

Efficient Intrusion Detection for IoT Networks Using Machine Learning Models on ESP32 Microcontroller

Henrik Steen

Faculty of Computer Science,
Engineering and Economics
Østfold University of Applied Sciences
Halden, Norway
henrik.steen@hiof.no

Godspower Ogaga Uti

Faculty of Computer Science,
Engineering and Economics
Østfold University of Applied Sciences
Halden, Norway
godspowu@hiof.no

Thi Thuy Nga Dinh

Faculty of Computer Science,
Engineering and Economics
Østfold University of Applied Sciences
Halden, Norway
thi.t.dinh@hiof.no

Abstract—The number of devices connected to the Internet of Things (IoT) is growing rapidly, and many of these devices can communicate directly over wireless technology and other non-Ethernet links that bypass the centralized firewalls and Intrusion Detection Systems (IDSs) that traditionally protect a network. Running an IDS locally on the device is therefore attractive, but typical IoT microcontrollers offer only hundreds of kilobytes of memory and clock speeds in the low hundreds of MHz, which is well below what conventional machine-learning-based IDSs require. This paper investigates whether IDS inference can run directly inside an IoT network node, how much labelled test data can be hosted on-device, and which performance metrics can be measured locally. We trained a broad set of classifiers (Decision Tree, Random Forest, XGBoost, a Feedforward Neural Network, a Support Vector Machine and LightGBM) on the TON_IoT network dataset and deployed the four that fit the device: a Decision Tree and a LightGBM model for multi-class attack detection, and a Feedforward Neural Network and a Support Vector Machine for binary detection. Each model is evaluated on 7,360 test samples embedded directly in flash, at CPU clock frequencies of 240, 160, 80 and 40 MHz. The deployed models are compact, ranging from 4.8 to 237 KiB of flash and 0 to 7.9 KiB of RAM. For ten-class detection the Decision Tree reached 99.14% accuracy at 68,050 classifications per second; for binary detection the Neural Network reached 99.92% accuracy and the Support Vector Machine 99.89%. The two tasks are reported separately and are not compared directly. Classification quality stayed essentially constant as the clock was throttled. Experimental results show that competitive on-device IDS inference is achievable on a low-cost microcontroller operating well below its peak clock, leaving 84–91% of flash and 91–93% of RAM free for the device's primary task.

Keywords—TinyML, intrusion detection systems, Internet of Things (IoT), edge computing, machine learning, ESP32, network security, TON_IoT dataset.

I. INTRODUCTION

The number of connected IoT devices continues to grow at an accelerating pace, with estimates of roughly 55.7 billion devices online by 2025 [1]. Many of these devices will communicate directly over a wireless connection or other non-Ethernet links, and therefore bypass the centralized routers and firewalls where network-level intrusion detection has traditionally been deployed. As the volume of transmitted data continues

to increase, effective monitoring of network traffic becomes essential for identifying malicious activities that may disrupt network operations or enable unauthorized access to sensitive information. Although IDSs can be deployed at access points or base stations that connect IoT devices to the network, such centralized deployments may introduce long latency for deployed applications in large-scale IoT networks. At the same time, the microcontrollers that typically power such IoT devices have only hundreds of kilobytes of SRAM for data and instructions and clock speeds in the low hundreds of MHz, which rules out conventional machine-learning-based IDSs that rely on large models and high-performance CPUs.

This paper addresses three concrete problems that follow from this mismatch. First, can a regular IoT microcontroller execute IDS inference entirely on-device for both *multi-class* attack detection (distinguishing between specific attack families) and *binary* detection (benign versus malicious), without offloading to the cloud? Second, how much of the labelled test data can be embedded into the same device alongside the model, so that the evaluation reflects a realistic, fully self-contained deployment? Third, what performance and efficiency metrics can be collected directly from the device across a range of CPU frequencies, so that the trade-off between detection quality, throughput and resource use can be characterized empirically? We target the ESP32 as a representative low-cost microcontroller and use the TON_IoT telemetry dataset [2]–[4], which is widely adopted as a benchmark for realistic IoT network intrusion data.

The scope of this work is the detection stage of an IDS. Packet capture, flow reconstruction and feature computation are not implemented on the device: the models classify feature vectors that were prepared offline and stored in flash. Throughput figures therefore characterize classifier execution and bound, rather than measure, what a complete on-device pipeline could sustain.

A. Related work

The most directly related study is [5], which evaluated four lightweight models (a Decision Tree, LightGBM, a Feedforward Neural Network and a TinyML neural network) on the TON_IoT dataset. To approximate IoT-class hardware the

cores and clock of a Raspberry Pi 4 were throttled, giving an F1-score of 0.976 and an estimated 120,000 packets per second for the TinyML network. That evaluation ran on a Linux host under CPython, and microcontroller performance was derived from a packets-per-MHz formula rather than measured; deployment on real microcontrollers was left as future work. Our study closes that gap, taking the same dataset and model families and running them on a physical ESP32.

