Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Byte Budgeted Decision Tree for Lightweight Intrusion Detection in Industrial IoT with Ascon Protected Messages

Mohammed Taher amine Sahraoui
Djillali Liabes University
Ali Kourtiche
Ali.kourtiche@univ-sba.dz

Djillali Liabes University
Souaad Belhia
Djillali Liabes University
Santiago Felici-Castell
Universitat de València

Research Article

Keywords: industrial IoT, intrusion detection, lightweight cryptography, Ascon, decision trees, dataset artefacts

Posted Date: November 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11733144/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A Byte Budgeted Decision Tree for Lightweight Intrusion Detection in Industrial IoT with Ascon Protected Messages

Mohammed Taher Amine Sahraoui[1], Ali Kourtiche[1], Souaad Belhia[1] and Santiago Felici-Castell[2]

[1] Department of Computer Science, EEDIS Laboratory, Djillali Liabes University, Sidi Bel Abbès, Algeria

[2] Departament d'Informàtica, Escola Tècnica Superior d'Enginyeria (ETSE), Universitat de València, Burjassot, Valencia, Spain

Corresponding author: Ali Kourtiche, Ali.kourtiche@univ-sba.dz

E-mail addresses: Mohamed.sahraoui@univ-sba.dz (M. T. A. Sahraoui); Ali.kourtiche@univ-sba.dz (A. Kourtiche); Souaad.belhia@univ-sba.dz (S. Belhia); santiago.felici@uv.es (S. Felici-Castell)

ORCID: Mohammed Taher Amine Sahraoui, https://orcid.org/0009-0007-7405-1555; Ali Kourtiche, https://orcid.org/0000-0001-6248-2234; Souaad Belhia, https://orcid.org/0000-0002-1640-6369; Santiago Felici-Castell, https://orcid.org/0000-0002-8703-8575

Abstract

Intrusion detectors for industrial IoT routinely report accuracy above 99% on Edge-IIoTset. We wanted to know how much of that survives when the data stop giving the answer away. Auditing the dataset feature by feature, we found that part of the score comes from how the files were written (capture counters, the source port, even whether an empty field is spelled “0” or “0.0”) rather than from how attacks behave. Once these shortcuts are removed, the macro-F1 of a decision tree falls from 0.95 to 0.83: a humbler figure, but one that means something. With this honest baseline, we ask a practical question: how little must a sensor tell a gateway for the gateway to recognise an attack? Fifteen bytes are enough. They fit in a single block of Ascon-AEAD128, the lightweight cipher standardised by NIST in 2025, and a tree under 7 kB classifies them. On two ESP32 boards, sealing such a message takes about 25 µs, and on the ESP32-C3 it is 2.8 times faster and uses 64% less energy than AES-128-GCM with hardware acceleration (31% less on the DevKit), while the detector adds less than 8% to the time. We also report what is less flattering. On a second dataset, TON_IoT, scores fall from 0.96 to about 0.8 when training and test data are split by capture order, and larger models then beat our tree; and password guessing, SQL injection and cross-site scripting stay hard to catch one packet at a time.

Keywords: industrial IoT; intrusion detection; lightweight cryptography; Ascon; decision trees; dataset artefacts

1 Introduction

Industrial plants now run on thousands of small networked devices: sensors, actuators, gateways, programmable controllers. Connecting them improves monitoring and maintenance, and it also gives attackers a large surface to probe [1]. Two defences are usually proposed side by side. Encryption keeps the traffic confidential and authentic, and an intrusion detection system (IDS) looks for traffic that should not be there. Both cost something, and the devices that need them most have the least to spare: a few hundred kilobytes of memory, a battery, a processor that is already busy with its real job.

Encryption has become cheaper to adopt. In August 2025 the U.S. National Institute of Standards and Technology published SP 800-232, which standardises the Ascon family for constrained devices, including the authenticated cipher Ascon-AEAD128 [2,

3]. Detection is harder to budget, because its cost depends on what the model needs to see. If every packet must travel to a gateway as a long feature vector, the saving made by a light cipher is spent again on the radio.

A second difficulty is the evidence that IDS research stands on. Edge-IIoTset [4] is one of the most used IIoT datasets, and many papers report accuracy above 0.99 on it. Such figures deserve suspicion. Machine-learning studies in security are known to suffer from sampling bias, data snooping and unrealistic splits [5], and models readily learn shortcuts that carry no meaning in deployment [6, 7]. Recent work on IoT intrusion datasets points the same way [8, 9], and Galal [10] shows that the categorical placeholders distributed with Edge-IIoTset are spelled “0” for normal traffic and “0.0” for attacks, so that four columns separate the two on their own. We met the same artefact independently while preparing this study, and we treat it as one of several effects to measure rather than as the whole story.

This paper takes the two questions together. How much of the apparent performance on Edge-IIoTset survives an honest feature audit? And, with only what survives, how few bytes must a node send for a gateway to classify its traffic, if those bytes are protected with Ascon?

Contributions.

1. A five-level evaluation ladder for Edge-IIoTset that removes, one family at a time, capture counters, the client source port, format artefacts and the destination port. It shows where the score comes from: macro-F1 falls from 0.949 (level A) to 0.574 (level D) with the same tree.
2. A correction we did not expect to need. Counters such as `icmp.checksum` are noise as numbers but informative as presence flags, because they reveal the protocol of the packet. Replacing the values by presence indicators lifts the reference level from 0.733 to 0.831 and recovers the UDP flood class (recall from 0.09 to 1.00).
3. A byte-budgeted model. We measure macro-F1 as a function of the number of one-byte features and show that 10 to 15 features are enough to stay within 0.3 points of the 70-feature tree.
4. A message profile for Ascon-AEAD128 that exploits the one-block property of a payload of at most 15 bytes, and a decision tree exported as portable C

code, 7 bytes per node, with a procedure that compares its output to the Python model.

5. A reproducible protocol, with the embedded measurements (time, energy and memory on two ESP32 variants) specified in advance.

Related systems have shown that tree models fit on microcontrollers [11, 12] and that Ascon can be benchmarked in a controlled way [13]. To our knowledge none of them ties the size of the protected message to the detection score on a dataset audited for shortcuts, which is the gap this paper addresses.

The rest of the paper is organised as follows. Section 2 reviews related work, Section 3 states the threat model and architecture, Sections 4 to 6 describe the audit, the model and the message profile, Sections 7 and 8 give the setup and the results, including a second dataset and a comparison with other models, Section 9 reports the embedded evaluation, and Sections 10 and 11 discuss limits and conclude.

2 Background and related work

2.1 Edge-IIoTset and the evaluation problem

Edge-IIoTset was built on a seven-layer IoT and IIoT testbed and released with traffic from normal operation and fourteen attack types, in a form meant for both centralised and federated learning [4]. The version used in most studies is a table of packet-level records, one row per packet, with protocol fields from TCP, HTTP, MQTT, DNS, ARP and others. Its popularity is deserved, and so is some caution. A review of IoT and IIoT datasets traces inflated accuracy to dataset simplicity, class imbalance, repetitive attack patterns and evaluation choices rather than to the learning algorithms [8]. On a different IoT benchmark, Roy and Minu [9] show that host identifiers and absolute timestamps act as shortcuts and that random splitting of flows raises attack recall by up to 14 points; they propose a checklist that includes per-attack reporting and multi-seed validation. Closest to our subject, Galal [10] found that the preprocessing recipe shipped with Edge-IIoTset one-hot encodes categorical columns whose absent-field placeholder is “0” in the normal-traffic branch and “0.0” in the attack branch; four of seven columns then separate normal from attack on their own, and corrected models fall to a macro-F1 between 0.70 and 0.95. These findings fit a broader lesson of the security literature: the pitfalls of machine learning in security are often in the pipeline rather than in the model [5].

2.2 Intrusion detection on constrained devices

Fitting a classifier on a small device is a long-standing topic. Bonsai showed that a tree-based model can run in 2 kB of RAM on an IoT board [14]. On Edge-IIoTset, Diab et al. [11] optimise tree ensembles and compact neural networks under flash and compute limits; their LightGBM model reaches 95.3% accuracy with 75 kB of flash and about 1.2 thousand operations, and they run the pipeline on a Raspberry Pi 3 B+. De Bernardis et al. [12] quantise several classifiers for TON_IoT [15], run them on an ATmega2560 and on an ESP32-C3, and measure a decision-tree inference of 5.6 µs on the RISC-V board; they also see a clear generalisation gap on a second dataset. Tekin et al. [16] measure the energy of intrusion-detection models for smart-home IoT across cloud, edge and microcontroller placements, and find that a decision tree run on the device is the best option for training time, inference time and power. Ieropoulos et al. [17] run a decision tree and a neural network on an ESP32 without an operating system, using traffic they recorded themselves: the tree reaches about 99% accuracy, the network about 96% with a smaller memory footprint. These works settle that small models are feasible. What they leave open is the cost of the data the model consumes: the features still have to travel from the sensing node to wherever the model runs, and, in a secured network, they travel encrypted.

2.3 Lightweight authenticated encryption

After a multi-year competition, NIST selected Ascon for lightweight cryptography [18] and in 2025 published it as SP 800-232, which defines Ascon-AEAD128 together with a hash and two extendable-output functions [2]. Ascon is a sponge-based design whose permutation is built from simple bitwise operations, which suits small processors and hardware implementations [3]. Surveys compare lightweight ciphers by security margin, footprint and speed [19], but measurements are scattered and rarely done under identical conditions. On ESP32 and Raspberry Pi Pico boards, Vaz et al. [20] compare Ascon with five other lightweight ciphers and with AES on execution time, throughput, memory and energy. Atul et al. [13] address this with a reproducible benchmark protocol for Ascon-AEAD128, AES-128-GCM and ChaCha20-Poly1305 that fixes message sizes, associated data and timing boundaries; they deliberately report no hardware figures until a frozen platform exists. Our embedded evaluation (Section 9) follows the same spirit.

2.4 Gap

Three threads are usually pursued apart: honest evaluation of IDS datasets, models small enough for microcontrollers, and standardised lightweight encryption. We connect them with a single quantity, the number of bytes that leave the node. The audit decides which bytes carry real information, the model decides how many are needed, and the cipher decides what protecting them costs.

