JURNAL INFOTEL
VOL. 18, NO. 3, AUGUST 2026, PP. 490–509. DOI:10.20895/INFOTEL.V18I3.10763

RESEARCH ARTICLE

Benchmarking Classical And Neural Machine Learning for Intrusion Detection on Resource-Constrained IoT Platforms

Nur Dwi Priyambodo[1,*] and Edi Sugiarto[2]

[1,2]Department of Informatics Engineering, Universitas Dian Nuswantoro, Semarang 50131, Indonesia

*Corresponding email: 122202303004@mhs.dinus.ac.id

Received: March 01, 2026; Revised: August 11, 2026; Accepted: August 16, 2026.

Abstract: Intrusion detection system (IDS) research commonly evaluates models on server-grade hardware, overlooking whether these models actually run on the low-cost microcontrollers deployed at the network edge. This study benchmarks five machine learning models–Decision Tree (DT), Random Forest (RF), k-Nearest Neighbors (KNN), Deep Neural Network (DNN), and 1D Convolutional Neural Network (CNN)—for binary intrusion detection across three platforms: ESP32, Arduino Uno, and Arduino Nano. Classical models are exported through MicroMLGen, while neural models are deployed via TensorFlow Lite for Microcontrollers with INT8 quantization on ESP32 and through a framework-free float32 PROGMEM approach on 8-bit AVR. Evaluation uses NSL-KDD and Edge-IIoTset v2 under a stratified balanced protocol with multiple random seeds. On NSL-KDD, DT obtains the best trade-off (F_1=0.829, 2.65 μs latency on ESP32), outperforming all neural variants. The INT8-quantized DNN deployed on ESP32 shows severe precision-recall imbalance (F_1=0.637), whereas the same DNN deployed in float32 on AVR reaches F_1=0.762; since host-side INT8 evaluation is lossless and the AVR float32 deployment reproduces host predictions exactly, the on-device gap is a deployment-specific effect of the INT8 TFLite Micro path on ESP32, not of post-training quantization. CNN runs only on ESP32 (F_1=0.720). Uniform perfect scores on Edge-IIoTset v2 serve as deployment-context evidence only; the findings above are limited to NSL-KDD. These results challenge the widespread assumption that neural networks automatically surpass classical methods for embedded IDS, and offer practitioners evidence-based guidance.

Keywords: intrusion detection system, IoT security, microcontroller, model quantization, TinyML

© Jurnal Infotel ISSN: 2085-3688; e-ISSN: 2460-0997

1 Introduction

The explosive growth of Internet of Things (IoT) devices has substantially widened the attack surface of networked systems [1, 2]. High-profile incidents such as the Mirai botnet, which enlisted hundreds of thousands of poorly secured devices into a coordinated denial-of-service army, illustrate how vulnerable low-cost endpoints can become weapons when left unprotected [3]. Traditional intrusion detection systems assume abundant compute and memory resources that microcontrollers simply do not possess [4–8]. There is, therefore, a concrete need to evaluate IDS models not only by their predictive accuracy on a desktop machine but also by their actual deployment feasibility, covering inference latency, memory footprint, and platform-specific limitations.

Several studies have explored deep learning approaches to IoT intrusion detection [9–12]. Although these methods achieve high accuracy on desktop hardware, they frequently exceed the memory and compute budget of microcontrollers [13, 14]. This observation has motivated research into lightweight classical machine learning [15, 16], TinyML paradigms [17–19], and recent evaluations of TinyML-based IDS on resource-constrained devices [20]. Despite these advances, three important gaps remain in the existing literature.

First, most IDS evaluations report only offline accuracy and never verify whether the model works on actual embedded hardware, a disconnect between published metrics and practical deployment [21]. Second, cross-architecture comparisons that span different microcontroller families (for instance, 32-bit ESP32 versus 8-bit AVR) are scarce, leaving practitioners without hardware-specific guidance [19]. Third, to the best of our knowledge, no prior study has directly compared classical machine learning exported through MicroMLGen [22] against quantized neural networks deployed via TensorFlow Lite for Microcontrollers [23] on the same hardware, with the same datasets, and under identical benchmarking conditions.

This work addresses these gaps by benchmarking five models on ESP32, Arduino Uno, and Arduino Nano: Decision Tree (DT), Random Forest (RF), k-Nearest Neighbors (KNN), Deep Neural Network (DNN), and 1D Convolutional Neural Network (CNN). We formulate four research questions: (RQ1) Which model achieves the best detection-latency trade-off, and do neural models outperform classical alternatives on-device? (RQ2) How does dataset choice, legacy NSL-KDD versus modern Edge-IIoTset v2, affect benchmarking conclusions? (RQ3) What platform-specific constraints determine the feasibility for each model? (RQ4) How does INT8 post-training quantization affect neural IDS on tabular data?

The main contributions are: (1) a cross-paradigm benchmark comparing classical and neural models on identical embedded platforms using a unified pipeline; (2) a dual-dataset evaluation on NSL-KDD [24] and Edge-IIoTset v2 [25] with explicit leakage analysis; (3) a framework-free PROGMEM inference methodology for deploying neural networks on 8-bit AVR, where TFLite Micro is incompatible; and (4) quantified evidence that INT8 post-training quantization is lossless on the host and that the observed on-device degradation of the DNN path is a deployment-specific effect of the TFLite Micro runtime, not of the quantization scheme.

2 Research Method

2.1 Datasets and preprocessing

Two dataset families are used. NSL-KDD [24] is a widely used benchmark derived from the KDD Cup 1999 dataset. Although it predates IoT-era traffic and has known limitations [26, 27], it enables comparison with prior IDS literature and its moderate difficulty realistically stresses models. Edge-IIoTset v2 [25, 28] is a modern IoT/IIoT dataset (2022) containing 14 attack categories from a realistic testbed. We use a Full variant (top-10 mutual information features) and a Sanitized variant that removes four potentially leaky features following data-leakage mitigation practices [29]. All labels are harmonized into binary form (Benign versus Attack) to align with practical "alert or not" decisions.

For on-device benchmarking, 1,000 test samples are drawn per dataset using a balanced protocol (500 Benign, 500 Attack), repeated across ten random seeds for classical models and three seeds for neural models. All $\pm$ values are inter-seed standard deviations. The 95% CI of the mean F_1 of DT on the 10-seed authoritative set is 0.829 ± 0.007; the binomial sampling CI of accuracy for a single $N{=}1{,}000$ balanced subset is ± 0.023, wider than inter-model gaps considered here. The balanced subset serves experimental control on-device: the fixed 500/500 composition isolates classifier behavior from base-rate effects, while evaluation under the natural distribution is reported host-side.

All datasets are split 80/20 for training and testing using stratified random sampling. Ten random seeds (42–51) are used for classical model evaluation and three seeds (42–44) for neural models; the reduced seed count for neural models reflects the higher computational cost of re-training and deploying DNN/CNN firmware. Each seed controls both the stratified train/test split and the balanced test-subset sampling, so that inter-seed variance captures the combined effect of training partition and evaluation sampling. Features are selected by mutual information (top-10) and normalized with MinMaxScaler fitted on the training set. Scaler parameters are embedded directly in device firmware so that raw feature values are normalized exactly once on-device.