Broader work confirms a shift from centralized detection toward on-device approaches. A survey of IDS architectures for IoT [6] stresses the need for resource-efficient detection, but implements nothing on constrained hardware. Classifiers such as CART and XGBoost have been compared on a Raspberry Pi 3 [7], which remains substantially more capable than a microcontroller. A two-stage Random Forest for mobile IoT [8] is evaluated in simulation and reports only accuracy, and a TinyML IDS on a smart thermostat [9] covers only binary classification on one target. Table I summarizes what each study measures.

The enabling technologies are TinyML and embedded-inference frameworks. TinyML uses quantization, pruning and compact architectures to fit neural networks into a few hundred kilobytes [10], [11]. TensorFlow Lite for Microcontrollers [12] executes quantized models on Arm Cortex-M and Xtensa devices, while emlearn [13] converts trained models into portable C. This paper addresses the gaps in Table I directly, by implementing and measuring four trained models on the ESP32 itself.

TABLE I. RELATED WORK ON ML-BASED IDS FOR IOT: WHAT WAS ACTUALLY MEASURED.

Work	Hardware	Task	On-device measured	Size/RAM/throughput
[5]	RPi 4 (throttled)	Multi	No	Estimated
[6]	—	Survey	No	No
[7]	RPi 3	Multi	Yes	Partial
[8]	Simulated	Binary	No	No
[9]	Thermostat	Binary	Yes	Partial
This paper	ESP32	Both	Yes	Yes

B. Contributions

To answer these questions we trained and compared four model families: Decision Tree, LightGBM, Feedforward Neural Network, and Support Vector Machine. These four met the size and throughput budget of the ESP32 and were deployed on real hardware.

The aim of this paper is to establish what is achievable on commodity IoT silicon and to quantify the gap between estimated and measured microcontroller performance.

- We deploy and measure four machine-learning IDS models directly on a commodity ESP32 microcontroller, covering both multi-class and binary detection on a single device, and test the packets-per-MHz extrapolation used in [5] against measurement.
- We measure each model’s true cost on the device as its contribution to the compiled firmware, which makes generated C and interpreted flatbuffers comparable for the first time, and report flash, static RAM, peak heap and stack for every deployment.

- We show that the frequency response separates the deployed models into two groups: flash-resident generated code is memory-bound and retains most of its throughput under clock reduction, whereas linear-kernel inference is compute-bound and loses it proportionally.
- We characterize each model across four CPU clock frequencies (240, 160, 80 and 40 MHz) using metrics collected locally, and report cross-validated detection quality alongside the on-device results.
- We embed 7,360 labelled test samples in flash so that on-device inference does not depend on any host-side data delivery, giving a self-contained evaluation closer to a real deployment than a host-driven one.

II. DATASET

The training and evaluation of all models is based on the TON_IoT Train-Test Network dataset, originally produced at the UNSW Canberra Cyber testbed [2], [3]. The testbed was designed to replicate realistic Industry 4.0 and Industrial IoT architectures, generating telemetry, operating-system and network data under both benign and attack conditions. Among the dataset variants released by the authors, the Train-Test Network subset is the network-flow split intended for benchmarking machine-learning-based network intrusion detection, and is widely used in follow-up work [4].

The Train-Test Network subset contains 211,043 labelled network flows with 44 raw flow-level features. Two label fields are available: a binary label (benign versus malicious) and a multi-class type field that distinguishes between nine attack families plus the normal class. The attack families are Scanning, Distributed Denial of Service (DDoS), Denial of Service (DoS), Backdoor, Ransomware, Cross-Site Scripting (XSS), Password cracking, Data injection and Man-in-the-Middle (MITM). The class distribution of the original dataset is shown in Table II. Most attack categories contribute 20,000 flows each, the normal class contributes 50,000 flows, and MITM is heavily underrepresented at 1,043 flows.

TABLE II. CLASS DISTRIBUTION OF THE TON_IoT TRAIN-TEST NETWORK SUBSET, AS RELEASED AND AFTER PREPROCESSING.

Class	As released	After preprocessing
Normal	50,000	41,946
Backdoor	20,000	18,710
DDoS	20,000	19,993
DoS	20,000	14,669
Injection	20,000	19,964
Password	20,000	19,859
Ransomware	20,000	14,704
Scanning	20,000	18,615
XSS	20,000	14,521
MITM	1,043	1,031
Total	211,043	184,012

For our experiments we follow standard practice for this dataset. Leakage-prone identifiers (such as the destination IP address) are removed, and when training the multi-class models, the binary label are removed; columns populated in fewer than 60% of flows are dropped; duplicate rows are removed; categorical fields are label-encoded; and the data is then split into an 80/20 train/test partition with a fixed random

seed. Because MITM is heavily underrepresented, we verify after splitting that it is present in both partitions rather than enforcing stratification. The neural network and the Support Vector Machine use standardized features, with the scaler fitted on the training partition only so that no test information reaches training. After preprocessing, each input flow is represented by 83 features, which is the input dimensionality embedded into the device.