3 Threat model and system architecture

The node observes a packet, turns a handful of its fields into bytes, seals them with Ascon-AEAD128, and sends the frame to a gateway, where the tag is checked and a decision tree classifies the packet (Fig. 1).

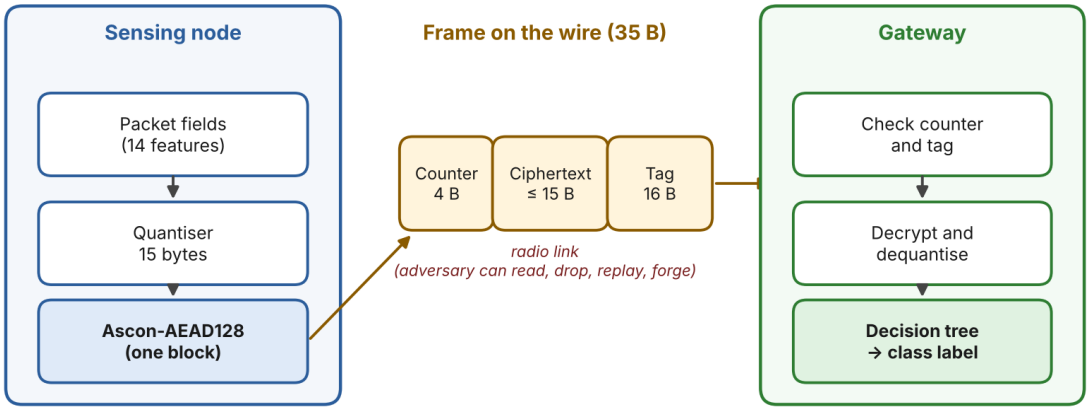

Fig. 1 Architecture of the proposed pipeline: the node seals a 15-byte feature vector with Ascon-AEAD128, and the gateway verifies the tag and classifies the vector with a decision tree

Assets and adversary. We want the gateway to act only on authentic, fresh feature vectors, and we want the vectors to be unreadable on the link. The adversary controls the network between node and gateway: it can read frames, delete or reorder them, replay old ones and fabricate new ones. Keys held inside the node and the gateway are out of its reach. We do not cover physical key extraction, side channels, a compromised gateway, or an attacker who tailors its own packets to evade the tree; the last point matters for features such as the source port, and Section 10 returns to it.

Keys and nonces. Each node shares a 128-bit key with the gateway. The 128-bit nonce is built from a 4-byte node identifier known to both sides, a 4-byte message

counter and eight zero bytes. Only the counter travels in the frame. The gateway stores the last accepted counter per node and rejects any frame whose counter is not strictly greater. A node must be re-keyed before its counter wraps, after 2^{32} messages.

Where the model runs. Classifying on the gateway keeps the node simple and lets one model see all nodes. The tree is small enough, a few kilobytes, to run on the node itself and send a one-byte verdict; we examine that variant in Section 9, because it trades radio bytes for node-side computation.

4 Dataset audit and feature levels

Data. We use the ML-EdgeIIoT-dataset.csv file of Edge-IIoTset [4], obtained from IEEE DataPort (https://doi.org/10.21227/mbc1-1h68; see Availability of data and materials): 157,800 packet-level records in 15 classes, one benign and fourteen attacks. After removing exact duplicates and rows with undefined labels, 152,429 records remain. Records are split 80/20 with stratification, and every reported score is the macro-averaged F1 on the held-out part [21].

Two artefacts we found. A first model reached a macro-F1 near 0.95, which looked too good for per-packet features. Inspecting the columns showed two sources of shortcut learning [6]. The first is a set of counters that behave like capture bookkeeping rather than traffic: checksums, raw sequence and acknowledgement numbers, and stream indices. The second is a formatting artefact. Fields that are empty are written as “0” in some capture branches and as “0.0” in others, and which spelling appears depends on the attack class. A text column such as the HTTP query string therefore carries the label through its length and character counts, without describing any packet. Galal [10] reports the same serialisation effect independently. We map every empty-like value (““,”0”, “0.0”, “0.0.0.0”, “nan”) to one token before encoding, and we publish the audit table in the appendix.

Feature levels. To see how much of the score comes from legitimate evidence, we train the same model on five nested feature sets. Level A keeps everything. Level B drops the bookkeeping counters and adds the text-derived features (length and character counts of the text fields, after the empty-value mapping). Level C also drops the source port, which attackers can pick freely. Level C+ is level C plus three presence indicators (ICMP checksum present, ICMP sequence field present, UDP

stream present); removing the raw ICMP and UDP counters had erased the only trace of these protocols, and DDoS-UDP recall fell to 0.09 until the indicators were added. Level D drops both ports.

Table 1 Macro-F1 of a depth-12 decision tree on Edge-IIoTset at five feature levels (mean over five random splits)

Level	Content	Macro-F1 (tree, depth 12)
A	all fields	0.949
B	no bookkeeping counters; text-derived features added	0.884
C	B without the source port	0.733
C+	C plus three protocol-presence indicators	0.831
D	C+ without any port	0.574

The drop from 0.949 to 0.831 is the price of removing evidence that would not survive a real deployment. We take level C+ as the honest operating point and use it for the rest of the paper. Level D shows how much the destination port still tells the tree.

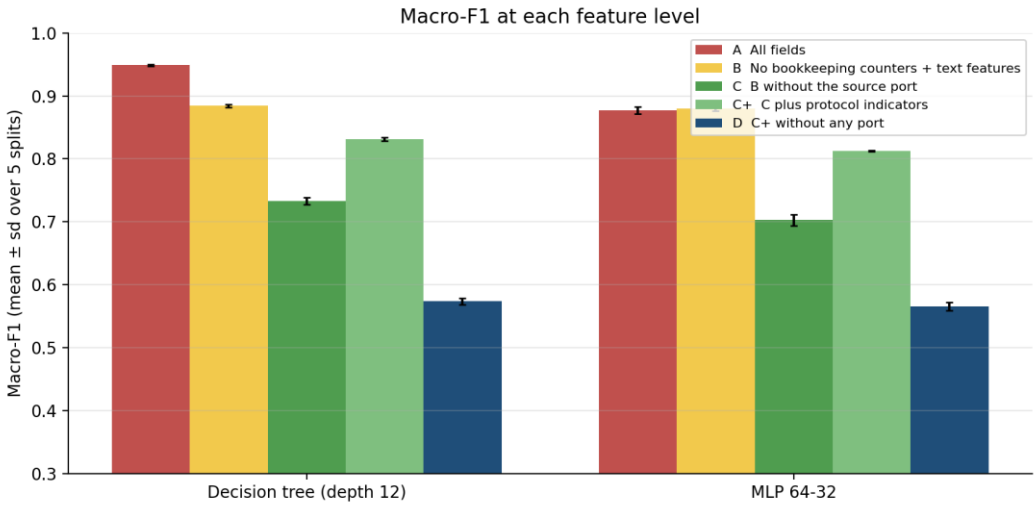

Fig. 2 Macro-F1 of a depth-12 decision tree and of an MLP at each feature level

5 Byte-budgeted model design

The radio frame, not the classifier, is the scarce resource on a constrained node. We therefore design the feature vector around a byte budget and let the model follow.

Quantisation. Each feature is stored in one byte. Integer-valued features that already fit in 0–255 keep their value. For every other feature j we take the 0.5th and 99.5th percentiles of the training split, called l_j and u_j, and map the raw value x to the integer code of Eq. (1):

$$q_j(x) = \mathrm{clip}\left(\left\lfloor 255\,\frac{x-l_j}{u_j-l_j}\right\rfloor, 0, 255\right) \quad (1)$$

The percentile clipping keeps rare extreme values from flattening the resolution of the bulk.

Ports. Ports are categorical, yet a linear byte code treats them as ordinal. We compare three encodings: an 8-bit linear code, a service identifier (a table of the 254 most frequent training ports, with 255 for anything else), and the raw 16-bit port, which costs two bytes.

Selecting features under a budget. Features are ranked by mutual information with the class [22] and by tree importance. We then train trees on the top k features for several depths and keep the smallest tree whose macro-F1 lies within one point of the best (Algorithm 1). The payload budget is 15 bytes because Ascon-AEAD128 absorbs 16 bytes per block, and a message of at most 15 bytes fits one padded block, which saves a permutation call (Section 6).

Algorithm 1 Budget-constrained feature selection

```
Input:  training data D, byte budget B, depth grid P, port encoding m
Output: quantiser Q, feature list S, tree T

fit quantiser Q on D                        // percentiles, port table
rank features by mutual information and by tree importance
for k = 1 .. B:                            // bytes, port may take 2
    S_k <- top-k features under encoding m
    for each depth d in P:
        train tree T_{k,d} on Q(D[S_k]); score macro-F1 over several seeds
best <- max macro-F1 over all (k, d)
choose (k, d) with the fewest nodes such that F1(k, d) >= best - 0.01
return Q, S_k, T_{k,d}
```

Why a tree. A depth-14 tree is a few hundred to a thousand nodes. Each node takes 7 bytes (feature index as uint8, threshold and both child indices as uint16), so the whole model is a constant array of a few kilobytes. Because the inputs are integers, the test $x \le \lfloor t \rfloor$ is exact, and the C export is intended to match the Python model by construction; Section 9 checks this on 300 test vectors. A two-layer perceptron (64 and 32 units) serves as a heavier reference.

6 Secure message profile with Ascon-AEAD128

Ascon won the NIST lightweight cryptography competition [18] and is standardised in NIST SP 800-232 [2, 3]. Ascon-AEAD128 uses a 320-bit sponge state, a 128-bit rate of 16 bytes, and the permutation applied 12 times for initialisation and finalisation and

8 times per absorbed or squeezed block. The cost of a message is therefore easy to count without a clock, which matters here because pure-Python timings say nothing about a microcontroller.

Two profiles. The standard profile mirrors what a generic application would do: float32 features (32 bytes for eight features), a header as associated data, and a 16-byte tag. The optimised profile follows Section 5: one byte per feature, no associated data, and at most 15 payload bytes. Associated data costs an extra block of 8 permutation rounds, and a payload over 15 bytes costs another. Table 2 gives the resulting frame sizes.