2.2 Model selection and training

Five models spanning two paradigms are evaluated. The classical set (DT, RF, and KNN) represents rule-based, ensemble, and instance-based learning, respectively. These are the model types supported by MicroMLGen [22], the only open-source tool that exports scikit-learn classifiers to portable C/C++ for microcontrollers. The neural set (DNN and CNN) is included to bridge the gap between classical and deep learning approaches for embedded IDS.

DT and RF hyperparameters are selected in two stages: an unconstrained grid search via 5-fold cross-validation, followed by hardware-aware pruning to the smallest model within 2% F_1 of the optimum that fits within Arduino's 32 KB flash. For NSL-KDD, the DT uses `max_depth` $= 5$ (61 nodes); for Edge-IIoTset, `max_depth` $= 3$ already achieves perfect cross-validation F_1. RF uses 3–5 trees to control flash usage. KNN stores a stratified subsample of 5,000 out of approximately 100,000 training points (200 KB), which is the maximum that fits in ESP32 flash; on Arduino (32 KB flash), KNN is infeasible regardless of subsample size. KNN was included in the same unconstrained grid search with 5-fold cross-validation applied to DT and RF, with $n_{\text{neighbors}}$ in the range of 3–7 (final values: $k{=}7$

on NSL-KDD and k=3 on Edge-IIoTset, Table 1), distance metric, and weighting scheme as grid dimensions, and its results are evaluated on the same ten seeds (42–51) as the other classical models. Because the 200 KB reference table exceeds the 32 KB flash of the 8-bit AVR boards, KNN is deployed on the ESP32 only.

The DNN architecture is Input(10) → Dense(32, ReLU) → BatchNorm → Dropout(0.3) → Dense(16, ReLU) → BatchNorm → Dropout(0.2) → Dense(8, ReLU) → Dense(1, Sigmoid), totaling approximately 1,100 parameters. The CNN architecture is Input(10,1) → Conv1D(16, kernel=3) → MaxPool1D(2) → Conv1D(8, kernel=3) → GlobalAvgPool → Dense(8) → Dropout(0.3) → Dense(1, Sigmoid), with approximately 700 parameters. Both are trained with Adam (lr=10^{-3}), binary cross-entropy loss, batch size 64, and early stopping (patience=10). Training converges within 20–35 epochs.

Why a 1D CNN on ten tabular features. The neural arm of this benchmark intentionally covers two architectures that are representative of TinyML practice. The CNN is included not because convolution is expected to excel on ten independent scalar features, but because (i) 1D ConvNet blocks (Conv1D + pooling) are the standard architecture exported by the TinyML toolchains we evaluate and appear routinely in embedded IDS literature, and (ii) it tests whether the deployment pipeline behaves differently for convolution kernels than for dense layers, which proved decisive in isolating the ESP32 DNN anomaly (section 4). We acknowledge that on ten independently scaled features the CNN has no structural advantage over the DNN, and indeed its on-host F_1 (0.728) is lower than the DNN (0.762). The comparison therefore answers an engineering question—which architecture, as deployed, is more robust on the tested hardware—rather than claiming CNN superiority on tabular data. Its compact size (633 parameters) also stresses the quantization sensitivity of very small models. Future work may consider attention or tabular-specific learners, but such models are not yet supported by the microcontroller deployment path used here. Table 1 presents the final hyperparameters for the classical models.

Table 1: Classical model hyperparameters

Model	Parameter	NSL-KDD	Edge-IIoT
DT	criterion	entropy	gini
	max_depth†	5	3
	min_samples_leaf	5	5
RF	n_estimators†	3	5
	criterion	entropy	gini
	max_depth†	5	3
	min_samples_leaf	5	5
KNN	n_neighbors	7	3
	metric	Manhattan	Minkowski
	weights	uniform	uniform

†Hardware-constrained

2.3 Embedded deployment

Classical models are exported to C/C++ headers via MicroMLGen and compiled into firmware. On ESP32, neural models are deployed using TensorFlow Lite for Microcon-

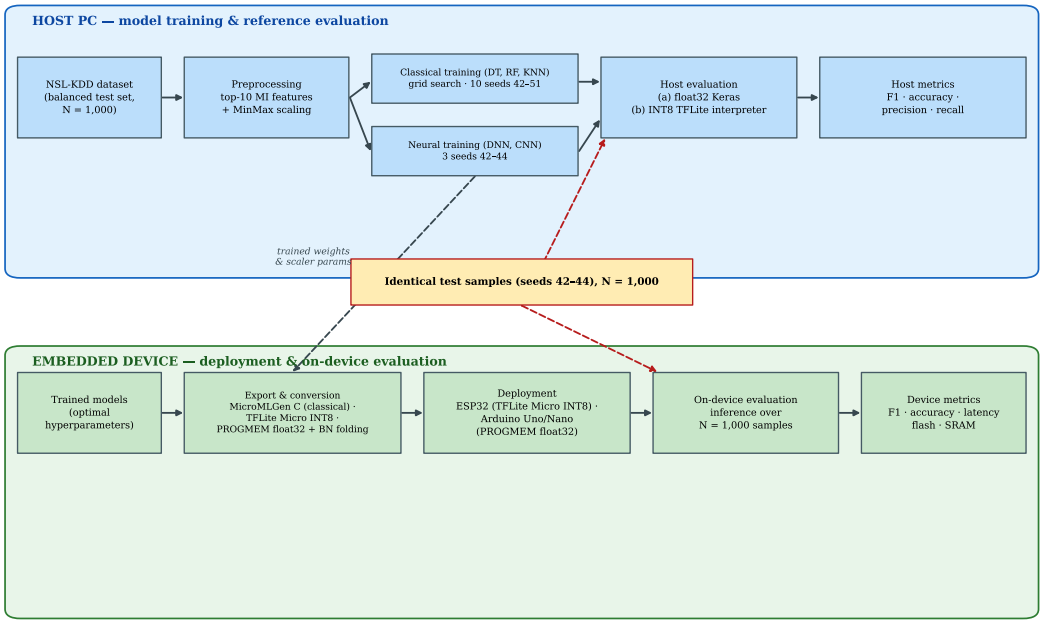


Figure 1: Two-lane evaluation pipeline: host-side (float32 Keras and INT8 TFLite) and on-device (ESP32 with INT8 TFLite Micro and 8-bit AVR with float32 PROGMEM inference), sharing identical test samples (seeds 42–44, N=1,000).

trollers [23] with INT8 post-training quantization. Representative calibration draws 1,000 training samples (MinMax-scaled, $[0, 1]$) with a fixed seed for reproducibility.

To isolate quantization effects from deployment effects, the INT8 models were also evaluated on the host with the TensorFlow Lite interpreter using the exact model files embedded in the firmware and the exact test samples transmitted to the devices. All three pipelines in Table 4–(a) float32 Keras on the host, (b) INT8 TFLite on the host, and (c) on-device firmware (INT8 on ESP32; float32 PROGMEM with batch-normalization folding on AVR)–share the same checkpoints and test samples; "±" denotes inter-seed standard deviation over seeds 42–44. Figure 1 illustrates this two-lane pipeline: host-side evaluation (float32 Keras and INT8 TFLite) and on-device evaluation (ESP32 with INT8 TFLite Micro, AVR with float32 PROGMEM), with both lanes consuming the identical checkpoints and test samples.

