 ISSN:2302-8149
e-ISSN:2540-9719

A Lightweight NAS-Optimized Temporal Convolutional Network (TCN) for MQTT Brute-Force Detection in Resource-Constrained IoT Networks

[1]Mahmood Mohammed Mahmood*
[1]Department of Computer Science, College of Computer Science and Mathematics, University of Mosul, Mosul, Iraq
*e-mail: mahmood_mohammed@uomosul.edu.iq

(***received*****:** 15 October 2026, ***revised*****:** 11 November 2026, ***accepted*****:** 12 November 2026)

Abstract

It is worth pointing out that there is little inbuilt protection against brute force attacks on the authentication step in the MQTT protocol, which is widely used in IoT systems. According to published reports, accuracy is high; however, we are not aware of any reporting model size or latency and ones that test on independently collected data. We created a lightweight temporal convolutional network (TinyTCN) for window-based bidirectional flow record classification for a single host and reduced its size through architecture search with Optuna. For MQTT-IoT-IDS2020, we windowed the data at the host level (and for a single attacker, the file order), and rectified incorrectly classified non-MQTT traffic that MQTT-IoT-IDS2020 benchmarked as attacks. The number of parameters in the model was reduced from 13,729 to 1,129 by search. It requires 11.66 KB with 8-bit weight quantization, and processes 8 windows in 0.0123 ms, which is no different than the test performance (F1 = 1.00, AUC-ROC = 1.00 on test set; 5-fold F1 = 0.9995 ± 0.0012). Ablation demonstrated the importance of using multi-component features involving both connection volume and a single timing feature for decision making. A unidirectional (Uni-flow) variant further reduced the size of the model to 817 parameters, and its performance was good on all the folds of the cross-validation set, but one broker was misclassified in one fold.This model performed poorly when applied to MQTTset without retraining. The cross-dataset evaluation on MQTTset demonstrated that direct generalization is challenging, and that the detection performance is affected by the characteristics of network traffic.

Keywords: brute-force attack detection, cross-dataset generalization, data leakage, edge deployment, Internet of Things, MQTT, Neural Architecture Search, Temporal Convolutional Network

1. Introduction

Real-time monitoring, automation, and enhanced decision-making skills have been made possible by the Internet of Things (IoT) across several industries. IoT technologies, for instance, have greatly increased the productivity and efficiency of healthcare, smart homes, smart cities, and even industrial processes [1].

MQTT is an open standard messaging protocol for telemetry information delivery that is lightweight, straightforward, and data-centric. It is intended to achieve reliability while being appropriate for constrained environments with varying levels of latency, bandwidth constraints, and unreliable networks for resource-constrained devices. It operates on top of the TCP/IP stack or over other network protocols that provide ordered, lossless, bidirectional connections [2].

Its topic-based architecture, like that of other publish/subscribe protocols, classifies shared data according to hierarchically arranged themes such that each message is linked to a topic. Selective messaging occurs when clients publish messages on a specific topic, which are then received by all other clients that have subscribed to this subject. Only destinations with similar topic interests receive messages after that [3]

Because the MQTT was created mainly for use in secure, air-gapped, and closed IoT networks. As a result, during its creation, cybersecurity was not a major worry. Therefore, it is essential to guaranty that the MQTT protocol is safe from cyberattacks. The great majority of this protocol lacks security measures and is vulnerable to attacks as it was created with performance as its primary

 ISSN:2302-8149
e-ISSN:2540-9719

objective. Researchers are developing ways to safeguard the MQTT protocol for the purpose of secure communication within the Internet of Things [4].

Port scanning (OS), HTTP flooding, ACK flooding, man-in-the-middle attacks, sniffing, buffer overflow, replay attacks, pre-shared key attacks, brute force authentication, Slow DoS against Internet of Things Environments (SlowITe), malformed data, distributed denial-of-service (DDoS) attacks, and MQTT publish flood attacks are just a few of the attacks that can affect it [5].

Attempting to gain user data that MQTT uses is known as a brute force attack [6]. Brute-force assaults can be executed in a variety of methods, including by using trial and error. Hackers test every conceivable combination in an attempt to make an accurate estimate [7].