Table 2 Frame size and permutation rounds of the standard and optimised message profiles

Profile	Payload	Frame on the wire	Permutation rounds
Standard (float32, with associated data)	32 B	56 B	48
Optimised (int8, no associated data)	8 B	28 B	24
Optimised at the 15-byte budget	15 B	35 B	24

The frame is a 4-byte counter, the ciphertext and the 16-byte tag. Halving the frame matters on duty-cycled radios, where airtime dominates the energy budget. The permutation count is a structural proxy only; the energy and time per message are measured on the ESP32 in Section 9.

Algorithm 2 Seal (node side)

```
Input:  key K, node id N, counter c, quantised vector x (at most 15 bytes)
Output: frame

 1  if c = 2^32 - 1: request re-keying; abort
 2  nonce <- N || c || 0^64
 3  (ct, tag) <- Ascon-AEAD128.Encrypt(K, nonce, AD = empty, x)
 4  c <- c + 1                                  // persist before sending
 5  return c_sent || ct || tag
```

Algorithm 3 Open and classify (gateway side)

```
Input:  key K, node id N, last accepted counter L[N], frame f, tree T
Output: class label or reject

 1  parse f as (c, ct, tag)
 2  if c <= L[N]: reject                        // replay or reorder
 3  nonce <- N || c || 0^64
 4  x <- Ascon-AEAD128.Decrypt(K, nonce, AD = empty, ct, tag)
 5  if x = failure: reject                      // forged or corrupted
 6  L[N] <- c                                   // update only after success
 7  return T(dequantise(x))
```

The counter is stored before transmission so that a reboot can never reuse a nonce, which would break confidentiality. Step 6 of Algorithm 3 updates the stored counter only after authentication succeeds, so a forged frame cannot push the window forward.

Functional check. In a software test, 300 forged frames were rejected out of 300, and a second run of 500 forgeries was rejected 500 times. These runs confirm correct tag handling; they are not performance results.

7 Experimental setup

Protocol. All Edge-IIoTset experiments use a stratified 80/20 split of the cleaned data (the TON_IoT experiments of Section 8 use 70/30). Quantisers, port tables and feature rankings are fitted on the training part only, which avoids the leakage pitfalls catalogued by Arp et al. [5] and Kapoor and Narayanan [7]. Level comparisons are averaged over five random seeds, and the tree-size sweeps over three. We report macro-F1, which weights the fifteen classes equally and so does not let the large benign and flooding classes hide failures on small attack classes, as defined in Eq. (2), where C is the number of classes and P_c and R_c are the precision and recall of class c.

$$\mathrm{F1}_{\mathrm{macro}} = \frac{1}{C}\sum_{c=1}^{C}\frac{2\,P_c R_c}{P_c + R_c} \tag{2}$$

Models. The main model is a CART decision tree (scikit-learn [23]), depth 12 for level comparisons and depth 14 for the exported model. Class imbalance is handled by the tree itself; we apply no oversampling. The reference model is an MLP with two hidden layers of 64 and 32 units.

Software and hardware. Training and evaluation run on Google Colab in Python. The ESP32 evaluation uses a WROOM-32 board, which has hardware AES, and an ESP32-C3 board (RISC-V). Energy is computed as in Eq. (3),

$$E = \frac{1}{N}\sum_{k} V_k \; I_k \, \Delta t_k \tag{3}$$

where N is the number of messages in the window, V_k, I_k and Δt_k are the voltage, current and duration of sample k, and the readings come from an INA219 monitor with a 0.1 Ω shunt, read by a second microcontroller so that the measurement does not disturb the device under test.

Comparison baselines. Ascon-AEAD128 is compared with AES-128-GCM on the same boards, using C implementations. ChaCha20-Poly1305 is left out, because the board package we used does not provide it.

Use of language models. A large language model (Claude, Anthropic) was used as a writing and coding assistant during the preparation of this study: to draft and revise parts of the text, to help write analysis and plotting scripts, and to check numerical consistency between the text, tables and result files. The authors designed the study, ran all experiments and measurements, verified every reported figure against the raw outputs, and take full responsibility for the content.

8 Results

Honest evaluation costs accuracy. Table 1 in Section 4 shows the ladder: 0.949 with every field, 0.831 once the bookkeeping counters, the format artefact and the free source port are removed and the protocol indicators are added back, and 0.574 without any port. Group ablations at the earlier level C point the same way. Removing the ARP fields lowered macro-F1 by 0.086, the HTTP fields by 0.181 and the destination port by 0.081. The classifier relies on protocol evidence that a network operator would recognise, and the indicators show how little is needed to separate flooding classes from each other.

Where the errors are. The confusion matrix (Fig. 3) shows that the remaining errors concentrate in the HTTP-based classes, whose packets look alike one at a time. We expect window-level aggregates (requests per second, distinct URIs) to help these classes, but they require state that a per-packet byte budget does not allow, so we leave them for future work.

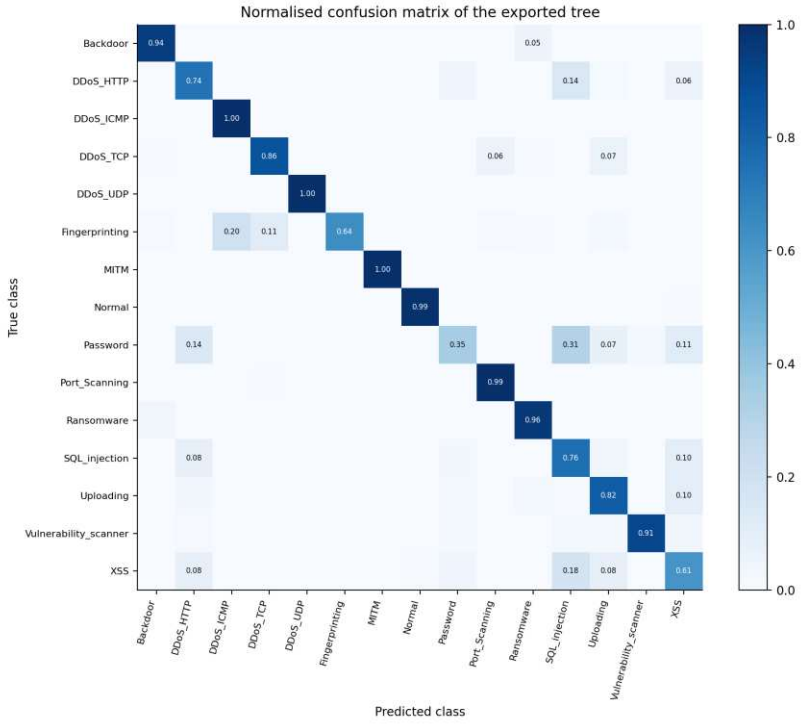

Fig. 3 Confusion matrix of the exported decision tree

A 15-byte payload is enough. Fig. 4 plots macro-F1 against the number of payload bytes. With 10 bytes the model gives up 0.28 points of macro-F1 against the full 70-feature set, and with 15 bytes only 0.11 points. The curve is flat from about 10 bytes on, which is why one Ascon block suffices.

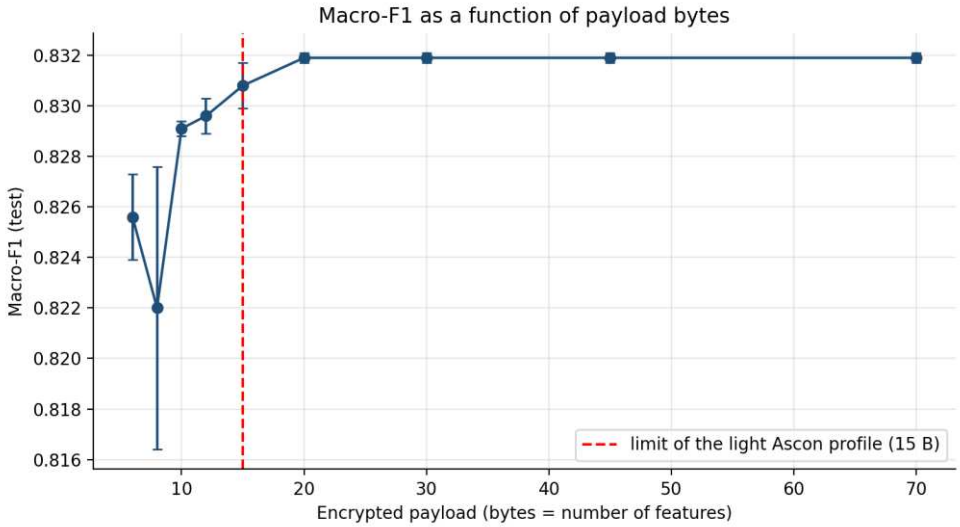

Fig. 4 Macro-F1 as a function of payload bytes

Tree size. Fig. 5 shows model size against macro-F1. Accuracy plateaus near 0.84 at the reference level (C+) and extra nodes beyond the knee buy almost nothing, so the smallest tree within one point of the best is a safe choice. The selected tree

(depth 14, ranked features) has 984 nodes on average over three splits and a macro-F1 of 0.843, or about 6.7 kB at 7 bytes per node; depth 12 stays below 0.832 at any number of features, and depth 16 adds only 0.1 point for almost twice the nodes.