A key finding of this work is that TensorFlow Lite for Microcontrollers is incompatible with 8-bit AVR (ATmega328P): the runtime requires 32-bit addressing and ARM-specific kernels that are unavailable on the AVR architecture. To overcome this, we developed a framework-free manual inference approach: weights are extracted from the trained Keras model, BatchNormalization layers are algebraically folded into adjacent Dense layers, and all parameters are stored as `const float PROGMEM` arrays in flash. A minimal C forward pass reads weights via `pgm_read_float()` using two alternating RAM buffers totaling

approximately 200 bytes. This approach fits within the 2 KB SRAM and 32 KB flash of ATmega328P and produces predictions identical to the original Keras model.

The BatchNormalization folding is mathematically expressed as follows. Given BN parameters γ, β, μ, σ^2, we define $s = \gamma/\sqrt{\sigma^2 + \epsilon}$ and $t = \beta - s \cdot \mu$. The subsequent Dense layer's weights and biases are transformed:

$$W_{\text{folded}} = W_{\text{next}} \cdot \text{diag}(s), \quad b_{\text{folded}} = W_{\text{next}} \cdot t + b_{\text{next}} \tag{1}$$

This eliminates BN at inference time while preserving algebraic equivalence, verified to below 10^{-6} absolute difference on 1,000 test samples.

2.4 Benchmarking protocol

Test samples are transmitted from a host PC via serial connection (115,200 baud). For each sample, the device receives raw features, normalizes them, runs inference, and reports prediction and timing. Latency is measured using `micros()` on Arduino (4 μs resolution) and `esp_timer_get_time()` on ESP32 (1 μs resolution). The large sample count averages out timer quantization effects. Numerical verification confirms 100% prediction agreement between Python and on-device code for DT and RF; DNN float32 PROGMEM inference matches the Python baseline, while the on-device INT8 predictions differ from the host interpreter in a deployment-specific manner isolated to the TFLite Micro runtime (section 4).

Firmware footprints were recorded from the `arduino-cli compile` outputs (arduino-cli 1.5.1). The combined DNN/CNN ESP32 firmware (240 MHz, `-Os`, esp32 core 2.0.17, TensorFlowLite_ESP32 1.0.0) uses 540,829 B of flash (41%) and 68,596 B of SRAM (20%), with TFLite Micro interpreter arenas of 964 B (DNN) and 2,772 B (CNN). The PROGMEM DNN sketch on AVR (16 MHz, `-Os -flto`, avr core 1.8.8, avr-gcc 7.3.0) uses 12,926 B of flash (42%) and 380 B of SRAM (18%) on the Nano and 12,924 B (40%) and 380 B on the Uno. Inference latency was timed around the inference call only—excluding serial I/O and normalization—with N=1,000 samples per seed on ESP32 and N=4,000 on AVR, without warm-up. Table 2 summarizes the three target platforms.

Table 2: Target platform specifications

Spec.	ESP32	Uno	Nano
Architecture	32-bit Xtensa	8-bit AVR	8-bit AVR
Clock (MHz)	240	16	16
SRAM (KB)	520	2	2
Flash (KB)	4,096	32	32
FPU	Hardware	Software	Software
Classical	DT, RF, KNN	DT, RF	DT, RF
Neural	DNN, CNN	DNN only	DNN only

3 Results

3.1 Classification performance

Table 3 presents the NSL-KDD results across all five models. Contrary to expectations, DT achieves the highest F_1 of 0.829 ± 0.010, surpassing every neural variant by a substantial margin. The INT8-quantized DNN on ESP32 exhibits extreme precision-recall imbalance: precision reaches 0.990 but recall drops to only 0.470, yielding F_1=0.637. CNN on ESP32 provides somewhat better balance (F_1=0.720) but still falls short of classical models. Importantly, the same DNN deployed via float32 PROGMEM on AVR achieves F_1=0.762—markedly higher than its INT8 counterpart on ESP32. This gap, however, is deployment-specific rather than a property of INT8 quantization: on the host, the INT8 model is statistically indistinguishable from its float32 baseline (F_1 0.7634 vs. 0.7624, p>0.1, paired over seeds), and the on-device degradation is isolated to the ESP32 TFLite Micro runtime (section 4, Table 4).

Table 3: NSL-KDD results (balanced, N=1,000, mean ± std)

Plat.	Model	Acc.	Prec.	Rec.	F_1
ESP32	DT	0.840±0.009	0.888±0.009	0.778±0.014	0.829±0.010
	RF	0.804±0.009	0.958±0.010	0.637±0.015	0.765±0.013
	KNN	0.843±0.014	0.957±0.009	0.718±0.023	0.821±0.017
	DNN	0.733±0.011	0.990±0.006	0.470±0.024	0.637±0.021
	CNN	0.775±0.008	0.956±0.012	0.577±0.009	0.720±0.010
Uno	DT	0.840±0.009	0.888±0.009	0.778±0.014	0.829±0.010
	RF	0.804±0.009	0.958±0.010	0.637±0.015	0.765±0.013
	DNN	0.803±0.009	0.958±0.007	0.633±0.014	0.762±0.012
Nano	DT	0.840±0.009	0.888±0.009	0.778±0.014	0.829±0.010
	RF	0.804±0.009	0.958±0.010	0.637±0.015	0.765±0.013
	DNN	0.803±0.009	0.958±0.007	0.633±0.014	0.762±0.012

For Edge-IIoTset v2, every model achieves 100.0% accuracy and F_1=1.000 on both Full and Sanitized variants across all platforms. Four leakage controls were applied to the evaluation protocol, and all passed: (i) the three attack types never seen during training (Backdoor, Ransomware, Uploading) are still detected at F_1=1.000; (ii) permutation tests with 1,000 label shuffles reject chance-level performance (p=0.0); (iii) the Sanitized variant, which removes protocol-specific features, reaches the same F_1=1.000; and (iv) deduplication of the 48 overlapping rows changes nothing. These controls show no indication of leakage, so F_1=1.000 is reported as a result of this limited evaluation on the sanitized split—deployment-feasibility evidence, not a claim of generalization. Two honest caveats qualify even this restricted reading. First, per-attack-type evaluation gives F_1=0 on the Normal class for DT, RF, and KNN: because every feature correlates with attack density in the controlled testbed, the models always predict "Attack", driving Normal precision and recall to zero—a consequence of the imbalanced composition of the per-type test split rather than of individual model failure. Second, the top mutual information features (`dns.qry.name.len` and `mqtt.conack.flags`, each with MI $>$ 0.44) are protocol-specific, which limits transferability beyond IoT/IIoT traffic. The contrast with NSL-KDD

Table 4: Triple evaluation of the neural models on identical checkpoints and identical test samples (N=1,000 balanced per seed, seeds 42–44, mean ± inter-seed std). INT8 conversion is evaluated on the host to isolate quantization effects from deployment effects