In order to create Intrusion Detection Systems (IDSs) that can recognize cyberattacks automatically, researchers have been using machine learning (ML) and deep learning (DL) techniques more and more in recent years. For brute-force assaults against the MQTT protocol, a number of models, including the Random Forest method, have shown detection accuracy over 99% [8]. However, most previous research has concentrated on optimizing detection accuracy while paying little attention to important deployment factors, such as model size, inference latency, and the viability of using these models on Internet of Things (IoT) devices with severely limited computational and energy resources. This restriction is a major barrier to the practical application of such solutions.

In IoT applications, it is typically not desired to perform computationally demanding and time-sensitive security activities. Therefore, an effective detection mechanism is urgently needed to maintain the security and effectiveness of IoT services deployed on edge [9].

However, due to restricted processing power and memory resources, implementing deep learning (DL) based security solutions on edge devices continues to be difficult. While frequent data transfers between memory and the CPU increase latency and power consumption, DL models are resource-intensive and may surpass device capability [10].

The current study suggests using Temporal Convolutional Networks (TCNs), which have been demonstrated in recent studies, to close this gap [11] to model sequential data more effectively than Long Short-Term Memory (LSTM) and recurrent neural networks (RNNs), especially in terms of parallelization efficiency and training stability. Because of these features, TCNs are especially good at collecting the unique temporal patterns of repeated connection attempts that define brute-force attacks.

TCNs alleviate the memory bottleneck on embedded platforms by buffering feature maps with a lot less memory than LSTMs. As a result, TCNs are being used more and more in many application scenarios where data classification is closely related to its temporal characteristics [12].

Because the TCN model is lightweight, quantization minimizes resource use while maintaining accuracy by reducing precision from FP32 to lowerbit representations (e.g., INT8). In deep learning and edge deployment, INT8 quantization is a commonly used standard, especially for TensorFlow Lite (TFLite) and ARM-based devices [13], [14].

Neural architecture search is increasingly being used to automate the time-consuming architecture design stage, just as automated feature learning through deep learning has recently supplanted manual feature engineering. The method of automating the construction of neural architectures for a particular job, known as neural architecture search (NAS), has already surpassed the best human-designed designs on numerous tasks [15].

In a number of areas, including model performance and inference delay on the real hardware, the resulting searched models perform better than manually created network [16].

The following three elements usually make up a NAS method: (1) Search space, a manually constructed primitive operation set and structure, either layer-wise or cell-wise, (2) Search Strategy, or the fundamental search method that searches the search space for a network; (3) Evaluation phase, in which the performance of a predicted network is assessed [17]

Neural Architecture Search (NAS) is used to automate the optimization of important architectural hyperparameters, such as the dilation rate and the number of channels, in order to circumvent the limitations of manual and heuristic network design. This process allows for the identification of a compact TinyTCN architecture that minimizes model complexity while maintaining high detection performance, utilizing lightweight NAS methodologies specifically designed for effective deployment of TCN-based models on resource-constrained edge devices [18].

 ISSN:2302-8149
e-ISSN:2540-9719

In order to meet the identified research gaps and to critically assess the proposed approach, the following research questions are raised:

RQ1. Will a small model that can be searched with architecture search match a big seed model in MQTT brute force detection, and be sufficiently small and fast for a gateway-class device?

RQ2. How to partition, window and label the data to ensure that reported scores do not include leakage or that the benchmark is not misrepresented by labelling errors?

RQ3. What does bidirectional flow aggregation have to offer besides what accuracy demonstrates?

RQ4. Does information gathered by a detector on one testbed transfer to a testbed gathered independently?

The following are the key contributions of this study for answering the research questions:

1-We run architecture search and save the seed TCN from 13,729 to 1,129 parameters with the same classification performance reported. The quantized model has a size of 11.66 KB and the median inference latency is 0.0123 ms [update after re-measurement] (RQ1).

2-We propose a leakage-aware protocol where benign traffic is divided per host and the single attacker's traffic per file order before windowing, windows are labelled per majority, and the scaler is refitted per cross validation fold (RQ2).

3-We audit the benchmark and look into mislabeled non-MQTT traffic as attacks. Correcting it eliminates a class of errors that was responsible for most errors prior to correction (RQ2).