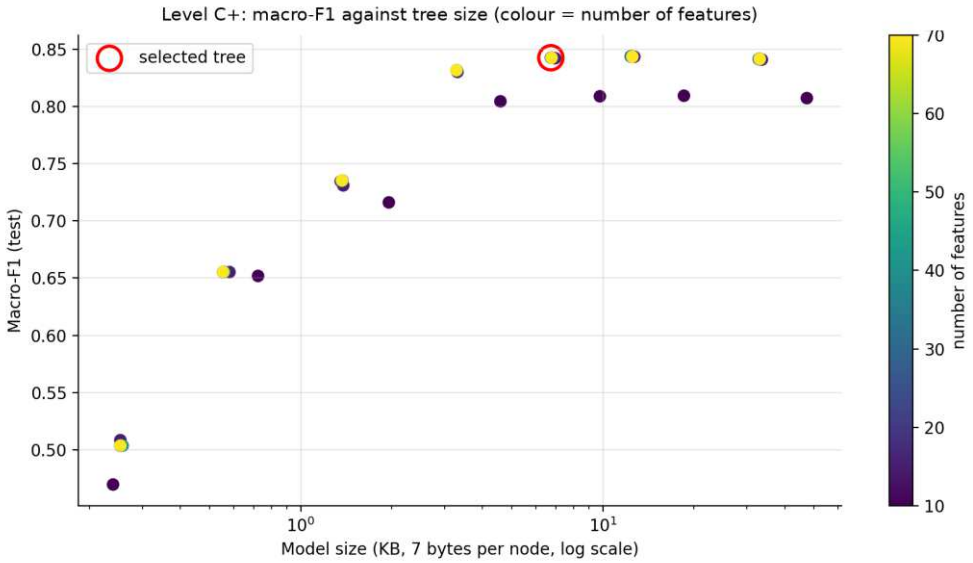

Fig. 5 Model size versus macro-F1 for the tree sweep at level C+

Quantisation and port encoding. Table 3 compares the encodings of the destination port on the exported feature set (14 features, 15 bytes), with the mean and standard deviation of macro-F1 over several seeds. The un-quantised reference scores 0.8407 ± 0.0009 with 14 bytes. The 16-bit port reaches 0.8409 ± 0.0008, so it costs nothing. The 8-bit linear code gives 0.8374 ± 0.0021, a loss of 0.3 points. The service-identifier code is clearly worse at 0.7664 ± 0.0025, because the table of 254 frequent ports merges rare ports that the tree uses to separate attack classes. We therefore export the 16-bit variant.

Table 3 Macro-F1 of the exported feature set under four encodings of the destination port (mean ± standard deviation over several seeds)

Port encoding	Bytes	Macro-F1 (mean ± sd)
Reference (raw values)	14	0.8407 ± 0.0009
8-bit linear	15	0.8374 ± 0.0021
Service identifier (8-bit)	15	0.7664 ± 0.0025
16-bit	15	0.8409 ± 0.0008

Exported model. The final tree has depth 14 and 977 nodes, stored in 6,839 bytes. On the held-out set, with quantised inputs, it reaches a macro-F1 of 0.8417. The 14 features are tcp.dstport, udp.stream presence, tcp.options (digit count), icmp.seq_le

presence, tcp.payload length, tcp.connection.rst, http.file_data length, http.request.method length, tcp.flags.ack, tcp.connection.syn, http.request.full_uri length, tcp.flags, tcp.len and tcp.connection.synack. Compiled with gcc on a PC, the C code and the Python model disagree on 0 of the 300 test vectors; the same check on the ESP32 is part of Section 9.

Message cost. The optimised profile cuts the frame from 56 to 28 bytes and the permutation count from 48 to 24, which halves both quantities relative to the standard profile (Section 6). Wall-clock time and energy are not claimed here.

Second dataset: TON_IoT. To test whether the picture holds outside Edge-IIoTset, we repeated the audit on the network part of TON_IoT [15, 24], file Train_Test_Network.csv obtained from the UNSW Canberra TON_IoT page (https://research.unsw.edu.au/projects/toniot-datasets; see Availability of data and materials): 211,043 connection records taken from Zeek logs, 44 columns, nine attack types and normal traffic (the class mitm has only 1,043 records). The records describe whole connections, not packets, so the 15-byte budget now holds connection summaries (byte and packet counters, duration, connection state, service, protocol and a few DNS flags) instead of header fields. We removed 20,569 exact duplicates (9.7%) and kept the pipeline of Section 5: 8-bit quantisation with percentile clipping, a logarithmic scale for counters, ranking by mutual information, a depth-14 tree and a 16-bit destination port. The text fields (URI, user agent, TLS subject and similar) are replaced by presence indicators, as for level C+.

Table 4 Macro-F1 of a depth-12 tree on TON_IoT at five feature levels (random 70/30 split, mean ± standard deviation over five seeds)

Level	Content	Macro-F1
A	all fields, including source and destination addresses	0.979 ± 0.002
B	no addresses	0.959 ± 0.002
C	B without the source port	0.957 ± 0.003
D	C without any port	0.946 ± 0.002
E	D without byte and packet counters and duration	0.594 ± 0.002

The audit finds shortcuts of the same kind as before. The attacks come from two to five hosts per class, against forty for normal traffic, so a tree that sees the source address reaches 0.979 and one that does not reaches 0.959; removing the ports costs a further 0.013. What carries the detection is the volume of the connection:

without the counters and the duration the score falls to 0.594 (Table 4). At level D the weak classes are mitm (F1 0.69, on only 1,043 records), xss (0.92) and ransomware (0.95). The byte budget matters little on this dataset: with a random split, four one-byte features reach 0.934 and 15 bytes with the destination port reach 0.957, and the features kept are mostly counters (source and destination bytes, packets and IP bytes, then connection state, service and duration).

Two cautions apply. First, the records are highly redundant: after rounding the duration to 0.01 s, only 32,255 of the 190,474 records are distinct, so a random split puts near-copies of each test record in the training set. When we train on the first 70% of each class in file order and test on the last 30%, which is a proxy for a split in time because the file has no timestamp and we assume that its order follows the capture, the score at level D falls from 0.946 to 0.815, and the 15-byte models score between 0.79 and 0.83 whatever the budget. We therefore read the random-split figures as an upper bound. Second, the two datasets differ in their features, classes and balance, so their scores are not comparable with each other; what carries over is the method and the 15-byte message profile of Section 6, which does not depend on the dataset, so the timing, energy and memory results of Section 9 apply to TON_IoT as well.

Comparison with other models on the same 15 bytes. To check that the tree is not a weak choice, we gave the same 15-byte input (13 quantised features chosen by mutual information, plus the 16-bit destination port) to other classifiers and report the macro-F1 and the number of tree nodes, which sets the size of the model on the device (Table 5). The random forests gain only 0.2 to 0.4 points under the random split but 5 to 7 points under the split in file order, at 11 and 59 times the number of nodes of the single tree; a depth-8 tree with a fifth of the nodes loses 4 points. Logistic regression and Gaussian naive Bayes, which cannot separate classes that depend on thresholds of several counters, fall far below. The single depth-14 tree is therefore the best choice at this size, and a forest would need roughly ten times the 7 kB of flash that the tree needs on the ESP32 (our estimate from the node count, not measured).

Table 5 Models on the same 15-byte input, TON_IoT (random split: mean of 3 seeds ± sd; file-order split: one run)

Model	Nodes	Macro-F1, random split	Macro-F1, file-order split
Decision tree, depth 4	31	0.619 ± 0.001	0.366
Decision tree, depth 8	284	0.919 ± 0.003	0.717
Decision tree, depth 14 (ours)	1,332	0.957 ± 0.000	0.795
Random forest, 10 trees, depth 14	15,340	0.960 ± 0.002	0.847
Random forest, 50 trees, depth 14	78,753	0.962 ± 0.002	0.861
Logistic regression	none	0.536 ± 0.032	0.316
Gaussian naive Bayes	none	0.340 ± 0.024	0.320

The same comparison on Edge-IIoTset. Table 6 repeats the test on the exported feature set of Section 5 (14 features, 15 bytes, 16-bit port), with the same two splits (three random splits; one split that takes the first 70% of each class in file order for training and the last 30% for test, which has the same limitation as on TON_IoT). Under random splits the depth-14 tree (0.841) is within 0.4 points of a 50-tree forest (0.845) and 3 points above the MLP (0.807), at 57 times fewer nodes than the forest. Under the file-order split the picture changes: the tree falls to 0.678, while the forests reach 0.729 and 0.732 and the MLP 0.737, the best of all. A tree of 970 nodes is therefore not the most accurate model on unseen sessions; its argument is its size and its cost per message (1.2 to 1.8 µs, Table 7), not a lead in accuracy. The MLP has only 3,535 parameters, which is smaller in memory than the tree, so the choice between them would rest on inference time on the ESP32, which we have not measured. Depth-4 trees and the linear and naive Bayes models are far below on both datasets.

Table 6 Models on the same 15-byte input, Edge-IIoTset (random split: mean of 3 seeds ± sd; file-order split: one run)

Model	Nodes (parameters for the MLP)	Macro-F1, random split	Macro-F1, file-order split
Decision tree, depth 4	12	0.291 ± 0.000	0.289
Decision tree, depth 8	84	0.654 ± 0.009	0.539
Decision tree, depth 14 (ours)	970	0.841 ± 0.001	0.678
Random forest, 10 trees, depth 14	11,361	0.842 ± 0.003	0.729
Random forest, 50 trees, depth 14	55,044	0.845 ± 0.001	0.732
MLP 64-32	3,535	0.807 ± 0.004	0.737
Logistic regression	none	0.658 ± 0.022	0.602
Gaussian naive Bayes	none	0.399 ± 0.008	0.374

9 Embedded evaluation on ESP32

Table 7 reports the decision-tree timing, Table 8 the time to seal one message on both boards, Table 10 the energy to seal one message on both boards, and Table 11 the memory footprint.

Decision tree on the boards. The exported tree (977 nodes, 6,839 bytes of constant tables) was compiled for two boards and run on the 300 stored test vectors, repeated 200 times. Time per prediction is the mean over 60,000 calls, measured with micros().

Table 7 Decision-tree inference on the two boards (300 stored test vectors, repeated 200 times)

Board	Chip	CPU	Free RAM	C vs Python disagreements	Time per prediction
ESP32 DevKit	ESP32-D0WD-V3 (Xtensa, 2 cores)	240 MHz	323 kB	0 of 300	1.23 µs
ESP32-C3-MINI-1	ESP32-C3 rev 4 (RISC-V, 1 core)	160 MHz	294 kB	0 of 300	1.77 µs

On both chips the C code reproduces the Python decisions exactly. A prediction costs about 295 clock cycles on the Xtensa core and about 283 on the RISC-V core, so the two chips run the same tree at nearly the same cycle count and differ in time only through their clock rates. The tables take roughly 2% of the free RAM. Accuracy on the 300 vectors is 85.0%; this is a small sample and does not replace the macro-F1 of Section 8. The time is a warm-cache steady-state figure, since the same vectors are replayed.