Model	Pipeline	Acc.	Prec.	Rec.	$\mathbf{F_1}$	Latency (μs)
DNN	(a) float32 Keras (host)	.803±.009	.958±.007	.633±.014	.762±.012	–
	(b) INT8 TFLite (host)	.803±.010	.958±.011	.635±.013	.763±.013	–
	(c) INT8 ESP32 (on-device)	.733±.011	.990±.006	.470±.024	.637±.021	215.0±5.7
	(c) float32 AVR (on-device)*	.803±.009	.958±.007	.633±.014	.762±.012	19207.8±1.8
CNN	(a) float32 Keras (host)	.781±.010	.957±.008	.588±.015	.728±.014	–
	(b) INT8 TFLite (host)	.778±.012	.958±.011	.582±.019	.724±.017	–
	(c) INT8 ESP32 (on-device)	.775±.008	.956±.012	.577±.009	.720±.010	2913.2±9.9

All pipelines use the same checkpoint (`dnn_tinyids.keras` / `cnn1d_tinyids.keras`) and the same test samples; the ESP32 firmware embeds byte-identical INT8 models (SHA-256 verified). "–" = host evaluation only. *PROGMEM float32 on Arduino Nano/Uno; latency reported for Nano. Re-verified 2026-08-08 on the same firmware and test samples: the on-device INT8 numbers reproduce per-sample exactly (100% agreement with the February 2026 runs, all seeds); enlarging the DNN arena 16→32 KB changes nothing (F_1 identical); and replacing the embedded model with the float32 checkpoint of the same weights restores host-identical predictions on the same board (F_1=0.754, 100% per-sample agreement with the host), isolating the degradation to the INT8 fully-connected kernel of the TFLite Micro runtime (TensorFlowLite_ESP32 1.0.0). Latency re-measured 2026-08-08 (N=1,000, mean ± per-sample std over seeds 42–44): DNN 215.0±5.7 μs, CNN 2913.2±9.9 μs; AVR re-measured 2026-08-09 (N=4,000): 19,207.8 μs.

highlights that dataset difficulty varies enormously, and researchers should avoid treating testbed results as evidence of real-world robustness.

Figure 2 visualizes the accuracy comparison across all models and platforms, where the reversal of the expected classical-neural hierarchy is clearly apparent. DT and KNN consistently occupy the top tier, while the INT8-quantized DNN sits at the bottom.

3.2 Statistical analysis

Statistical testing confirms the performance hierarchy observed in Table 3. A paired t-test on ten seeds shows DT significantly outperforms RF (t=16.01, p=6.39 × 10^{-8}, Cohen's d=5.06, 95% CI [0.055; 0.073]), whereas the DT–KNN difference is not statistically significant (p=0.116). For the classical-neural comparison (three seeds), DT significantly outperforms ESP32 DNN (F_1 gap = 0.196, Holm-Bonferroni corrected p=0.008) and ESP32 CNN (F_1 gap = 0.113, Holm-Bonferroni corrected p=0.005). The Friedman test across all five ESP32 models yields χ^2=11.47 (p=0.022), with post-hoc analysis ranking DT and KNN at the top and INT8 DNN at the bottom—a reversal of the commonly expected neural-over-classical ordering.

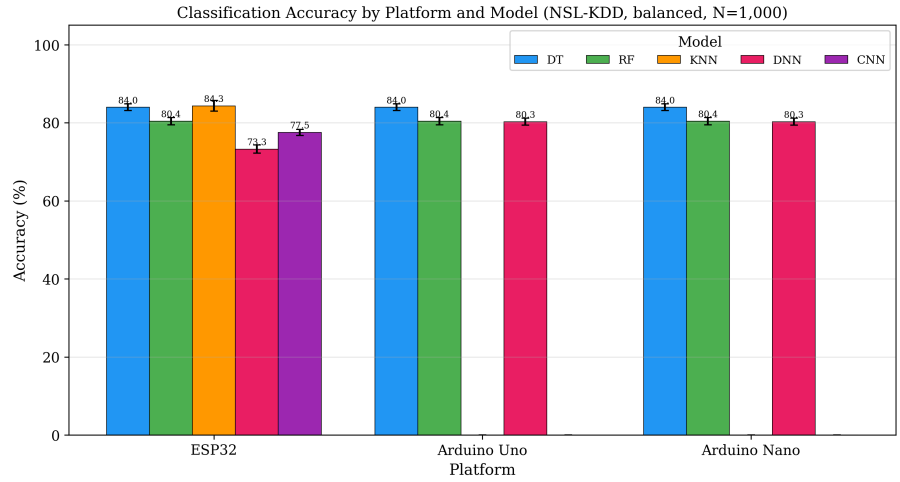


Figure 2: Accuracy comparison across all five models and platforms (NSL-KDD, balanced, N=1,000). Classical DT and KNN outperform INT8-quantized neural models (DNN, CNN), reversing the expected performance hierarchy.

3.3 Confusion matrix analysis

Table 5 presents the mean confusion matrix across evaluation seeds for each model-platform combination. DT achieves the highest true-positive count (389) but at the cost of the highest false positives (49). In contrast, DNN INT8 on ESP32 produces only 2 false positives (99% precision) but misses 265 out of 500 attacks (47% recall), confirming that the deployed INT8 DNN on ESP32 is an overly conservative classifier. The float32 DNN on AVR partially recovers recall (317 TP) compared to its ESP32 counterpart, illustrating the deployment-specific cost of the INT8 DNN path on ESP32 (section 4).

Table 5: Mean confusion matrix per model-platform (NSL-KDD, balanced, N=1,000). Classical models averaged over 10 seeds; neural models over 3 seeds

Platform	Model	TP	FP	TN	FN
ESP32	DT	389	49	451	111
ESP32	RF	318	14	486	182
ESP32	KNN	359	16	484	141
ESP32	DNN (INT8)	235	2	498	265
ESP32	CNN (INT8)	289	13	487	211
AVR	DNN (f32)	317	14	486	183

3.4 Inference latency

Table 6 reports inference latency. DT is the fastest model on every platform: $2.65 \pm 0.03\,\mu$s on ESP32 and approximately 20.7 μs on AVR. DNN requires 215 μs on ESP32 (INT8) and 19,208 μs on AVR (float32 PROGMEM). CNN requires 2,913 μs on ESP32 only. KNN is

impractically slow at 14,879 μs due to brute-force distance computation over 5,000 reference samples.

Table 6: Inference latency (μs), NSL-KDD, mean $\pm$ std

Platform	Model	Latency (μs)
ESP32	DT	2.65 ± 0.03
	RF	8.94 ± 0.05
	KNN	$14{,}879 \pm 1$
	DNN (INT8)	215.0 ± 5.7
	CNN (INT8)	$2{,}913.2 \pm 9.9$
Uno	DT	20.73 ± 0.09
	RF	63.06 ± 0.20
	DNN (f32)	$19{,}207.8 \pm 1.8$
Nano	DT	20.76 ± 0.10
	RF	63.07 ± 0.22
	DNN (f32)	$19{,}207.8 \pm 1.8$

The cross-platform latency ratio differs between paradigms. For classical models, AVR latency is roughly $8\times$ the ESP32 value, consistent with the 15:1 clock ratio offset by architectural differences. For neural models, the ratio climbs to approximately $89\times$ because AVR must emulate float32 arithmetic entirely in software for over 1,000 multiply-accumulate operations per inference, whereas ESP32 uses INT8 fixed-point arithmetic with hardware support. On ESP32, DT achieves a theoretical inference-only throughput of 377,358 inferences per second—excluding feature-extraction and serial-communication costs, so it is not an end-to-end claim for a full 100 Mbps pipeline. DNN manages 4,651 inf/s, while CNN and KNN are limited to 343 and 67 inf/s, respectively.

