer	-	-	0.840	0.67

In these measurements we did not include SUIT encryption, only signature verification and integrity check. Memory and storage consumption are presented in KB.

9.3 Preliminary assessment of network transfer costs

We additionally measured in Table 6 the size of SUIT manifest and update binary payloads which must be transmitted over the network to carry out the model update process. These measurements can help gauge the incurred load on the network. Concerning the payloads, the serialized firmware's size and the weights of the final layer are 123.44 KB and 0.84 KB, respectively. Comparatively, the SUIT manifest's size is 0.471 KB, which represents a rough ratio of 260:2:1. This means in particular that the overhead on network load incurred by using SUIT security mechanisms in RIOT-ML (for integrity, authentication and authorization) remains well under 1% when the approach of firmware updates is used to update the models.

Discussion on Model Update Granularity – The simplest approach to model update over-the-air on IoT devices is firmware updates, as the embedded system is relieved of the complexities stemming from dynamic loading on heterogeneous hardware. However, this convenience comes with a high price in terms of network transfer costs. The measurements in Table 6 hint that updating only the model (49KB) instead of the full firmware (126KB) would shave more than 60% off the network transfer costs. Going further, if in some cases model updates could get away with transfer learning, i.e. updating only parts of the model network transfer costs become comparatively negligible. In our measurements, for instance, we updated only the last layer, which eliminated more than 99% of the network transfer costs. However, with this approach inference accuracy improvements with the update may be more limited. Thus, users are obliged to balance between network overheads, technical complexity, and model accuracy when designing their model update scheme.

10 Reproducible & Custom RIOT-ML Experiments

We released the full source code of the RIOT-ML toolkit on Github at https://github.com/TinyPART/RIOT-ML under an open source LGPL v3 license. For further details on how to start with RIOT-ML hands-on, the reader is referred to the comprehensive *Readme.md* in the repository.

On the one hand, researchers and practitioners who possess IoT hardware that is supported by the open source operating system RIOT (currently 250+ types of boards, using 60+ types of CPUs[9] can use RIOT-ML out-of-the-box, directly on their boards.

On the other hand, combined with the use of the free open-access testbed IoT-Lab[10], even researchers and practitioners who do not have such hardware on-premises can conduct large-scale experimental evaluation campaigns using RIOT-ML.

Perspectives – As RIOT board and CPU support expands and improves over time, and as uTVM also expands support to other architectures in parallel (both open source communities are very active) RIOT-ML can in a very short time expand its support for new use cases, automatically adding the support of uTVM for new boards, and the support of RIOT for new models. As such RIOT-ML may organically grow and become a useful link between the two communities.

Moreover, while the work on RIOT-ML in this paper has been focused on inference only on single-core microcontrollers, there is strong potential to extend the toolkit provided by RIOT-ML to support on-device learning scenarios, and for optimizing exploitation of multi-core microcontrollers.

[9] see https://github.com/RIOT-OS/RIOT/tree/master/boards

[10] see https://www.iot-lab.info/

11 Conclusion

This paper introduces RIOT-ML, a novel toolkit we designed to streamline the endeavors of AIoT practitioners. RIOT-ML facilitates TinyML model continuous integration, secure continuous deployment and performance evaluation, remotely over the network, on fleets of heterogeneous IoT devices under test. More precisely: with RIOT-ML, users can take the model zoo output by various traditional machine learning frameworks (such as PyTorch, TensorFlow) and automate their adapting, their deployment and the assessment of their performance when executed on a wide selection of IoT boards and development kits based on various types of low-power microcontrollers (ARM Cortex-M, ESP32, RISC-V). Enabling and facilitating such large test matrices is indeed crucial for advancing the field of AIoT. To this end, we also published a highly reusable, well-documented, and customizable open-source implementation of RIOT-ML, which leverages the vibrant open-source communities associated with RIOT and uTVM. Last but not least, we showcase the practical application of RIOT-ML by providing preliminary experimental evaluation results on an open-access testbed.

Acknowledgment

The authors would like to thank Cedric Adjih, Nadjib Achir and Felix Biessmann for useful discussions and suggestions. The research leading to these results partly received funding from the MESRI-BMBF German/French cybersecurity program under grant agreements No. ANR-20-CYAL-0005 and 16KIS1395K. The paper reflects only the authors' views. MESRI and BMBF are not responsible for any use that may be made of the information it contains.

Conflicts of Interest

Not applicable.

Data Availability

Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

References

[1] Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., et al. (2015). Fit iot-lab: A large scale open experimental iot testbed. *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 459–464.

[2] Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M. S., Petersen, H., Schleiser, K., Schmidt, T. C., & Wählisch, M. (2018). Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, *5*(6), 4428–4440.

[3] Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D., et al. (2021). Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597*.

[4] Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., Fazel, A., Holleman, J., Huang, X., Hurtado, R., Kanter, D., Lokhmotov, A., et al. (2020). Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*.

[5] Banegas, G., Zandberg, K., Baccelli, E., Herrmann, A., & Smith, B. (2022). Quantum-resistant software update security on low-power networked embedded devices. *International Conference on Applied Cryptography and Network Security*, 872–891.

[6] Bauwens, J., Ruckebusch, P., Giannoulis, S., Moerman, I., & De Poorter, E. (2020). Over-the-air software updates in the internet of things: An overview of key principles. *IEEE Communications Magazine*, *58*(2), 35–41.