Sealing time. Each value in Table 8 is the mean over a window of 10,000 consecutive messages, delimited by a GPIO marker so that the same windows can later be matched to the energy trace. Ascon-AEAD128 is the official ascon-c 32-bit implementation (opt32), which reproduces all 1,089 known-answer tests of the reference suite on a PC. It is compiled with function inlining and loop unrolling switched off; Fig. 9 shows why. AES-128-GCM uses mbedtls through the Arduino-ESP32 core, which drives the hardware AES block. Because the call includes the library's per-message setup, its time is an end-to-end figure and not a pure block-cipher cost. Every message gets a fresh nonce and a 16-byte tag; "AD" means 8 bytes of associated data. ChaCha20-Poly1305 could not be measured, because the board package we used does not compile that primitive in.

Table 8 Time to seal one message (mean of 10,000 messages). Cycles are the time multiplied by the clock rate

Board	Payload / AD	Ascon-AEAD128 (µs)	Ascon-AEAD128 (cycles)	AES-128-GCM (µs)	AES-GCM time / Ascon time
ESP32 DevKit	8 B / none	21.31	5,114	35.70	1.68×
ESP32 DevKit	15 B / none	22.24	5,338	36.71	1.65×
ESP32 DevKit	32 B / 8 B	41.34	9,922	50.95	1.23×
ESP32-C3	8 B / none	24.27	3,883	71.01	2.93×
ESP32-C3	15 B / none	25.40	4,064	71.60	2.82×
ESP32-C3	32 B / 8 B	46.82	7,491	98.60	2.11×

Reading Table 8. Ascon-AEAD128 is faster than hardware-assisted AES-128-GCM in all six cases: by 1.2 to 1.7 times on the ESP32 DevKit and by 2.1 to 2.9 times on the ESP32-C3. The 15-byte sensor frame, the profile our model uses, costs 22.2 µs and 25.4 µs. Ascon's cost follows its structure. Empty associated data and a payload of up to 15 bytes take 24 rounds of the permutation; each further 16-byte block, of data or of message, adds one 8-round permutation, which is why the 32 B + 8 B AD case (48 rounds) costs about twice the 15 B case.

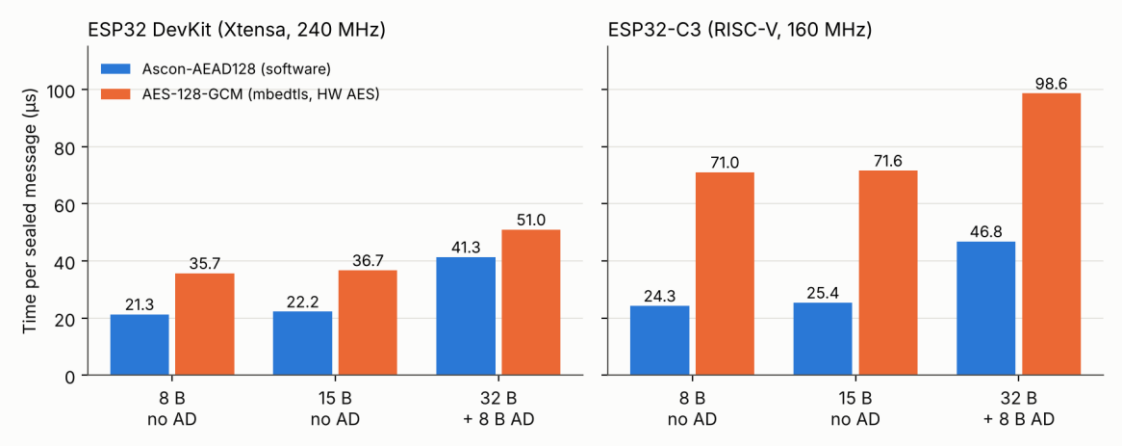

Fig. 6 Time per sealed message for Ascon-AEAD128 and AES-128-GCM on the two boards, for the three message profiles of Table 8

Fig. 6 draws the six pairs of Table 8 side by side. Each panel is one board, both panels share the vertical axis, and in every pair the blue bar is Ascon and the orange bar is AES-GCM. Ascon is lower in every pair. The gap is modest on the ESP32 DevKit, where AES-GCM needs 36.7 µs against 22.2 µs for the 15-byte frame, and large on the ESP32-C3, where it needs 71.6 µs against 25.4 µs.

The Ascon bars are almost the same on the two boards, although the C3 runs at two thirds of the clock rate of the DevKit, whereas the AES-GCM bars are about twice as tall on the C3. Within a board, both primitives grow with message size. The step to the 32 B + 8 B AD profile is larger for Ascon (about 1.8 times the 15-byte cost) than for AES-GCM (about 1.4 times), because Ascon pays for every additional block with a full permutation.

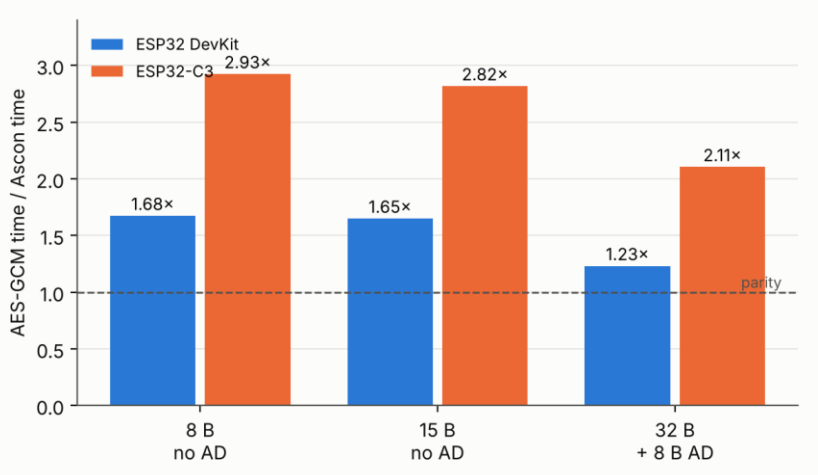

Fig. 7 Ratio of the AES-128-GCM time to the Ascon-AEAD128 time (last column of Table 8)

Fig. 7 turns Table 8 into a speed-up. A bar above the dashed line at 1 means that Ascon is faster, and the height of the bar is by how much. The advantage is largest for the short sensor frame and shrinks as the message grows, from 1.65× to 1.23× on the DevKit and from 2.82× to 2.11× on the C3. A plausible reason is that AES-GCM pays a fixed cost per message that a longer message spreads over more bytes, while Ascon's cost keeps growing with the number of blocks; we have not tested this.

The C3 bars are higher mainly because AES-GCM is slower on that board, not because Ascon is faster. In cycles, Ascon needs about 5,300 on the Xtensa core and about 4,100 on the RISC-V core for the 15-byte frame, while AES-GCM needs about 8,800 and 11,500. We did not isolate the cause of the AES-GCM gap; the per-message overhead of the library is a plausible one.

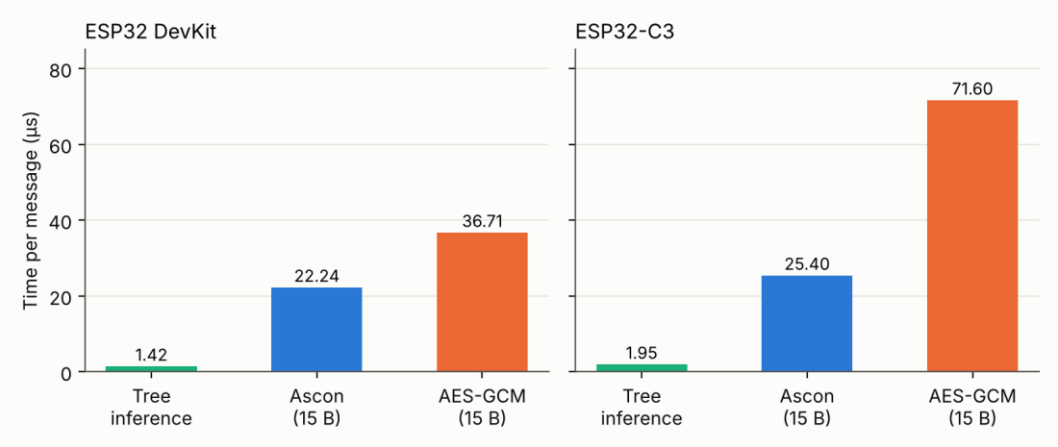

Fig. 8 Time per message for the decision tree and for sealing a 15-byte frame with each cipher

Fig. 8 puts the cost of detection next to the cost of protection. The green bar is the tree inference, measured in the same run as the sealing windows; the other two bars are the two ciphers for the 15-byte frame. Detection is cheap: the tree takes 1.42 µs on the DevKit and 1.95 µs on the C3, which is 6.4% and 7.7% of the Ascon sealing time. A node that already seals with Ascon therefore spends about 24 µs per message on the DevKit instead of 22 µs once it also classifies.

These tree times are a little higher than those of Table 7 (1.23 µs and 1.77 µs) because they come from a different benchmark program, which adds the loop and call overhead of the sealing bench. The order of magnitude is the same.

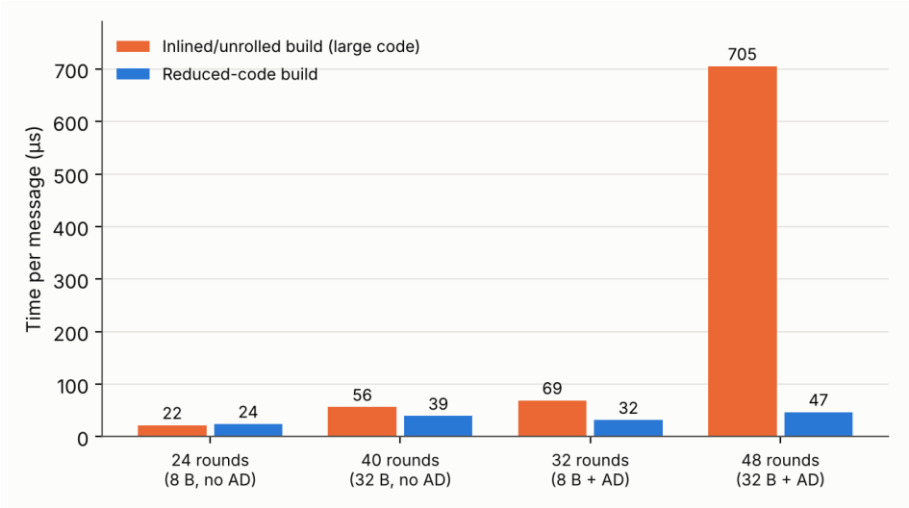

Fig. 9 Ascon time on the ESP32-C3 for four cases, labelled by their number of permutation rounds, with the default build (inlined and unrolled, large code) and with the reduced-code build used for Table 8

If the cost followed the number of rounds, the bars of Fig. 9 would grow steadily from left to right. With the reduced build they do, at about 1 µs per round. With the large build they do not: the 8-byte case is slightly faster (21.8 µs against 24.3 µs), but the cases with associated data or several blocks become disproportionately slow, up to 705 µs for 48 rounds, about 15 times the reduced build (46.8 µs).