Figure 3 visualizes the F_1 vs. latency trade-off. DT consistently occupies the ideal upper-left region (highest F_1, lowest latency), while neural models cluster in the high-latency, lower-F_1 region—illustrating the practical "deployment tax" of quantized neural IDS.

3.5 Memory footprint

Table 7 presents the memory analysis. DT fits comfortably on all platforms (approximately 2 KB, 6.3% of Arduino flash). RF is feasible but tight at 25% flash utilization. KNN's 200 KB reference data exceeds Arduino's 32 KB flash by a factor of $6.25\times$, rendering it categorically infeasible on 8-bit platforms. DNN occupies roughly 4.1 KB (12.8% of Arduino flash; this figure covers the PROGMEM weights alone, 4,100 B, while the complete sketch including the inference code uses 42% of flash) and requires only 200 bytes of SRAM via the alternating-buffer PROGMEM approach. CNN is deployable solely on ESP32 because the TFLite Micro runtime is unavailable on AVR and manually implementing convolutional operations in bare C would add substantial complexity for marginal benefit.

Feature importance analysis (Figure 4) reveals that for NSL-KDD, the `flag` feature dominates DT splitting decisions (importance = 0.493), followed by `service` (0.168) and `dst_host_same_srv_rate` (0.143). These top-3 features collectively account for more than 80% of the classification power, which explains why a shallow tree (`max_depth` $= 5$) is sufficient for high performance.

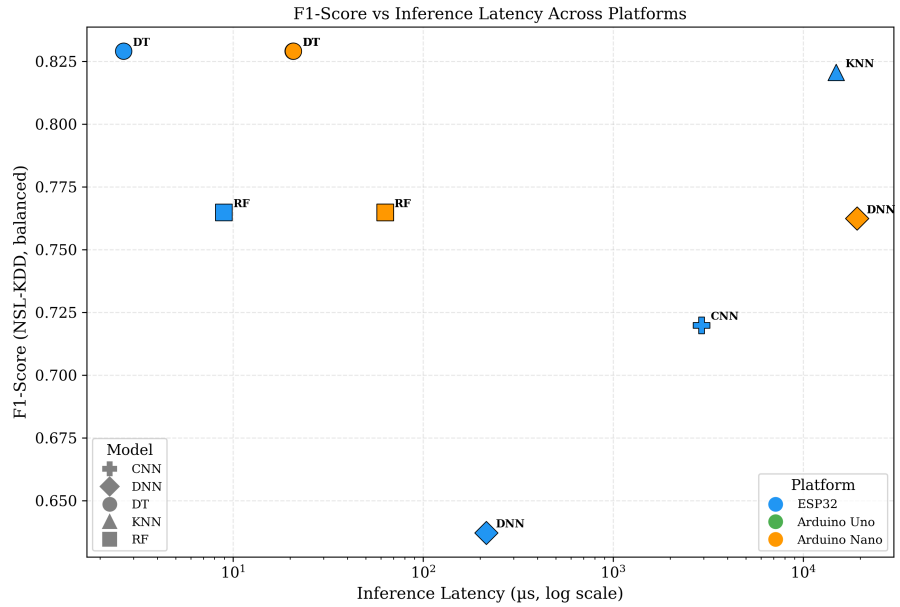


Figure 3: F_1-score vs. inference latency (log scale) for all five models across platforms. DT occupies the ideal upper-left region (highest F_1, lowest latency). Neural models cluster in the high-latency, lower-F_1 region, illustrating the "deployment tax" of quantized neural IDS.

Table 7: Memory footprint per model-platform combination

Model	Plat.	Model (KB)	SRAM (B)	Feas.
DT	ESP32	∼2	56	✓
	Uno/Nano	∼2	56	✓
RF	ESP32	∼8	68	✓
	Uno/Nano	∼8	68	✓
KNN	ESP32	∼200	20K	✓
	Uno/Nano	∼200	20K	×
DNN	ESP32	∼4.2	16K	✓
	Uno/Nano	∼4.1	∼200	✓
CNN	ESP32	∼3.8	24K	✓
	Uno/Nano	—	—	×

4 Discussion

4.1 Answering the research questions

Regarding RQ1, Decision Tree delivers the best overall trade-off among all five models. It achieves the highest F_1 (0.829 ± 0.010) while maintaining the fastest inference ($2.65\,\mu$s on ESP32). The INT8-quantized neural models fall short: DNN reaches only F_1=0.637 and CNN reaches F_1=0.720. Even DNN in float32 mode on AVR (F_1=0.762) does not match DT.

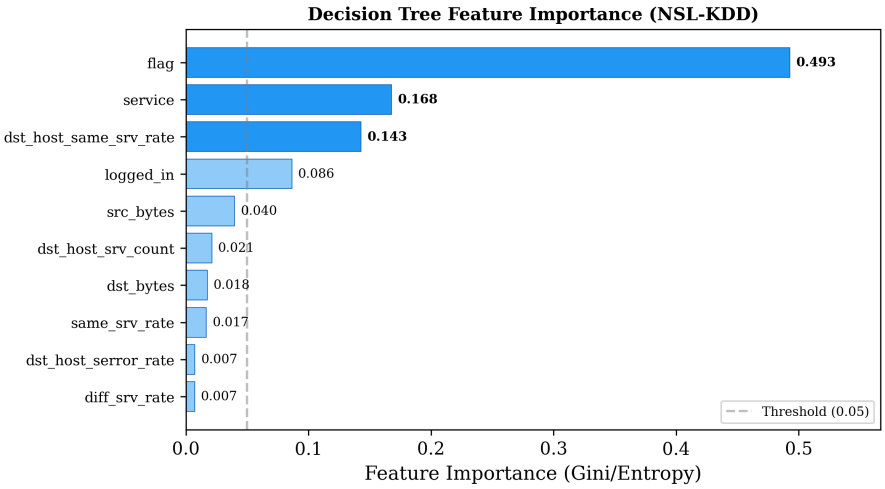


Figure 4: Decision Tree feature importance for NSL-KDD (top-10 MI features). The top-3 features (`flag`, `service`, `dst_host_same_srv_rate`) account for more than 80% of splitting decisions, explaining why shallow trees are effective for this dataset.

This outcome runs counter to the common expectation that neural networks automatically outperform classical approaches on tabular data, a finding consistent with recent large-scale tabular benchmarks [30, 31].