4-We compare Bi-flow and Uni-flow representations, and find that Uni-flow allows a smaller model (only 817 parameters); we also find that when a broker is not present in the training set, there is a misclassification in a single cross-validation fold (RQ3).

5-We test the proposed model on MQTTset after not training it to study cross-dataset generalization and examine the influence of the attacker connection strategy on the predictive behavior of traffic-volume features (RQ4).

The rest of this paper is organized as follows: Section II presents previous related work. Section III describes the methodology. Section IV describes the results. In Section V, the discussions are explained. Section VI describes the limitations. Finally, the conclusions and suggestions for future work are presented in Section VII.

2. Related Work

Since of the increasing proliferation of IoT devices and the resulting increase in security concerns, the topic of IoT security is garnering substantial attention from academics around the globe. This section is a quick description of recent research focused on boosting security in IoT, specifically the security of the MQTT protocol in IoT environments. Table 1 briefly summarizes the relevant studies carried out in this domain.

[19] Al Hanif and Ilyas proposed an ensemble-based intrusion detection system targeting both denial-of-service and brute-force attacks in MQTT traffic. Their methodology applied three feature-selection techniques K-Best, Pearson Correlation Coefficient, and Principal Component Analysis to reduce the MQTTset dataset's original 33 features to the ten most discriminative attributes; four supervised classifiers (Decision Tree, Random Forest, KNN, and XGBoost) were combined through stacking, voting, and bagging ensembles, with stacking and voting achieving the highest accuracy of 95.38%, outperforming two prior studies that used the same dataset without feature selection or cross-validation. This systematic multi-technique feature-selection methodology and the positive comparison with existing baselines represents a methodological strength but the study is based only on classical machine-learning models while the feasibility of real-time deployment on resource constrained IoT edge devices is left entirely in the air for brute-force detection approaches.

To investigate the impact of flow granularity on attack detection, which is not a subject of most of the literature surveyed, [20] Otoom et al. introduced an approach in the literature based on deep learning that they tested on a dataset they introduced, called MQTT-IoT-IDS2020, to detect brute-force attacks in MQTT-IoT networks, using a benchmark similar to the one used in the present study, and evaluated their model for two levels of abstraction; Bi-flow and Uni-flow. This granularity analysis is an interesting contribution, but the specific deep-learning architecture used is not explicitly

 ISSN:2302-8149
e-ISSN:2540-9719

reported and there is no complexity analysis, deployment latency or memory analysis of the model, so it remains unclear whether it can be deployed on constrained IoT hardware.

[21] Ullah et al. presented a Transformer-based deep learning model, TNN-IDS, which utilizes the self-attention mechanism to learn the contextual dependencies of the MQTT network traffic data, tested on the MQTT-IoT-IDS2020 dataset on three feature abstraction levels, and attained accuracy of up to 99.9%. The self-attention mechanism offers a principled approach to capturing long-range contextual dependencies in traffic sequences directly relevant to sequential brute-force attack patterns representing genuine architectural strength. However, Transformer architectures are computationally heavy relative to the resource budgets typical of IoT edge devices, and the study does not report model size, parameter count, or inference latency, leaving the real-time edge-deployment feasibility of this otherwise high-performing approach entirely unaddressed.

[22] Omotosho et al. proposed IDS-MA, an intrusion detection system for MQTT-based IoT networks comparing centralized and federated learning approaches, explicitly motivated by the observation that prior MQTT-IDS research often covered a limited attack scope, overlooked memory-corruption attacks, or relied on accuracy alone as an evaluation metric. The federated-learning configuration offers genuine privacy advantages by avoiding centralized raw-data aggregation as a practically relevant consideration for real-world multi-organization IoT deployments and the explicit critique of accuracy-only evaluation reflects methodological awareness shared by the present study. However, the work does not clearly or specifically target brute-force authentication attacks as a distinct detection objective within its evaluated attack scope, and, consistent with every other study reviewed, reports no model size, inference latency, or energy consumption, nor evaluates deployment feasibility on resource-constrained IoT edge hardware.