Table II reports the dataset as released and after preprocessing. Dropping the 22 sparsely populated columns and removing exactly duplicated flows eliminates 27,031 rows, or 12.81% of the data, leaving 184,012 flows split into 147,209 training and 36,803 test flows. Both the binary and the multi-class models are evaluated on the same 36,803-flow test partition.

Of the 36,803 test flows, 7,360 are embedded as a binary array directly in device flash, so that on-device inference does not depend on any host-side data delivery. As Section IV-C shows, this is the largest set that fits alongside the largest of the four models; the constraint is flash capacity, not SRAM.

III. PROPOSED SOLUTION

The implementation is split into three components: an offline training and conversion pipeline; a small set of Python scripts that turn the held-out test split into C header files; and a per-model PlatformIO project that compiles each converted model together with its embedded test data and runs it on the ESP32.

The target hardware is an Adafruit HUZZAH32 ESP32 Feather carrying an ESP32-WROOM-32 module with ESP32-D0WDQ6 silicon (revision 1): a dual-core Tensilica Xtensa LX6 at up to 240 MHz with 520 KiB of SRAM, 4 MiB of flash and integrated Wi-Fi and Bluetooth. Firmware uses a custom `huge_app` partition table with a 3 MiB application partition. We selected this ESP32 because it is widely deployed in IoT products and represents a realistic low-cost network node.

A. Training and model selection

First the TON_IoT dataset is loaded and preprocessed as described in Section III. We then trained a broad set of model families so that the on-device comparison would not be limited to a single algorithm class. The candidates were a Decision Tree and a Random Forest, XGBoost as a gradient-boosted baseline, a Feedforward Neural Network, a Support Vector Machine (SVM), and a LightGBM gradient-boosted ensemble. The same 83-feature train/test view of the data is used for every model, which keeps the comparison consistent across model families.

Not every model could be deployed. When converted to portable C with `emlearn` [13], the Random Forest expanded to a roughly 48 MiB source file that exceeded both the device flash and the limits of the ESP32 toolchain, and the XGBoost ensemble likewise did not fit the size and throughput budget of the microcontroller. We therefore retained the four classifiers that ran efficiently on-device while still covering both detection tasks: a Decision Tree and a LightGBM model for multi-class attack detection, and a Feedforward Neural Network and a Support Vector Machine for binary detection. The Decision Tree and LightGBM represent two very different points in

tree-based learning, a single compact tree versus a gradient-boosted ensemble, while the Neural Network and SVM pair a modern TinyML neural workload with a classical kernel-based baseline. The specific hyperparameters used for each deployed model are reported alongside its results in Section V.

B. Model conversion and size

No single conversion library covers all four model families well, so each model is exported with the tool that fits it best. The Decision Tree is converted with `emlearn` [13] using the in-line method, which emits self-contained C99 branch-only code that the Xtensa core can run without any external runtime; the resulting model header is approximately 332 KiB of C source, which compiles to considerably less; Table V provides this number. Exporting LightGBM models as fully self-contained C code is not supported by `emlearn`, so LightGBM is instead converted with `Treelite` and `TL2cgen`, which produce a compact predict function in C. The Feedforward Neural Network is converted with the standard TensorFlow Lite converter into a `.tflite` flatbuffer and executed at runtime by TensorFlow Lite for Microcontrollers [12]. Finally, the Support Vector Machine is converted with `m2cgen`, which emits a stand-alone C function that computes the linear decision directly from the model coefficients.

Memory sizes in this paper are binary: 1 KiB is 1024 bytes. Reporting the size of a converted artifact does not measure what a model costs on the device: generated C source is compiled, not copied, and a flatbuffer needs an interpreter linked alongside it. We therefore measure each model as the difference between its firmware image and that of a reference build that is identical in board, framework, partition table and serial output but contains no model and no test data. The difference isolates the model's contribution to flash, static RAM, peak heap and peak stack, and it is comparable across all four toolchains. Table V reports the result.

Because each model is exported with a different tool, the runtimes reported in Section V reflect the combination of model and generated-code environment rather than the model family in isolation, and are attributed as such.

C. Embedding the test data

For the on-device evaluation to reflect the device itself rather than the serial port that feeds it, the held-out test split must live in flash next to the model. The test data scripts reload the same preprocessed dataset used by the models and emit the features and labels as a C header, with an array containing the raw bytes of the matrix instead of a textual literal of floating-point numbers. This byte-level representation is what allows 7,360 samples of 83 features to fit alongside the model in the ESP32's flash without overflowing the source-code parser or the compiler's RAM during the build, and it keeps the on-device read path a simple cast, so the inference loop's measurement is free of formatting overhead.