For RQ2, dataset choice profoundly affects what researchers can conclude. On Edge-IIoTset, every model achieves perfect accuracy, which misleadingly suggests that model selection does not matter. Only NSL-KDD reveals the meaningful performance differences between classical and neural approaches. This observation reinforces earlier warnings about the disconnect between research accuracy and real-world deployment [21]. Figure 5 provides a side-by-side comparison of F_1 scores across both datasets, visually illustrating how Edge-IIoTset's perfect accuracy masks the meaningful classical-neural performance gap that NSL-KDD reveals.

Addressing RQ3, memory emerges as the binding constraint on 8-bit platforms. KNN cannot fit on Arduino at any subsample size. RF is feasible but consumes a quarter of the available flash. DT and DNN both fit comfortably, making them the only viable neural and classical candidates for ultra-constrained devices. CNN can only run on ESP32 because TFLite Micro is fundamentally incompatible with the ATmega328P architecture.

For RQ4, to isolate the effect of quantization from the effect of deployment, we evaluated the same checkpoints on the same 1,000-sample balanced test subsets (seeds 42–44) along three pipelines: (a) the original float32 Keras models on the host, (b) the INT8 post-training-quantized TFLite models on the host, using the byte-identical model files embedded in the ESP32 firmware (verified by SHA-256), and (c) the deployed firmware on the target devices. Table 4 reports the outcome.

On the host, INT8 post-training quantization is essentially lossless for both neural architectures: DNN F_1 changes from 0.762 ± 0.012 (float32) to 0.763 ± 0.013 (INT8)—means 0.7624 and 0.7634, paired t-test p=0.139 (n.s.)—and CNN F_1 from 0.728 ± 0.014 to 0.724 ± 0.017 (p=0.358, n.s.). The previously reported 12.5-point drop therefore cannot be attributed to

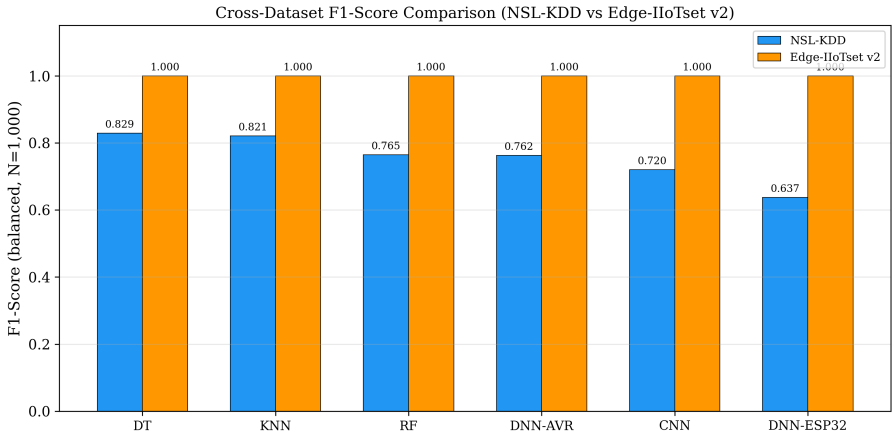


Figure 5: Cross-dataset F_1-score comparison for all five models (balanced, $N{=}1{,}000$). Edge-IIoTset yields uniform perfect scores, while NSL-KDD reveals the classical-neural performance hierarchy: DT $>$ KNN $>$ RF $>$ DNN-AVR $>$ CNN $>$ DNN-ESP32.

INT8 quantization per se. Instead, the drop is specific to the DNN deployment path on ESP32: the same INT8 model reaches $F_1{=}0.6372{\pm}0.021$ inside TensorFlow Lite Micro, while the CNN on the same board, same firmware, and same inputs degrades only marginally (F_1 0.724 host vs. 0.720 device, $p{=}0.533$). A per-sample audit shows the ESP32 DNN never emits a false positive relative to the host (0/1000 in all seeds) but misses roughly one in five attacks that the host flags with high confidence (median host probability $\approx$ 0.90), i.e., a systematic shift of the DNN path toward the normal class. Because the CNN shares the preprocessing and serialization pipeline, and because the float32 PROGMEM deployment of the same DNN on 8-bit AVR reproduces host predictions exactly ($F_1{=}0.7624$), we conclude that the degradation originates in the fully-connected (Dense) INT8 inference path of the TFLite Micro runtime on ESP32, not in the quantization scheme, the checkpoint, or the test protocol. This mechanism was verified on the hardware: replacing the embedded model file with the float32 checkpoint of the same weights (same board, same firmware, same test samples) restores host-identical predictions on the ESP32 ($F_1{=}0.7536$, 100% per-sample agreement with the host, 0/1000 disagreement), and enlarging the interpreter arena from 16 to 32 KB leaves the results unchanged. The degradation is therefore caused by the INT8 fully-connected kernel of the TFLite Micro runtime (TensorFlowLite_ESP32 1.0.0) on the DNN path, not by post-training quantization, the arena configuration, or the serialization protocol. Consequently, the claim that "post-training quantization is responsible for the performance degradation" is revised: quantization was shown lossless on the host, and the observed on-device loss is a deployment-specific effect of the ESP32 DNN pipeline (runtime kernel FullyConnected INT8). This 0.6372 figure remains a verified, reproducible measurement (per-sample agreement is 100% against the February 2026 runs).

4.2 Comparison with prior work

Table 8 contextualizes our deployed results against prior work on NSL-KDD. Our DT F_1 of 0.829 aligns well with values reported in survey literature [4,5]. Banerjee et al. [16] demonstrated TinyML neural networks for edge IDS but used different hardware and did not compare against classical ML on the same platform. Lidholm et al. [15] investigated ML-based IDS on edge devices without cross-architecture evaluation. De Carvalho Bertoli et al. [32] presented an end-to-end ML-based IDS framework highlighting reproducible pipelines, a concern our work addresses through open-source deployment code. These comparisons are contextual rather than head-to-head: the cited studies evaluate on desktop-class hardware with their own datasets, feature sets, and evaluation protocols, whereas our values come from models actually deployed on 8-bit and 32-bit microcontrollers under a fixed NSL-KDD pipeline. Differences in these settings, not only model choice, can account for part of the observed score gaps, so Table 8 is best read as a coarse plausibility check rather than a ranking. Our work adds three novel elements: a direct classical-versus-neural comparison on identical embedded hardware, quantified evidence isolating the deployment-specific degradation of the INT8 DNN path on ESP32 from host-side quantization effects, and a reusable PROGMEM-based deployment methodology for 8-bit AVR where no ML runtime exists.