[8] Akbar et al. proposed a Random Forest-based approach for detecting brute-force attacks on the MQTT protocol, motivated by the algorithm's computational efficiency and resistance to overfitting on large data volumes, and notably combined primary data collected from a controlled hacking-lab environment with the secondary MQTT-IoT-IDS2020 dataset partially mitigating the single-source-attacker limitation common to several benchmark-only studies in this review (see Section 6, Limitation 1, of the present work) and constituting a meaningful methodological strength. However, as a classical ensemble method operating on hand-crafted, per-instance statistical features, Random Forest lacks the capacity to automatically learn temporal dependencies across sequential connection attempts, and the study reports neither model size nor inference latency nor deployment feasibility on constrained IoT hardware.

[23] Chesney and Roy evaluated seven machine learning and deep learning algorithms Random Forest, Logistic Regression, Decision Tree, KNN, SVM, a Deep Neural Network, and a Convolutional Neural Network for binary intrusion detection on the MQTT-IoT-IDS2020 dataset, using Random Forest-based feature selection to reduce the dataset to five key features; most classifiers achieved near-perfect accuracy (> 99%) on brute-force traffic specifically, though the DNN and CNN models notably collapsed on the Aggressive Scan attack type (≈ 63% accuracy). This is the most methodologically comprehensive of the reviewed studies, providing per-attack-type performance breakdown that reveals heterogeneity a single aggregate accuracy figure would obscure, and the authors explicitly acknowledge this limitation, recommending future evaluation with sequential architectures (RNNs, LSTMs) a recommendation directly motivating the temporal-convolutional approach adopted in the present study. However, the DNN and CNN architectures employed are shallow (three to four layers, with no dilation or explicit temporal-receptive-field design) and were trained for only ten epochs; flow instances are classified independently rather than as part of a temporal sequence, discarding potentially informative sequential context across repeated connection attempts; no model-size or latency analysis is provided; and random instance shuffling prior to K-fold cross-validation raises the possibility of temporal or instance-level leakage effects not explicitly addressed by the authors.

[24] However, Mosaiyebzadeh et al. proposed a hybrid CNN-RNN-LSTM based deep learning network intrusion detection system (NIDS) for MQTT enabled IoT networks and evaluated its performance on the MQTT-IoT-IDS2020 dataset achieving 97.09% accuracy and 98.33% F1 score.The explicit incorporation of temporal sequence modeling distinguishes this study from the majority of reviewed work, which treats flow instances as independent, non-sequential samples, and

 ISSN:2302-8149
e-ISSN:2540-9719

the strong F1-score indicates balanced precision–recall performance. Nonetheless, combining three distinct deep architectures substantially increases computational cost and model complexity relative to a single lightweight architecture, and the study reports no model size, inference latency, or energy consumption, nor any assessment of feasibility for resource-constrained edge deployment.

Table 1 Summary of related work

Study	Method	Dataset	Attack Scope	Temporal Modeling	Model Size Reported	Latency Reported	Edge Deployment Evaluated
Al Hanif & Ilyas (2024) [19]	Ensemble ML (Stacking / Voting / Bagging: DT, RF, KNN, XGBoost)	MQT Tset	DoS Brute Force	✗	✗	✗	✗
Otoom et al. (2023) [20]	Deep Learning (architecture unspecified)	MQT T-IoT-IDS2020	Brute Force	Unconfirmed	✗	✗	✗
Ullah et al. (2023) [21]	Transformer (self-attention)	MQT T-IoT-IDS2020	Brute Force Aggressive scan UDP scan Sparta	✓	✗	✗	✗
Omotosho (2023) [22]	Centralized vs. Federated Learning	Custom MQTT-IoT testbed	Brute Force Aggressive scan UDP scan,Sparta MitM ,DoS Buffer Overflow	Unconfirmed	✗	✗	✗
Akbar et al. (2023) [8]	Random Forest	Primary (hacking-lab) + MQTT-IoT-IDS2020	Brute Force	✗	✗	✗	✗
Chesney & Roy (2022) [23]	RF / LR / DT / KNN / SVM / DNN / CNN	MQT T-IoT-IDS2020	Brute Force Aggressive scan UDP scan Sparta	✗	✗	✗	✗
Mosaiyebzadeh et al. (2021) [24]	CNN-RNN-LSTM (hybrid)	MQT T-IoT-IDS2020	Brute Force Aggressive scan UDP scan	✓	✗	✗	✗