Writing the matrix as raw bytes rather than as decimal literals is what makes this practical. The embedded test set occupies $7360 \times 83 \times 4 = 2,443,520$ bytes of feature data, since each feature is a 32-bit float, plus 7,360 bytes of one-byte labels. Emitting the same data as decimal floating-point literals would have produced a source file the toolchain could

not parse within its memory budget, which is the reason for the byte-level representation.

The test data, not the model, is what fills the device: it accounts for 99.1% of the flash the Decision Tree firmware occupies above the reference build. At 333 bytes per sample, the neural network firmware, the largest of the four, barely leaves room for 7,360 samples in total, which is why 7,360 was used for all four models. Embedding the full 36,803-flow test partition would require 11.7 MiB on a 4 MiB device, so complete on-device coverage was not possible in a single build; cross-validation is therefore reported from the host in Section V.

D. Deploying the models and the metrics collected

Each model has its own PlatformIO project, targeting the feather ESP32 board with the Arduino framework on top of the Espressif platform. These projects all use a custom `huge_app` partition table large enough to hold both the converted model and the embedded test data. The Decision Tree, LightGBM and SVM projects include the converted model header and the test data directly. The Neural Network project instead embeds its model into the firmware image and loads it at runtime through the TensorFlow Lite Micro interpreter.

The main file of every project follows the same pattern: it sets the CPU clock to either 240, 160, 80 or 40 MHz, iterates over all embedded samples, calls the model's predict function, and accumulates per-class true-positive, false-positive and false-negative counters. After the loop it computes accuracy and macro precision/recall/F1 for the multi-class models, or binary precision/recall/F1 with a confusion matrix for the binary models, and reports the total runtime together with three efficiency metrics that are collected directly on the device:

- **Inference time per sample** is the average time the model needs to classify a single input flow (one 83-feature vector). It is measured around the model's predict function only: so the figure isolates the cost of the classifier. The remaining overhead of the program falls outside the timed region. The total runtime reported in Section V is time for the complete program and does include these costs; the difference between the two is the overhead for that model. It is reported in microseconds.
- **Inferences per second** is the sustained classification throughput, i.e. the number of samples the model can classify each second; it is the reciprocal of the inference time per sample. It indicates how many network flows the device could screen per second if the IDS ran continuously.
- **Inferences per MHz** is the throughput normalized by the CPU clock frequency (inferences per second divided by the clock in MHz). It is a clock-independent measure of how much classification work a model extracts from each MHz of compute, which makes it possible to compare model efficiency across the four clock settings and to reason about whether a model is limited by raw clock cycles or by memory bandwidth.

- **Memory footprint** is collected from the same run: total and minimum free heap, peak heap use, minimum free stack, and, for the neural network, the TensorFlow Lite Micro tensor arena declared and actually used. Heap and stack figures are watermarks maintained since boot, so they already cover model setup and the whole inference loop. Static allocation is not visible at runtime and is taken from the the build's static allocation summary output instead; both are reported as differences against the reference build described in Section IV-B.

Throughput figures in this paper therefore characterize classifier execution rather than the end-to-end cost of processing a network flow, and should be read as an upper bound on what a complete detection stage could sustain.

The log files used in Section V are direct captures of this serial output.

IV. PERFORMANCE EVALUATION

Across all four clock speeds, classification quality on the embedded test set was essentially independent of CPU frequency, as expected for deterministic inference code.

Results are reported separately for the two detection tasks and are not compared across them. Both are evaluated on the same held-out partition of 36,803 flows, but binary benign/malicious detection is a strictly easier problem than ten-way family attribution, and accuracy figures for the two are not comparable. Table III therefore has a multi-class panel and a binary panel.

Five-fold stratified cross-validation is performed on the host, where every fold scores the complete held-out partition. Training cannot run on the ESP32 itself as the training partition alone is 46.6 MiB against 4 MiB of flash, and the conversion toolchains are inference-only, therefore cross-validation is necessarily host-side; the deployed artifacts remain the models trained on the original split.

Fold-to-fold standard deviations are at most 0.05 percentage points, and for three of the four models the cross-validated accuracy lies inside the 95% Wilson interval of the on-device measurement, so detection quality is stable across splits. The cross-validated figures are the better estimate of detection quality, and the on-device figures provide the evidence that the converted models run correctly.