Table 8: Comparison with prior work on NSL-KDD (binary classification). All values were obtained on desktop-class hardware with varying datasets, feature sets, and evaluation protocols unless marked "Deployed" (on-device evaluation); they are reported for context and are not strictly comparable with our deployed measurements. The "Setting" column records the dataset and evaluation environment of each study

Study	Setting	Model	Acc.	$\mathbf{F}_1$
Ahmad et al. [4]	NSL-KDD, desktop	DT (unconstrained)	0.857	0.91
Khraisat et al. [5]	BotnetIoT, desktop	DT (survey avg.)	–	0.85–0.90
Lidholm et al. [15]	Raspberry Pi, deployed	RF (edge)	0.82	–
Banerjee et al. [16]	BoT-IoT, desktop	TinyML DNN	0.95*	–
Ashraf et al. [17]	BoT-IoT, desktop	Statistical ML	0.95†	–
This work (deployed)	**NSL-KDD, deployed (ESP32/AVR)**	**DT (depth 5)**	**0.843**	**0.829**
This work (deployed)	**NSL-KDD, deployed (ESP32/AVR)**	**DNN INT8 (ESP32)**	**0.733**	**0.637**
This work (deployed)	**NSL-KDD, deployed (ESP32/AVR)**	**DNN f32 (AVR)**	**0.803**	**0.762**

*Different dataset and hardware; †BoT-IoT dataset [33]. Non-deployed values are desktop results reported as published; because hardware, dataset variant, and evaluation protocol differ across studies, comparisons are contextual rather than equivalent. Survey ranges are approximate.

The "deployment tax" for neural models extends beyond latency (81× slower than DT on ESP32) to include a deployment-specific accuracy loss of the DNN path on ESP32, a trade-off rarely reported in TinyML IDS literature. Quantization-aware training (QAT) may mitigate this deployment-specific degradation and represents important future work [34].

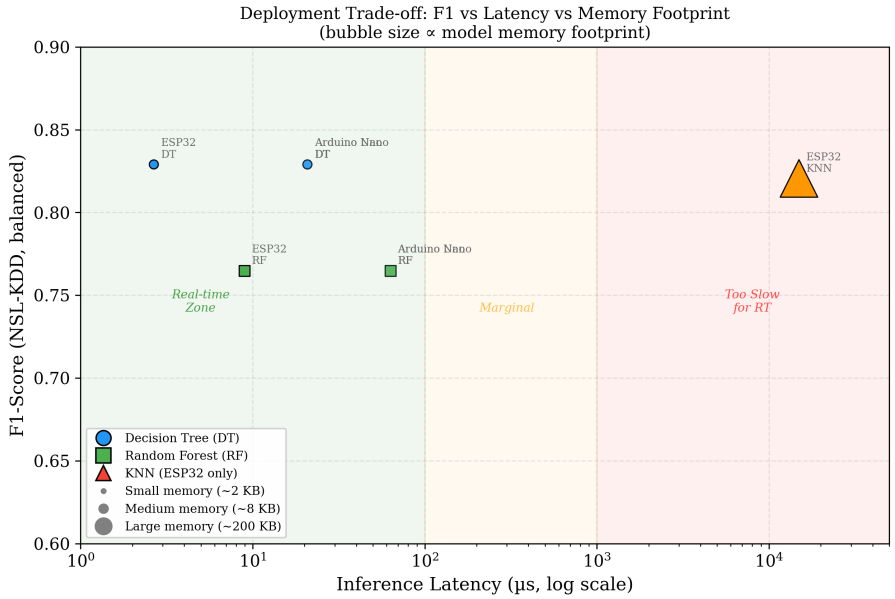


Figure 6: Deployment trade-off visualization: F_1 vs. latency (log scale) with bubble size proportional to memory footprint (KB). DT consistently occupies the ideal upper-left region across all platforms, offering the best combination of accuracy, speed, and memory efficiency.

4.3 Practical implications

DT's recall of 0.783 means approximately 22% of attacks go undetected. While standalone industry IDS typically targets above 90% detection [21], an embedded IDS is best positioned as one layer in a defense-in-depth architecture [35]. A lightweight on-device detector serves as a first-line volumetric attack filter for DoS and Probe attacks, with missed stealthy attacks handled by gateway-level or cloud-based systems. In this layered context, F_1=0.829 with sub-three-microsecond latency adds meaningful security to devices that would otherwise have no detection capability at all.

Figure 6 summarizes the three-way deployment trade-off among F_1, latency, and memory footprint across all model-platform combinations. DT consistently occupies the optimal region, confirming its suitability as the primary on-device IDS model.

A complementary multi-model strategy also emerges: DT as the primary detector (highest F_1, microsecond latency) combined with DNN as a high-confidence precision validator (99% precision, useful for confirming alerts with near-zero false positives). The continuous sigmoid output of neural models additionally enables adjustable thresholds for site-specific precision-recall tuning.

4.4 Deployment lessons

Three practical bugs emerged during deployment. First, a double-normalization error occurred when pre-normalized samples were re-normalized on-device, compressing values

and silently degrading accuracy. Second, MicroMLGen produced incorrect leaf-node assignments when scikit-learn's `class_weight` parameter was used, requiring manual verification of the exported C tree structure. Third, the discovery that TFLite Micro is incompatible with classic 8-bit Arduino boards (despite confusingly similar board names in the documentation) led to the development of the framework-free PROGMEM approach described in section 2. These experiences underscore that numerical verification is a mandatory step in any TinyML deployment pipeline.

4.5 Limitations and future work

This study has several limitations. Power consumption is not measured, although energy efficiency may matter more than latency for battery-powered IoT devices [36]. A rough estimation using typical active-mode current draws (ESP32: $\sim$80 mA at 3.3 V) yields per-inference energy of $\sim$0.70 μJ for DT versus $\sim$57 μJ for DNN INT8, an 81$\times$ difference that mirrors the latency gap and further favors classical models for battery-powered deployments. Only five models from two paradigms are evaluated; gradient-boosted trees such as XGBoost, the current state-of-the-art for tabular data [30, 31], are absent because MicroMLGen does not support them. Only post-training quantization is tested. Because host-side INT8 post-training quantization proved lossless for these models, the earlier motivation for quantization-aware training is weakened for the quantization step itself; QAT remains relevant as a potential remedy for the deployment-specific degradation observed on the ESP32 DNN path, which we leave for future work without speculating on its magnitude. The evaluation covers two dataset families; additional benchmarks such as UNSW-NB15 [37] and CIC-IDS2017 [38] would strengthen generalizability. Future directions include lightweight transformer architectures, MCUNet-style architecture search [39], power profiling, multi-class detection, evaluation on ARM Cortex-M platforms with CMSIS-NN optimized kernels, on-device concept drift monitoring, adversarial robustness evaluation [40], and federated on-device learning [41].

5 Conclusion

This study presented a deployment-oriented benchmark comparing five machine learning models—three classical and two neural—for intrusion detection on three resource-constrained IoT platforms. Four principal findings emerged. On NSL-KDD, Decision Tree achieved the best overall trade-off (F_1=0.829 $\pm$ 0.010, 2.65 μs on ESP32), outperforming all neural variants. The DNN deployed via INT8 TFLite Micro on ESP32 exhibited severe precision-recall imbalance (F_1=0.637), whereas the same DNN deployed in float32 PROGMEM on 8-bit AVR reached F_1=0.762; because host-side INT8 evaluation was lossless (F_1 0.7634 vs. 0.7624, n.s.) and the AVR deployment reproduced host predictions exactly, the gap is a deployment-specific effect of the INT8 TFLite Micro runtime on ESP32, not of post-training quantization. CNN was deployable only on ESP32 via TFLite Micro (F_1=0.720). Dataset difficulty profoundly influenced conclusions: uniform perfect scores on Edge-IIoTset served as deployment-context evidence only and masked the meaningful differences that NSL-KDD revealed; the quantitative findings are therefore limited to NSL-KDD. A limitation of this study is that quantization effects were evaluated with the host-side TensorFlow Lite interpreter (natural evaluation), which does not capture the on-device

TFLite Micro kernel behavior that had to be measured separately on hardware. These results challenge the assumption that neural models automatically outperform classical ML for embedded IDS on tabular data, and provide evidence-based guidance for practitioners selecting models under strict hardware constraints.