[7] Bormann, C., Castellani, A. P., & Shelby, Z. (2012). CoAP: An application protocol for billions of tiny Internet nodes. *IEEE Internet Computing*, *16*(2), 62–67.

[8] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. (2018). TVM: An automated End-to-End optimizing compiler for deep learning, 578–594.

[9] Diaz-de-Arcaya, J., Torre-Bastida, A. I., Zárate, G., Miñón, R., & Almeida, A. (2023). A Joint Study of the Challenges,

Opportunities, and Roadmap of MLOps and AIOps: A Systematic Survey. *ACM Computing Surveys*, *56*(4), 84:1–84:30. https://doi.org/10.1145/3625289

[10] Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., & Razafindralambo, T. (2011). A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, *49*(11), 58–67.

[11] Hahm, O., Baccelli, E., Petersen, H., & Tsiftes, N. (2015). Operating systems for low-end devices in the internet of things: A survey. *IEEE Internet of Things Journal*, *3*(5), 720–734.

[12] Hansen, T., & 3rd, D. E. E. (2006). US Secure Hash Algorithms (SHA and HMAC-SHA). https://doi.org/10.17487/RFC4634

[13] Huang, Z., Zandberg, K., Schleiser, K., & Baccelli, E. (2023). U-TOE: Universal TinyML On-Board Evaluation Toolkit for Low-Power IoT. *2023 12th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*, 1–6.

[14] Josefsson, S., & Liusvaara, I. (2017). Edwards-Curve Digital Signature Algorithm (EdDSA). https://doi.org/10.17487/RFC8032

[15] Kahan, W. (1996). Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, *754*(94720-1776), 11.

[16] Kreuzberger, D., Kühl, N., & Hirschl, S. (2023). Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, *11*, 31866–31879. https://doi.org/10.1109/ACCESS.2023.3262138

[17] Lê, M. T., & Arbel, J. (2023). TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. *Proceedings of the 3rd Workshop on Machine Learning and Systems*, 148–153.

[18] Lima, L. E., Kimura, B. Y. L., & Rosset, V. (2019). Experimental environments for the internet of things: A review. *IEEE Sensors Journal*, *19*(9), 3203–3211.

[19] Moran, B., Tschofenig, H., Brown, D., & Meriac, M. (2021). A Firmware Update Architecture for Internet of Things. https://doi.org/10.17487/RFC9019

[20] Mtetwa, N. S., Tarwireyi, P., Abu-Mahfouz, A. M., & Adigun, M. O. (2019). Secure firmware updates in the nternet of things: A survey. *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, 1–7.

[21] Osman, A., Abid, U., Gemma, L., Perotto, M., & Brunelli, D. (2022). Tinyml platforms benchmarking. In *Applications in electronics pervading industry, environment and society: Applepies 2021* (pp. 139–148). Springer.

[22] Qiu, H., Vavelidou, I., Li, J., Pergament, E., Warden, P., Chinchali, S., Asgar, Z., & Katti, S. (2022). Ml-exray: Visibility into ml deployment on the edge. *Proceedings of Machine Learning and Systems*, *4*, 337–351.

[23] Ray, P. P. (2022). A review on tinyml: State-of-the-art and prospects. *Journal of King Saud University-Computer and Information Sciences*, *34*(4), 1595–1623.

[24] Saha, S. S., Sandha, S. S., & Srivastava, M. (2022). Machine learning for microcontroller-class hardware-a review. *IEEE Sensors Journal*, *22*(22), 21362–21390.

[25] Sanchez-Iborra, R., & Skarmeta, A. F. (2020). Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, *20*(3), 4–18.

[26] Schaad, J. (2017). CBOR Object Signing and Encryption (COSE). https://doi.org/10.17487/RFC8152

[27] Selander, G., Mattsson, J. P., & Palombini, F. (2024). Ephemeral Diffie-Hellman Over COSE (EDHOC). https://doi.org/10.17487/RFC9528

[28] Sharma, H., Haque, A., & Blaabjerg, F. (2021). Machine learning in wireless sensor networks for smart cities: A survey. *Electronics*, *10*(9), 1012.

[29] Shelby, Z., Hartke, K., & Bormann, C. (2014). RFC 7252: The Constrained Application Protocol (CoAP).

[30] Sudharsan, B., Salerno, S., Nguyen, D.-D., Yahya, M., Wahid, A., Yadav, P., Breslin, J. G., & Ali, M. I. (2021). Tinyml benchmark: Executing fully connected neural networks on commodity microcontrollers. *2021*

IEEE 7th World Forum on Internet of Things (WF-IoT), 883–884.

[31] Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295–2329.

[32] Tschofenig, H., Housley, R., Moran, B., Brown, D., & Takayama, K. (2023). *Encrypted Payloads in SUIT Manifests* (Internet-Draft draft-ietf-suit-firmware-encryption-18) [Work in Progress]. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-suit-firmware-encryption/18/

[33] Yousefzadeh-Asl-Miandoab, E., Robroek, T., & Tozun, P. (2023). Profiling and Monitoring Deep Learning Training Tasks. *Proceedings of the 3rd Workshop on Machine Learning and Systems*, 18–25. https://doi.org/10.1145/3578356.3592589

[34] Zandberg, K., Schleiser, K., Acosta, F., Tschofenig, H., & Baccelli, E. (2019). Secure Firmware Updates for Constrained IoT Devices using Open Standards: A Reality Check. *IEEE Access*, *7*, 71907–71920.