A. Decision Tree – Multi-class

The Decision Tree was trained with scikit-learn (`DecisionTreeClassifier`, fixed random seed) using the Gini impurity criterion and grown to full depth, with the default `min_samples_split = 2` and `min_samples_leaf = 1`. The tree was allowed to grow unrestricted because the single-tree inference cost is only $O(\text{depth})$ and the converted branch-only code remains modest in flash, so full growth maximizes multi-class accuracy at almost no runtime cost. It was converted with emlearn using the inline method and float data type to match the device test data, and is the fastest of the four models by a wide margin, and among the two that perform ten-class attribution, it is also the smaller. Throughput stayed relatively constant

until the clock dropped below 80 MHz, and from there scaled almost linearly with frequency; even the slowest configuration processed all 7,360 samples in 310 ms.

The efficiency metric rises as the clock is lowered (Table IV), which tells us where the time goes. If the model were limited purely by how fast the processor executes instructions, the work done per MHz would stay the same at every clock speed, because halving the clock would simply halve the throughput. Instead the Decision Tree gets more done per MHz at 80 and 40 MHz than at 240 MHz, so part of each inference is spent waiting on something that does not speed up with the core clock. The converted tree is a block of branch-only code read from flash, and flash accesses run at their own fixed rate: at 240 MHz the processor spends much of its time waiting for those reads, while at 40 MHz the same waiting covers a larger share of useful work. This is why throughput falls by only a factor of 2.6 when the clock is reduced by a factor of six.

B. LightGBM – Multi-class

LightGBM was trained as a deliberately small gradient-boosted ensemble so that it would fit and run on the ESP32: a multi-class objective over ten classes with only 12 estimators, a maximum depth of 4, 10 leaves, a learning rate of 0.15, `min_child_samples` = 20 and an L2 regularization term (`reg_lambda` = 1.0), with early stopping after 15 rounds without improvement. The shallow, short ensemble and the comparatively high learning rate were chosen together: limiting the model to 12 boosting rounds of depth-4 trees keeps the converted C model small and the per-sample work low, and the learning rate was raised to 0.15 to recover accuracy that would otherwise be lost from the small number of rounds, while `reg_lambda` = 1.0 guards against overfitting the reduced ensemble. The model was converted with Treelite and TL2cgen. Even with this reduced configuration it is by far the slowest of the four: each inference takes between 1,010 and 3,112 μ s, two orders of magnitude longer than the Decision Tree. Throughput drops from 990.3 classifications per second at 240 MHz to 321.3 at 40 MHz, where evaluating all 7,360 samples takes 24.4 s. Classifications per MHz are correspondingly low (4–8), confirming that even a compact gradient-boosted ensemble does not fit the throughput budget of this microcontroller as well as a single tree does (Table IV).

C. Feedforward Neural Network – Binary

The Feedforward Neural Network is a compact, fully connected model: an 83-dimensional input, a single hidden layer of 16 ReLU units with 0.5 dropout, and a single sigmoid output. It was trained with the Adam optimizer and binary cross-entropy loss for up to 100 epochs, with early stopping (patience 5, restoring the best weights on the validation PR-AUC) and a ReduceLROnPlateau schedule (factor 0.1, minimum learning rate 10^{-3}); features were standardized, and the output-layer bias was initialized to $\log(\text{normal}/\text{intrusion})$ to counter the class imbalance. This single 16-unit hidden layer keeps the parameter count near 1,361, which is what allows the quantized model to occupy only 7.0 KiB as a flatbuffer, though executing it costs far more than that once the interpreter is included (Table V), while dropout and early stopping control overfitting. The model was quantized and exported with TensorFlow Lite for Microcontrollers. It misclassified only 6 of 7,360 samples

(3 false positives and 3 false negatives), giving an accuracy of 99.92% and a precision, recall and F1 of 0.9995. Inference time scales from 163.71 μ s at 240 MHz to 719.46 μ s at 40 MHz, with throughput dropping from 6,108 to 1,390 classifications per second. Efficiency improves as the clock is reduced, from 25.45 to 34.75 classifications per MHz, following the same pattern as the Decision Tree but less strongly: part of the work is waiting on memory rather than on the core clock. The confusion matrix is given in Table VII.

The neural network is the only deployment in this study with a substantial RAM cost. It adds 8,040 B of static RAM, of which 2 KiB is the TensorFlow Lite Micro tensor arena and 5,992 B the interpreter's own state; the arena is sized to the 1,040 B the interpreter reports as required. Its 237 KiB of flash is likewise dominated by the interpreter rather than by the 7.0 KiB model it executes.