Our explanation is that the large code overflows the small instruction cache of the C3, which runs from external flash, so that instructions are fetched repeatedly. We did not read cache counters, so this remains a hypothesis. What we can say is that the reduced build restores proportionality, and it is the one reported in Table 8. On the DevKit we measured only the reduced build.

What these results do and do not say. Both boards have an AES accelerator, and Ascon is still faster here, but the comparison is between two specific implementations: a portable C Ascon and mbedtls with its per-message overhead. A hand-written GCM driving the hardware block directly could narrow the gap. We have not measured a chip without an AES accelerator, where we would expect Ascon's advantage to grow, and the two ciphers add the same 16-byte tag, so frame size does not separate them. Vaz et al. [20] report time and energy for Ascon and AES on an ESP32 under their own protocol, so their figures and ours should be compared only with that difference in mind.

Energy measurement: principle. Energy per message cannot be read from a single 22 µs event, because the INA219 needs about 0.5 ms to convert one sample. We

therefore measure windows, as for time. The device under test seals N messages back to back while a GPIO marker is high; the monitor integrates the power over that window, subtracts the power the board draws when idle, and divides by N (Eq. 4).

$$E_{\mathrm{msg}} = \frac{1}{N}\sum_{k\in\mathrm{window}} (V_k I_k - P_{\mathrm{idle}})\,\Delta t_k \tag{4}$$

Energy measurement: set-up. Fig. 10 shows the wiring. The shunt of the INA219 module sits in series with the +5 V supply of the device under test, so that the whole board current, including the USB-serial chip, flows through it. The +5 V comes from a separate USB supply whose cable was cut open: its red wire enters the module at VIN+ and leaves it at VIN− for the 5 V pin of the board (the 5 V pin of our monitor board is not connected to its USB port, so it cannot serve as the source). A second board reads the INA219 over I²C (address 0x40) and supplies its 3.3 V logic: an ESP32-C3-MINI-1 (SDA on GPIO8, SCL on GPIO9) when the DevKit is under test, and the DevKit (SDA on GPIO21, SCL on GPIO22) when the C3 is under test. The device under test raises a GPIO (GPIO23 on the DevKit, GPIO5 on the C3) for the duration of each window and the monitor reads it on GPIO4, so that the samples taken while it is high belong to the window. All grounds are common, including that of the supply. The USB cable of the device under test stays unplugged during a measurement, so that no current reaches it except through the shunt; the results are read from the monitor.

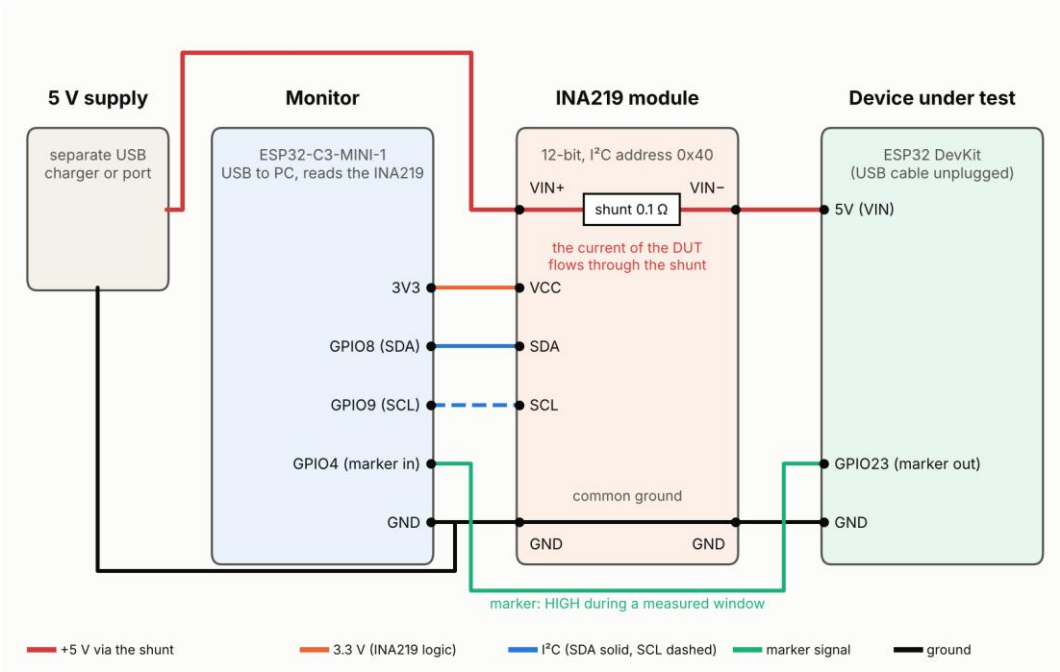

Fig. 10 Wiring of the energy measurement: the INA219 in series with the +5 V supply of the device under test, read by a second board that also receives the window marker. The figure shows the DevKit under test; for the C3 the two boards exchange their roles

Fig. 10 has four blocks. On the far left, a separate USB supply provides the 5 V; its red wire is the only path to the device under test and it runs through the INA219. Next, the monitor board supplies the 3.3 V to the INA219, reads it over the two I²C wires and listens to the marker; its ground is joined to that of the supply. In the middle, the INA219 module measures the voltage across its 0.1 Ω shunt, which gives the current, and the voltage at its output, which gives the supply voltage seen by the device. On the right, the device under test is powered only through the shunt and signals each measured window with the marker. The red path is the only route the device's current can take, which is what makes the reading the current of the whole board.

Table 9 Parameters of the energy measurement

Parameter	Value
Monitor board	ESP32-C3-MINI-1 (DevKit under test); ESP32 DevKit (C3 under test)
Sensor	INA219, 12-bit, I²C address 0x40
Shunt	0.1 Ω (marked R100 on the module; tolerance not measured)
Shunt-voltage resolution	10 µV, i.e. 0.1 mA at 0.1 Ω
Bus-voltage resolution	4 mV
Conversion time	0.53 ms per conversion (12-bit); shunt and bus together 1.06 ms
Supply of the device under test	+5 V from a separate USB supply, through the shunt
Window marker	GPIO23 (DevKit) or GPIO5 (C3) of the device to GPIO4 of the monitor
Idle reference	the 3 s pauses between windows, marker low

Energy measurement: procedure.

1. Scan the I²C bus and check that the INA219 answers at 0x40; check the shunt value, since a 0.1 Ω shunt is marked R100.
2. With the marker low, record the current for several seconds to obtain the idle power.
3. Run the benchmark. At every sample the monitor records the bus voltage, the current and the marker.
4. For each window, compute the energy per message with Eq. (4), repeat the window several times, and report the mean and the standard deviation.
5. Report the ratio between the primitives as well as the absolute values, since part of the offset is common to all of them.

Energy measurement: resolution and bias. At 0.1 Ω the 12-bit converter resolves 0.1 mA. It updates the shunt and bus readings together about every 1.1 ms, and the monitor read it every 0.44 ms (DevKit under test) or 0.58 ms (C3 under test), so successive readings repeat; the time-weighted average is not affected, and a window of 10,000 messages (0.21 to 0.99 s) contains 200 to 930 independent updates. The board draws a steady power inside a window, so this is enough: the run-to-run standard deviation is at most 0.012 µJ. A window edge can fall up to one conversion (about 1 ms) away from the marker edge, which shifts the energy of a 0.21 s window by up to 0.5%, of the order of the observed spread. What the monitor cannot give is an absolute calibration: the shunt was taken as its marked value of 0.1 Ω (R100), and a tolerance of one percent would scale every energy value by the same factor, which

is why we also report ratios. Differences smaller than the standard deviation are treated as not significant.

Energy results. Table 10 gives the energy per sealed message on both boards, as the mean and standard deviation of five runs; each run is a reset of the device under test followed by the nine windows of the benchmark. For the C3 the two boards exchange their roles: the C3 is the device under test, powered at its 5 V pin through the shunt and raising GPIO5 for each window, and the DevKit is the monitor (SDA on GPIO21, SCL on GPIO22, marker on GPIO4). The reference is the power of the board while it waits between windows: 239.6 mW on the DevKit (48.5 mA at 4.94 V) and 99.6 mW on the C3 (19.9 mA at 4.99 V); the energy reported is the extra energy above that level. For the first window of a run, the pause that precedes it follows the reset, and its own reading (303 to 321 mW on the DevKit, 24 to 125 mW on the C3) is not representative, so for that window we used the mean pause level of the other windows. The I²C error counter of the monitor did not increase during any run.