All code, firmware, trained model headers, and benchmark scripts are publicly available at: https://github.com/NDP4/tinyml-ids-benchmark.

Acknowledgments

The authors thank the anonymous reviewers for their constructive feedback.

References

[1] M. A. Al-Garadi, A. Mohamed, A. K. Al-Ali, X. Du, I. Ali, and M. Guizani, "A survey of machine and deep learning methods for Internet of Things (IoT) security," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 3, pp. 1646–1685, 2020.

[2] F. Hussain, R. Hussain, S. A. Hassan, and E. Hossain, "Machine learning in IoT security: Current solutions and future challenges," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 3, pp. 1686–1721, 2020.

[3] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[4] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, "Network intrusion detection system: A systematic study of machine learning and deep learning approaches," *Trans. Emerg. Telecommun. Technol.*, vol. 32, no. 1, p. e4150, 2021.

[5] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, 2019.

[6] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1153–1176, 2016.

[7] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Comput. Security*, vol. 28, no. 1–2, pp. 18–28, 2009.

[8] I. Ullah and Q. H. Mahmoud, "Design and development of a deep learning-based model for anomaly detection in IoT networks," *IEEE Access*, vol. 9, pp. 103906–103926, 2021.

[9] A. A. Diro and N. Chilamkurti, "Distributed attack detection scheme using deep learning approach for Internet of Things," *Future Gener. Comput. Syst.*, vol. 82, pp. 761–768, 2018.

[10] V. Dutta, M. Choras, R. Kozik, and M. Pawlicki, "A deep learning ensemble for network anomaly and cyber-attack detection," *Sensors*, vol. 20, no. 16, p. 4583, 2020.

[11] M. Almaini, A. Al-Sherbaz, and S. Sheridan, "Deep recurrent neural network for IoT intrusion detection system," *Simul. Model. Pract. Theory*, vol. 98, p. 102031, 2020.

[12] Y. Meidan *et al.*, "N-BaIoT—network-based detection of IoT botnet attacks using deep autoencoders," *IEEE Pervasive Comput.*, vol. 17, no. 3, pp. 12–22, 2018.

[13] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ. — Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, 2022.

[14] L. Dutta and S. Bharali, "TinyML meets IoT: A comprehensive survey," *Internet Things*, vol. 16, p. 100461, 2021.

[15] P. Lidholm, T. Markovic, M. Leon, and P. E. Strandberg, "Network intrusion detection using machine learning on resource-constrained edge devices," in *Proc. IEEE IJCNN*, 2024.

[16] A. Kundu, L. Wynter, R. D. Lee, and L. A. Bathen, "Transfer-Once-For-All: AI model optimization for edge," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, (Chicago, IL, USA), pp. 26–35, IEEE, 2023.

[17] J. Ashraf *et al.*, "IoTBot-IDS: A novel statistical learning-enabled botnet detection framework for protecting networks of smart cities," *Sustain. Cities Soc.*, vol. 72, p. 103041, 2021.

[18] V. J. Reddi *et al.*, "Widening access to applied machine learning with TinyML," *Harvard Data Sci. Rev.*, vol. 4, no. 1, 2022.

[19] C. R. Banbury *et al.*, "Benchmarking TinyML systems: Challenges and direction," in *Proc. NeurIPS Datasets Benchmarks Track*, 2021.

[20] A. Hamdouchi and A. Idri, "Evaluating the performance of TinyML singular and ensemble techniques for intrusion detection in IoT networks," *Microprocessors and Microsystems*, vol. 104, p. 105172, 2025.

[21] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE S&P*, pp. 305–316, 2010.

[22] Simone (EloquentArduino), "micromlgen: Generate C code for microcontrollers from Python's sklearn classifiers." GitHub repository, 2020. v1.1.13, repository archived 2024; accessed 2025-12-01.

[23] R. David *et al.*, "TensorFlow Lite Micro: Embedded machine learning for TinyML systems," in *Proc. MLSys*, pp. 800–813, 2021.

[24] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE CISDA*, pp. 1–6, 2009.

[25] M. A. Ferrag, O. Friha, D. Hamouda, L. Maglaras, and H. Janicke, "Edge-IIoTset: A new comprehensive realistic cyber security dataset of IoT and IIoT applications for centralized and federated learning," *IEEE Access*, vol. 10, pp. 40281–40306, 2022.

[26] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, "A survey of network-based intrusion detection data sets," *Comput. Security*, vol. 86, pp. 147–167, 2019.

[27] M. Sarhan, S. Layeghy, N. Moustafa, and M. Portmann, "Towards a standard feature set for network intrusion detection system datasets," *Mobile Netw. Appl.*, vol. 27, pp. 357–370, 2022.

[28] T. M. Booij, I. Chiscop, E. Meeussen, J. Hendriks, and D. Besselink, "ToN_IoT: The role of heterogeneity and the need for standardization of features and attack types in IoT network intrusion data sets," *IEEE Internet Things J.*, vol. 9, no. 1, pp. 485–496, 2022.

[29] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Pitfalls in machine learning for computer security," *Commun. ACM*, vol. 67, no. 11, pp. 104–112, 2024.

[30] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Inf. Fusion*, vol. 81, pp. 84–90, 2022.

[31] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?," *Mach. Learn.*, vol. 113, pp. 7679–7719, 2022.

[32] G. de Carvalho Bertoli, L. A. Pereira, and C. R. Nunes, "An end-to-end framework for machine learning-based network intrusion detection system," *IEEE Access*, vol. 9, pp. 106790–106805, 2021.

[33] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset," *Future Gener. Comput. Syst.*, vol. 100, pp. 779–796, 2019.

[34] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," *IEEE Trans. Neural Netw. Learn. Syst.*, 2022.

[35] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga, "A survey of intrusion detection in Internet of Things," *J. Netw. Comput. Appl.*, vol. 84, pp. 25–37, 2017.

[36] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Comput. Sci.*, vol. 2, no. 3, p. 160, 2021.

[37] N. Moustafa and J. Slay, "The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," *Inf. Secur. J.: A Global Perspect.*, vol. 25, no. 1–3, pp. 18–31, 2016.

[38] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "Towards a reliable intrusion detection benchmark dataset," *Software Networking*, vol. 2017, no. 1, pp. 177–200, 2017.

[39] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," in *Proc. NeurIPS*, vol. 34, pp. 2346–2358, 2021.

[40] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognit.*, vol. 84, pp. 317–331, 2018.

[41] M. A. Ferrag, O. Friha, L. Maglaras, H. Janicke, and L. Shu, "Federated deep learning for cyber security in the Internet of Things: Concepts, applications, and experimental analysis," *IEEE Access*, vol. 10, pp. 3572–3603, 2022.