D. Support Vector Machine – Binary

The Support Vector Machine is a linear classifier (scikit-learn `LinearSVC`) trained on standardized features with the default regularization (C = 1.0), a squared-hinge loss and a raised iteration cap (`max_iter` = 20,000) to ensure convergence on the high-dimensional scaled data. A linear kernel was chosen deliberately: it reduces on-device inference to a single dot product over 83 weights, which is what makes it the smallest deployment measured, at 4.8 KiB of flash, and the fastest binary classifier per cycle. It was converted with m2cgen and represents a classical kernel-based baseline. It misclassified 8 of 7,360 samples (4 false positives and 4 false negatives), reaching 99.89% accuracy and a precision, recall and F1 of 0.9993. The Support Vector Machine sits between the Decision Tree and the Neural Network on throughput: 16,615 classifications per second at 240 MHz down to 2,744 at 40 MHz, with efficiency stable at 68–70 per MHz across all clocks (Table IV). This is roughly three times faster than the Neural Network at every clock speed for binary detection, at the cost of a slightly higher error count (8 versus 6 misclassifications out of 7,360). The confusion matrix is given in Table VII.

Unlike the other three models the Support Vector Machine holds a nearly constant efficiency across all four clock settings. Its inference time is spent executing instructions rather than waiting for memory, and throughput therefore scales almost exactly with the clock. The contrast with the Decision Tree shows the effect: the same reduction in clock costs a flash-resident tree far less throughput than it costs a compute-bound linear model.

E. Scope and limitations

Three limits bound these results. First, only the classification stage runs on the device: packet capture, flow reconstruction, feature computation, encoding and scaling are performed offline, so the throughput figures are not network-flow processing rates. Second, all results come from a single dataset and a single 80/20 split; the cross-validation in Table VI varies the split but not the dataset, and cross-dataset validation, temporal and device-independent splits, unseen-attack analysis and concept drift are left to future work. The headroom reported in Section VI is only for compute and memory headroom.

TABLE III. SUMMARY OF ESP32 RESULTS AT 240 MHz AND 40 MHz ON 7,360 EMBEDDED TEST SAMPLES (83 FEATURES). THE TWO TASKS ARE REPORTED IN SEPARATE PANELS AND ARE NOT COMPARABLE. MODEL FLASH IS THE MEASURED CONTRIBUTION TO THE FIRMWARE IMAGE (TABLE V), NOT THE SIZE OF THE GENERATED SOURCE.

Model	Accuracy	Macro F1	Class./s @240 MHz	Class./s @40 MHz	Model flash
<i>Ten-class attack attribution</i>					
Decision Tree	99.14%	0.9813	68,050	26,287	21.9 KiB
LightGBM	98.86%	0.9665	990	321	51.1 KiB
<i>Binary benign/malicious detection</i>					
Feedforward Neural Net.	99.92%	0.9995 [†]	6,108	1,390	237.0 KiB
Support Vector Machine	99.89%	0.9993 [†]	16,614	2,744	4.8 KiB
[†] binary F1, not macro F1.					

TABLE IV. PER-MODEL ESP32 RESULTS ACROSS THE FOUR CPU CLOCKS ON 7,360 EMBEDDED TEST SAMPLES. CLASSIFICATION METRICS ARE IDENTICAL AT EVERY CLOCK FOR EVERY MODEL AND ARE LISTED ONCE. TIME IS PER SAMPLE, IN MICROSECONDS.

Model	Metric	240 MHz	160 MHz	80 MHz	40 MHz
Decision Tree (10-class)	Runtime (ms)	114	123	151	310
	Time/sample	14.69	15.66	18.56	38.04
	Class./s	68,050	63,865	53,869	26,288
	Class./MHz	283.5	399.2	673.4	657.2
	Acc. 99.14%, macro P 0.9796, R 0.9831, F1 0.9813				
LightGBM (10-class)	Runtime (ms)	7,834	8,841	11,911	24,391
	Time/sample	1,009.81	1,135.66	1,519.43	3,112.12
	Class./s	990.3	880.5	658.1	321.3
	Class./MHz	4.13	5.50	8.23	8.03
	Acc. 98.86%, macro P 0.9559, R 0.9820, F1 0.9665				
Neural Net (binary)	Runtime (ms)	1,400	1,757	2,851	5,844
	Time/sample	163.71	209.93	350.99	719.46
	Class./s	6,108	4,764	2,849	1,390
	Class./MHz	25.45	29.77	35.61	34.75
	Acc. 99.92%, P/R/F1 0.9995; 3 FP, 3 FN of 7,360				
SVM (binary)	Runtime (ms)	757	1,051	1,946	3,986
	Time/sample	60.19	89.30	177.98	364.50
	Class./s	16,615	11,199	5,619	2,744
	Class./MHz	69.23	69.99	70.23	68.59
	Acc. 99.89%, P/R/F1 0.9993; 4 FP, 4 FN of 7,360				

TABLE V. MEASURED MEMORY COST OF EACH DEPLOYMENT, AS THE DIFFERENCE BETWEEN ITS FIRMWARE IMAGE AND THE REFERENCE BUILD (21,464 B STATIC RAM, 270,345 B FLASH, 31,048 B PEAK HEAP). “GENERATED FILE” IS THE ARTIFACT SIZE THE SIZE COLUMN OF EARLIER WORK REPORTS.