Table 10 Energy to seal one message, above the waiting level of each board (mean ± standard deviation over five runs of 10,000 messages). The last column repeats the time ratio of Table 8

Board	Payload / AD	Ascon-AEAD128 (µJ)	AES-128-GCM (µJ)	AES-GCM energy / Ascon energy	AES-GCM time / Ascon time
ESP32 DevKit	8 B / none	2.43 ± 0.01	3.51 ± 0.01	1.44×	1.68×
ESP32 DevKit	15 B / none	2.52 ± 0.01	3.65 ± 0.01	1.45×	1.65×
ESP32 DevKit	32 B / 8 B	4.74 ± <0.01	5.01 ± 0.01	1.06×	1.23×
ESP32-C3	8 B / none	1.60 ± 0.01	4.64 ± 0.01	2.89×	2.93×
ESP32-C3	15 B / none	1.69 ± <0.01	4.65 ± 0.01	2.75×	2.82×
ESP32-C3	32 B / 8 B	3.11 ± <0.01	6.42 ± <0.01	2.06×	2.11×

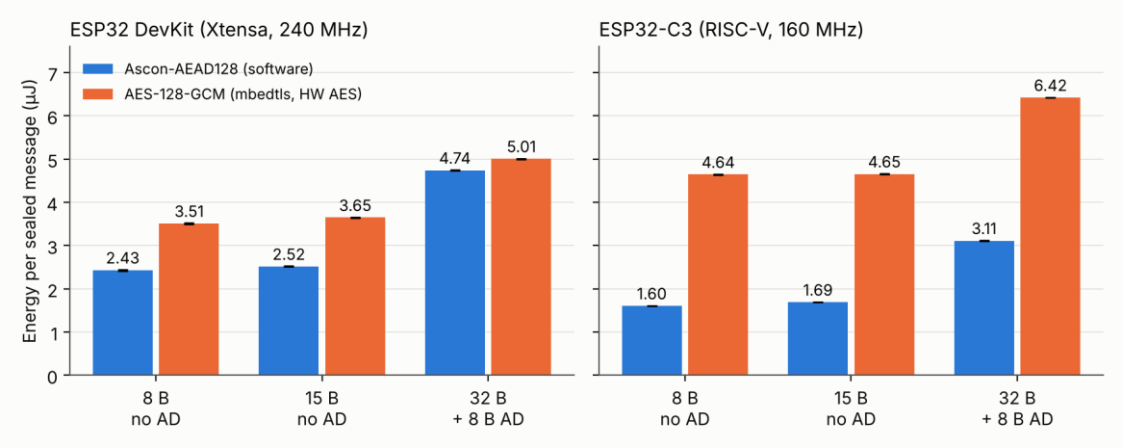

Fig. 11 Energy per sealed message for Ascon-AEAD128 and AES-128-GCM on the ESP32 DevKit (left) and the ESP32-C3 (right). Bars are means of five runs of 10,000 messages; error bars are one standard deviation

Fig. 11 draws the energy columns of Table 10. On the DevKit, for payloads of 8 and 15 bytes without associated data, sealing with Ascon costs 2.43 and 2.52 µJ against 3.51 and 3.65 µJ for AES-128-GCM, so Ascon saves 31% of the energy; with 32 bytes and 8 bytes of associated data the gap shrinks to 4.74 against 5.01 µJ, a saving of 5%. On the C3 the gap is much wider: 1.60 and 1.69 µJ against 4.64 and 4.65 µJ, a saving of 65% and 64%, and 3.11 against 6.42 µJ with associated data, a saving of 52%. The Ascon bars are lower on the C3 than on the DevKit (1.69 against 2.52 µJ for a 15-byte frame) although the C3 seals more slowly, whereas the AES-GCM bars are higher on the C3. The error bars are the standard deviation over five runs and are barely visible, at most 0.012 µJ, so the ordering is not in doubt on either board.

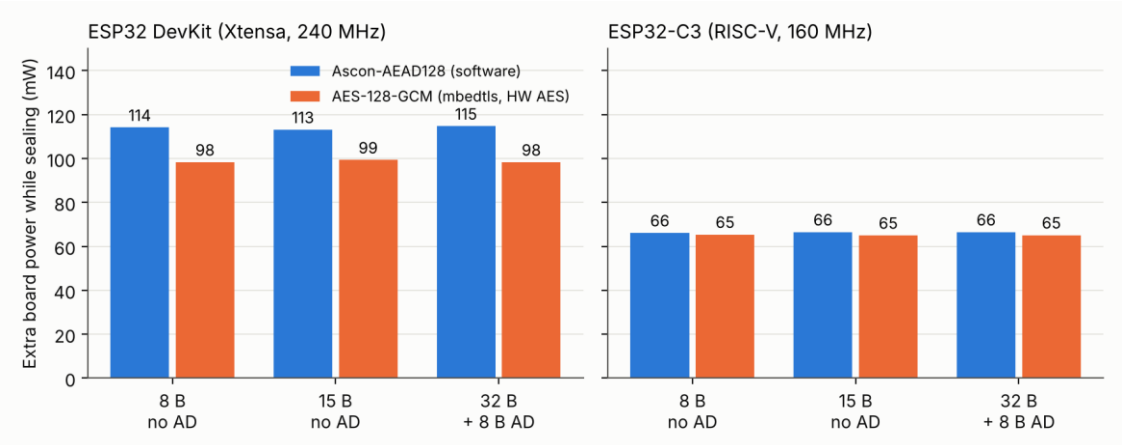

Fig. 12 Extra power drawn by the board above its waiting level while it seals, on the ESP32 DevKit (left) and the ESP32-C3 (right)

Fig. 12 shows why the energy gap differs from the time gap, and why it differs between the boards. Energy is the product of time and power. On the DevKit, the board draws about 113 to 115 mW above its waiting level with Ascon and about 98 to 99 mW with AES-GCM: the hardware AES block does part of the work, so the processor draws less during the call, but not enough to compensate for the longer call. For 15-byte payloads the time ratio of 1.65 in Table 8 becomes an energy ratio of 1.45, and for the 32-byte payload with associated data the time ratio of 1.23 becomes 1.06. On the C3 the two primitives draw almost the same extra power, about 66 mW for Ascon and 65 mW for AES-GCM, so the energy ratio follows the time ratio (2.75 against 2.82 for 15-byte payloads). The C3 draws about 40% less extra power than the DevKit while sealing, which is why its Ascon energy is lower although its Ascon time is longer.

On the DevKit, the total power of the board while it runs Ascon, about 354 mW, is close to the 353 mW that the board draws before and after each run, which we take to be the processor running without pause. The energy of Ascon is therefore set by its running time, and a faster implementation would lower it in the same proportion. The decision tree could not be measured on either board, because its window lasts 14 to 20 ms and holds only about 33 samples. With its times of 1.23 µs on the DevKit and 1.77 µs on the C3 (Table 7) and the power measured for Ascon, we estimate its cost at about 0.14 µJ and 0.12 µJ per prediction; these are estimates, not measurements. Sealing a 15-byte frame would then cost about 18 times more than classifying it on the DevKit and about 14 times more on the C3.

What the energy results do and do not say. The reference is a board waiting in a delay, not a board switched off; the values are therefore the extra energy of a node that wakes up, seals and waits again. They include the whole development board (voltage regulator and USB-serial chip) and the small loop that builds each message, as the timing does, so a bare chip would draw less in absolute terms, and the two boards, which carry different regulators and peripherals, should be compared through their ratios rather than their absolute values. The standard deviation is that of repeated runs on one board; it does not include the shunt tolerance or the gain error of the converter, which scale all values by a common factor of the order of one percent (the same INA219 module served for both boards) and leave the ratios unchanged. Finally, the radio was off: we did not measure it, and we expect it to

dominate the energy of a node that transmits the frame, so the saving matters most for nodes that seal many messages or have no hardware AES.

Memory footprint. Table 11 gives the flash and the static RAM that each component adds to a minimal sketch. Each figure is the difference between the sizes reported by the compiler (Arduino-ESP32) for a sketch that calls only that component and for a sketch that calls none. The figures are for the node side, that is, for sealing one 15-byte message; the Ascon decryption code, which runs on the gateway, is not counted.

Table 11 Flash and static RAM added by each component to a minimal sketch

Board	Component	Flash (bytes)	Static RAM (bytes)
ESP32 DevKit	Ascon-AEAD128	1,800	72
ESP32 DevKit	AES-128-GCM (mbedtls)	21,244	480
ESP32 DevKit	Decision tree	6,944	0
ESP32-C3	Ascon-AEAD128	2,096	64
ESP32-C3	AES-128-GCM (mbedtls)	24,918	600
ESP32-C3	Decision tree	6,962	0

Ascon adds about 1.8 kB of flash on the DevKit and 2.1 kB on the C3, roughly one twelfth of what AES-128-GCM adds through mbedtls (21.2 kB and 24.9 kB), and its static RAM is 72 and 64 bytes against 480 and 600 bytes. The AES-GCM figure covers the whole library path (the GCM mode, the cipher interface and the hardware AES driver); a GCM written by hand around the hardware block would be smaller, so this comparison, like the timing one, is between two specific implementations. The Ascon figure is for the build with function inlining and loop unrolling switched off. The decision tree adds 6.9 kB of flash, almost entirely its 6,839 bytes of constant tables, and no RAM, because the tables stay in flash. Together, the tree and the Ascon code take about 8.7 kB of flash on the DevKit and 9.1 kB on the C3, less than one percent of the 1.3 MB of program space of either board.

Still to measure. (1) The energy of the decision tree, whose window is too short for the INA219; a longer run of consecutive predictions would solve this. (2) ChaCha20-Poly1305, which the board package does not provide, so that Ascon is compared with AES-128-GCM only.

10 Discussion and limitations

What the numbers mean. Our headline macro-F1 is lower than many published figures for Edge-IIoTset, and the difference is deliberate. Several recent papers reach the same conclusion about shortcuts in IoT intrusion data [8, 9, 10]. Reporting the full ladder lets a reader choose the level they trust.

Limits. (1) Per-packet features cannot separate attacks that differ only over time, such as slow HTTP floods. (2) The source port and similar fields are attacker-controlled; a tree that depends on them can be evaded, which is why level C+ drops the source port but still uses the destination port. (3) The data come from two testbeds, each captured in a single campaign; on TON_IoT [15] the byte budget is easy to meet, but the records are highly redundant and the score depends on how the data are split, and we have no timestamps for a true split in time. (4) The security argument rests on the Ascon standard and on correct counter handling; we did not analyse side channels. (5) Energy was measured on both boards, relative to the waiting level of each board, with a 12-bit monitor and a shunt whose value we did not calibrate, so absolute values may carry an error of the order of the shunt tolerance (typically one percent) while the ratios do not; the energy of the decision tree is estimated from its time and not measured; and the timing comes from two boards and two implementations (a portable C Ascon and mbedtls AES-GCM). (6) Under a split that follows the order of the file, random forests and an MLP are more accurate than the single tree (Tables 5 and 6), so the case for the tree rests on its size and its cost per message, not on accuracy.