Model	Generated file	Flash (measured)	Static RAM	Peak heap	Peak stack
SVM (binary)	2.9 KiB	4.8 KiB	664 B	252 B	16 B
Decision Tree (10c)	332 KiB	21.9 KiB	0 B	372 B	32 B
LightGBM (10c)	173 KiB	51.1 KiB	924 B	256 B	64 B
Neural Net (binary)	7.0 KiB	237.0 KiB	8,040 B	416 B	32 B

TABLE VI. FIVE-FOLD STRATIFIED CROSS-VALIDATION ON THE HOST, MEAN $\pm$ STANDARD DEVIATION OVER FOLDS; EACH FOLD SCORES ROUGHLY 36,800 HELD-OUT FLOWS. HYPERPARAMETERS ARE IDENTICAL TO THE DEPLOYED MODELS, AND SCALERS ARE FITTED PER FOLD ON TRAINING DATA ONLY. F1 IS MACRO-AVERAGED FOR THE TEN-CLASS MODELS AND BINARY FOR THE OTHERS.

Model	Task	Accuracy	F1
Decision Tree	Multi-class (10)	99.36% $\pm$ 0.05	0.9823 $\pm$ 0.0012
LightGBM	Multi-class (10)	98.83% $\pm$ 0.03	0.9750 $\pm$ 0.0013
Feedforward Neural Net.	Binary	99.93% $\pm$ 0.04	0.9996 $\pm$ 0.0003
Support Vector Machine	Binary	99.94% $\pm$ 0.01	0.9996 $\pm$ 0.0000

V. DISCUSSION

A direct numerical comparison with prior work is only meaningful for studies that use the same TON_IoT Train-

TABLE VII. BINARY CONFUSION MATRICES ON THE 7,360 EMBEDDED TEST SAMPLES. POSITIVE DENOTES BENIGN TRAFFIC.

Model	Predicted	True label	
		Positive	Negative
Neural Network	Positive	5,729	3
	Negative	3	1,625
Support Vector Machine	Positive	5,728	4
	Negative	4	1,624

TABLE VIII. MULTI-CLASS COMPARISON ON THE TON_IoT TRAIN-TEST NETWORK DATASET. RPI 4 DENOTES A RASPBERRY PI 4.

Source	Model	Acc.	Macro F1
This paper (ESP32)	Decision Tree	99.14%	0.9813
This paper (ESP32)	LightGBM	98.86%	0.9665
[5] (RPI 4)	Decision Tree	98.7%	0.987
[5] (RPI 4)	LightGBM	97.6%	0.976
[5] (RPI 4)	FNN	97.4%	0.976
[5] (RPI 4)	TinyML FNN	97.4%	0.976
[4]	Random Forest	98.1%	0.973
[5]	Voting	96.3%	0.886

Test Network subset and report multi-class metrics, since our binary models cannot be aligned one-to-one with multi-class baselines. Multi-class numbers for the same four model families on a Raspberry Pi 4, together with a Voting ensemble baseline, are reported in [5], and a Random Forest trained on the same dataset in [4]. Table VIII compares our ESP32 multi-class results to these published values.

The models measured on the ESP32 attain higher accuracy but slightly lower macro F1 than the corresponding models in [5] on a throttled Raspberry Pi 4: 99.14% and 0.9813 against 98.7% and 0.987 for the Decision Tree, and 98.86% and 0.9665 against 97.6% and 0.976 for LightGBM.

Two observations stand out. First, moving from a throttled Linux host running CPython to a 240 MHz Xtensa microcontroller with 520 KiB of SRAM changed detection quality by fractions of a percentage point, on hardware costing roughly an order of magnitude less. Our Decision Tree differs from theirs by less than half a percentage point in accuracy and a few thousandths in macro F1, in opposite directions. Given the differences in runtime, memory and instruction set, parity is itself the non-trivial result. Second, our Decision Tree exceeds the Random Forest baseline of [4] and the Voting ensemble of [5] on macro F1. Combined with the binary results, where both the Neural Network and the Support Vector Machine exceed 99.89% accuracy on-device, this shows that the ESP32 can host competitive ten-class *and* binary IDS inference, which

is the gap left open by the related work discussed in Section I.

Within each task the choice is clear-cut, and it is settled by cost rather than by accuracy. For ten-class attribution the Decision Tree dominates LightGBM on every axis measured: higher accuracy, 2.3 times smaller in flash and 69 times faster. For binary detection the two candidates are close in accuracy, and the Support Vector Machine is three times faster while occupying 4.8 KiB against the neural network's 237 KiB. Size does not predict speed, however: the Decision Tree is 4.6 times larger than the Support Vector Machine and classifies four times faster.

The measurements also quantify the headroom left for other applications on the device. Excluding the embedded test harness, a deployed detector occupies between 275,277 and 513,005 bytes of the 3 MiB application partition and between 21,464 and 29,504 bytes of the 320 KiB of DRAM, that is 8.8–16.3% of flash and 6.6–9.0% of RAM. Between 84 and 91% of flash and 91 and 93% of RAM therefore remain for the device's primary application. Because all four models retain their accuracy at 40 MHz, detection can additionally run well below the peak clock. No power measurement was performed, so this should be done as future work.

VI. CONCLUSION

In this study, we deployed a series of lightweight machine-learning-based IDS classifiers directly on a resource-constrained, low-cost ESP32 microcontroller. From a broad set of candidate model families, we identified the four that fit the device (a Decision Tree and LightGBM for ten-class attribution and a Feedforward Neural Network and a Support Vector Machine for binary detection) and reported their on-device memory cost, detection quality and efficiency. Using a dataset (TON_IoT) designed to reflect a realistic IoT network testbed, our models detected and identified security threats at four CPU clock frequencies with minimal operational impact on the device itself.

The models deployed onto the microcontroller demonstrate that there is headroom for other applications, due to measurements on RAM and flash usage: the Decision Tree costs 21.9 KiB of flash and no static RAM at all, and the linear Support Vector Machine 4.8 KiB and 664 bytes, whereas executing a 7.0 KiB neural network requires 237 KiB of flash and 7.9 KiB of RAM for the interpreter that runs it. A deployed detector leaves 84–91% of flash and 91–93% of RAM free for the device's primary application.

While prior work relied on throttled host measurements, this paper instead presents empirical evidence from real hardware, establishing a practical alternative to offloading intrusion detection to the cloud and showing that an ESP32 can run IDS inference at a frequency well below its peak clock. Future work includes integrating live packet capture and feature extraction on the device, evaluating energy consumption directly with power instrumentation, validating across additional datasets and against unseen attack families, and extending the on-device evaluation to other microcontroller families.

ACKNOWLEDGMENT

The authors thank the Faculty of Computer Science, Engineering and Economics at Østfold University of Applied

Sciences (OUAS) for supporting this research.

REFERENCES

- [1] J. Hojlo, "Future of industry ecosystems: Shared data and insights," International Data Corporation (IDC), 2021, <https://blogs.idc.com/2021/01/06/future-of-industry-ecosystems-shared-data-and-insights/>.
- [2] N. Moustafa, "A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets," *Sustainable Cities and Society*, vol. 72, p. 102994, 2021.
- [3] N. Moustafa, M. Keshky, E. Debiez, and H. Janicke, "Federated TON_IoT Windows datasets for evaluating AI-based security applications," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 848–855.
- [4] T. M. Booi, I. Chiscop, E. Meeuwissen, N. Moustafa, and F. T. H. den Hartog, "ToN_IoT: The role of heterogeneity and the need for standardization of features and attack types in IoT network intrusion data sets," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 485–496, 2022.
- [5] J. Lundqvist, A. Hadzic, T. M. Kirkeluten, H. Pedersen, J. Holth, M. H. Johansson, and M. P. N. Halkjelsvik, "Lightweight machine learning models for intrusion detection on IoT devices," in *Norsk IKT-konferanse for forskning og utdanning (NIKT)*, 2025.
- [6] A. Thakkar and R. Lohiya, "A review on machine learning and deep learning perspectives of IDS for IoT: Recent updates, security issues, and challenges," *Archives of Computational Methods in Engineering*, vol. 28, no. 4, pp. 3211–3243, 2021.
- [7] A. Verma and V. Ranga, "Machine learning based intrusion detection systems for IoT applications," *Wireless Personal Communications*, vol. 111, no. 4, pp. 2287–2310, 2020.
- [8] A. Amouri, V. T. Alapathy, and S. D. Morgera, "A machine learning based intrusion detection system for mobile Internet of Things," *Sensors*, vol. 20, no. 2, p. 461, 2020.
- [9] A. Javed, A. Ehtsham, M. Jawad, M. N. Awais, A.-u.-H. Qureshi, and H. Larijani, "Implementation of lightweight machine learning-based intrusion detection system on IoT devices of smart homes," *Future Internet*, vol. 16, no. 6, p. 200, 2024.
- [10] N. N. Alajlan and D. M. Ibrahim, "TinyML: Enabling of inference deep learning models on ultra-low-power IoT edge devices for AI applications," *Micromachines*, vol. 13, no. 6, p. 851, 2022.
- [11] D. L. Dutta and S. Bharali, "TinyML meets IoT: A comprehensive survey," *Internet of Things*, vol. 16, p. 100461, 2021.
- [12] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019.
- [13] J. Nordby, "emlearn: Machine learning inference engine for microcontrollers and embedded devices," 2019.