Related embedded work. Recent studies deploy small models on microcontrollers [11, 12, 14, 16, 17] and benchmark lightweight AEAD in isolation [13, 20]. We combine the two views under one byte budget, and we audit the dataset first.

11 Conclusion

We asked how much intrusion-detection accuracy survives once shortcuts are removed and the whole pipeline must fit a constrained node. At the honest feature level, a depth-12 tree reaches a macro-F1 of 0.831. The exported model, a 977-node tree stored in 6.8 kB, reaches 0.842 on 15 bytes of quantised input, with the destination port kept at 16 bits. A single Ascon-AEAD128 block carries that payload in a 35-byte authenticated frame. On a PC, the exported C code reproduces the

Python predictions exactly. On the boards, Section 9 shows that sealing one 15-byte frame with Ascon takes about 22 µs on the DevKit and 25 µs on the C3, and that the tree adds less than 8% to that; sealing a 15-byte frame costs about 2.5 µJ on the DevKit and 1.7 µJ on the C3, which is 31% and 64% less energy than AES-128-GCM with the hardware AES block; with a 32-byte payload and associated data the savings fall to 5% and 52%. The Ascon code needs about 2 kB of flash and the tree about 7 kB. On TON_IoT the same pipeline reaches 0.957 with 15 bytes under a random split, but only about 0.8 when the split follows the order of the file. A comparison on the same 15 bytes shows that the tree is far smaller than a forest at similar random-split accuracy, but that forests and an MLP generalise better when the split follows the order of the file. Next come a capture with timestamps for a true split in time, and window-level features.

Statements and Declarations

Funding. The authors received no specific funding for this work.

Competing interests. The authors have no relevant financial or non-financial interests to disclose.

Author contributions. Mohammed Taher Amine Sahraoui: conceptualization, methodology, software, investigation, data curation, writing – original draft. Ali Kourtiche: methodology, validation, supervision, writing – review and editing. Souaad Belhia: methodology, validation, supervision, writing – review and editing. Santiago Felici-Castell: methodology, validation, writing – review and editing.

Availability of data and materials. The datasets analysed during the current study are publicly available. Edge-IIoTset (file ML-EdgeIIoT-dataset.csv; Ferrag et al. [4]) is available from IEEE DataPort at https://doi.org/10.21227/mbc1-1h68 (https://ieee-dataport.org/documents/edge-iiotset-new-comprehensive-realistic-cyber-security-dataset-iot-and-iiot-applications) and from a Kaggle mirror at https://www.kaggle.com/datasets/mohamedamineferrag/edgeiiotset-cyber-security-dataset-of-iot-iiot. TON_IoT (network dataset, file Train_Test_Network.csv; Moustafa [15]) is available from the UNSW Canberra Cyber page at https://research.unsw.edu.au/projects/toniot-datasets. The measurement results generated during the study (time, energy and memory on ESP32 boards) and the

code (Colab notebooks and Arduino sketches) are available from the corresponding author on reasonable request.

Ethics approval and consent to participate. Not applicable.

Consent to publish. Not applicable. The study uses only publicly available, de-identified network-traffic datasets and involves no human participants, human tissues or animal subjects, and no identifiable images or personal data.

Use of artificial intelligence. See the statement “Use of language models” in Section 7.

References

[1] Sisinni E, Saifullah A, Han S, Jennehag U, Gidlund M. Industrial Internet of Things: challenges, opportunities, and directions. IEEE Transactions on Industrial Informatics. 2018;14(11):4724–34. https://doi.org/10.1109/TII.2018.2852491

[2] National Institute of Standards and Technology. Ascon-based lightweight cryptography standards for constrained devices: authenticated encryption, hash, and extendable output functions. NIST Special Publication 800-232. 2025. https://csrc.nist.gov/pubs/sp/800/232/final

[3] Dobraunig C, Eichlseder M, Mendel F, Schläffer M. Ascon v1.2: lightweight authenticated encryption and hashing. Journal of Cryptology. 2021;34(3):33. https://doi.org/10.1007/s00145-021-09398-9

[4] Ferrag MA, Friha O, Hamouda D, Maglaras L, Janicke H. Edge-IIoTset: a new comprehensive realistic cyber security dataset of IoT and IIoT applications for centralized and federated learning. IEEE Access. 2022;10:40281–306. https://doi.org/10.1109/ACCESS.2022.3165809

[5] Arp D, Quiring E, Pendlebury F, Warnecke A, Pierazzi F, Wressnegger C, et al. Dos and don'ts of machine learning in computer security. In: Proceedings of the 31st USENIX Security Symposium. 2022. p. 3971–88. https://www.usenix.org/conference/usenixsecurity22/presentation/arp

[6] Geirhos R, Jacobsen J-H, Michaelis C, Zemel R, Brendel W, Bethge M, et al. Shortcut learning in deep neural networks. Nature Machine Intelligence. 2020;2(11):665–73. https://doi.org/10.1038/s42256-020-00257-z

[7] Kapoor S, Narayanan A. Leakage and the reproducibility crisis in machine-learning-based science. Patterns. 2023;4(9):100804. https://doi.org/10.1016/j.patter.2023.100804

[8] Reddy DS, Kumar KA. A dataset-centric review of IoT and IIoT intrusion detection: realism, evaluation biases, and future research directions. Frontiers in Big Data. 2026;9:1878260. https://doi.org/10.3389/fdata.2026.1878260

[9] Roy US, Minu MJ. Unmasking shortcut learning in IoT intrusion detection: a forensic, multi-paradigm evaluation of feature dependence and data leakage. arXiv:2609.28725. 2026. https://arxiv.org/abs/2609.28725

[10] Galal MM. Provenance, not behaviour: a serialisation artifact in Edge-IIoTset and a leakage-free benchmark for precision-agriculture intrusion detection. arXiv:2608.15761. 2026. https://arxiv.org/abs/2608.15761

[11] Diab A, Chehade A, Ragusa E, Gastaldo P, Zunino R, Baghdadi A, et al. Intrusion detection on resource-constrained IoT devices with hardware-aware ML and DL. In: IEEE ETECOM 2025. 2025. arXiv:2512.02272

[12] De Bernardis EP, Kuznetsov O, Arnesano M, Zhansaya P, Sydykova M. Lightweight machine learning intrusion detection for IoT/IIoT networks: quantisation strategies and physical deployment on resource-constrained microcontrollers. Electronics. 2026;15(13):2869. https://doi.org/10.3390/electronics15132869

[13] Atul, Shukla V, Das MK, Shukla V, Mishra D. A reproducible security–performance benchmark protocol for lightweight AEAD on resource-constrained IoT nodes. IACR Cryptology ePrint Archive, Report 2026/1882. 2026. https://eprint.iacr.org/2026/1882

[14] Kumar A, Goyal S, Varma M. Resource-efficient machine learning in 2 KB RAM for the Internet of Things. In: Proceedings of the 34th International Conference on Machine Learning, PMLR 70. 2017. p. 1935–44.

[15] Moustafa N. A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets. Sustainable Cities and Society. 2021;72:102994. https://doi.org/10.1016/j.scs.2021.102994

[16] Tekin N, Acar A, Aris A, Uluagac AS, Gungor VC. Energy consumption of on-device machine learning models for IoT intrusion detection. Internet of Things. 2023;21:100670. https://doi.org/10.1016/j.iot.2022.100670

[17] Ieropoulos V, Anthi E, Spyridopoulos T, Burnap P, Khan A, Carnelli P. Empowering IoT security: on-device intrusion detection in resource constrained devices. arXiv:2605.13159. 2026. https://arxiv.org/abs/2605.13159

[18] Turan MS, McKay K, Chang D, Bassham L, Kang J, Waller N, et al. Status report on the final round of the NIST lightweight cryptography standardization process. NIST IR 8454. National Institute of Standards and Technology; 2023. https://doi.org/10.6028/NIST.IR.8454

[19] Thakor VA, Razzaque MA, Khandaker MRA. Lightweight cryptography algorithms for resource-constrained IoT devices: a review, comparison and research opportunities. IEEE Access. 2021;9:28177–93. https://doi.org/10.1109/ACCESS.2021.3052867

[20] Vaz YS, Mattos JCB, Soares RI. Lightweight cryptography for the Internet of Things: a multi-metric experimental assessment. In: Anais do XV Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC). Sociedade

Brasileira de Computação; 2025. p. 91–6. https://sol.sbc.org.br/index.php/sbesc/article/view/40345

[21] Sokolova M, Lapalme G. A systematic analysis of performance measures for classification tasks. Information Processing & Management. 2009;45(4):427–37. https://doi.org/10.1016/j.ipm.2009.03.002

[22] Kraskov A, Stögbauer H, Grassberger P. Estimating mutual information. Physical Review E. 2004;69:066138. https://doi.org/10.1103/PhysRevE.69.066138

[23] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: machine learning in Python. Journal of Machine Learning Research. 2011;12:2825–30. https://jmlr.org/papers/v12/pedregosa11a.html

[24] Booij TM, Chiscop I, Meeuwissen E, Moustafa N, den Hartog FTH. 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. 2022;9(1):485–96. https://doi.org/10.1109/JIOT.2021.3085194

Appendix A Reproducibility

- **Data cleaning.** Empty-like values (“”,”0”, “0.0”, “0.0.0.0”, “nan”, “NaN”, “None”) are mapped to a single token before encoding and before deriving text-length features. The bookkeeping set is tcp.checksum, icmp.checksum, tcp.seq, tcp.ack, tcp.ack_raw, udp.stream and icmp.seq_le.
- **Level C+ indicators.** icmp.checksum__present, icmp.seq_le__present and udp.stream__present.
- **Tree export.** Seven bytes per node: feature index (uint8), threshold, left child and right child (uint16 each). A leaf has feature index 255 and stores the class in the threshold field. Inference tests x <= floor(threshold) on integer inputs.
- **Code.** The Colab notebooks (training, audit, C export) and the Arduino sketches are available from the corresponding author on request.
- **Seeds.** Five seeds for level comparisons, three for sweeps.