 ISSN:2302-8149
e-ISSN:2540-9719

This work	**TinyTCN + NAS (Optuna)**	**MQTT-IoT-IDS2020**	**Brute Force**	✓	✓	✓	✓

3 Methodology

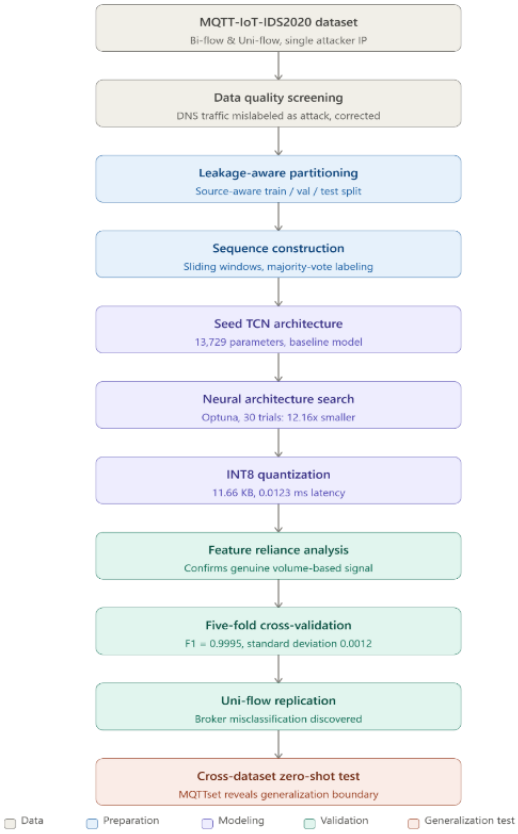


Figure 1 pipeline overview

This section outlines the full experimental pipeline used in this work: dataset characterization, a mandatory data-quality audit before any model can be trusted, the partitioning scheme used to ensure that evaluation was conducted in an honest manner, the construction of sequences to model the temporal aspects of the data, creation of the seed architecture and its subsequent compression with the tool of Neural Architecture Search (NAS), quantization to deploy to the edges, and the multiple validation procedures used to question and explore the resulting models and not merely report their score. The overall pipeline is shown in Figure 1.

3.1 Dataset and Flow Representations

The primary dataset used throughout this work is MQTT-IoT-IDS2020 [25], The simulated IoT setup involved twelve MQTT sensors, a broker, a camera, and an hostile node. Data collection took place at two levels, both of which were used in the analysis. At the time we couldn't predict which one would be more effective and efficient in terms of accuracy. The Bi-flow representation separates each TCP connection into client-to-broker and broker-to-client flows. It provides 29 metrics per direction including packet counts, arrival rates, sizes, total bytes, and TCP flags. Conversely, the Uni-flow representation treats each connection direction as an independent entry, yielding 16 undirected attributes. This translates to fewer rows for the same traffic for Uni-flow. For example, there were 171,836 normal rows and 33,079 attack rows with Bi-flow, compared to 86,008 normal and 16,696

 ISSN:2302-8149
e-ISSN:2540-9719

attack rows with Uni-flow. In Bi-flow, each connection is duplicated as two in Uni-flow. One of the striking features of the data description and confirmed by a review of the IP addresses is that all of the brute force attempts were from one machine: 192.168.2.5. This is a rather significant event. It impacted on how we arranged the partitioning, restricted what we can say about the performance of the model against different attackers and as we will see, it is relevant to the behavior of the flow representations. In Uni-flow the broker appears as a traffic source, rather than in Bi-flow.

3.2 Data-Quality Screening and Correction

The labels for the dataset were not assumed. Before training any model, we examined the distribution of all features among each class, as a few inter-arrival-time features appeared to be separating benign and attack traffic too well. This kind of outcome typically should be revisited and not celebrated. The issue that we identified was a genuine issue that came from the manner in how the benchmark itself has been constructed. All packets from the attacker's IP address were marked "attack," regardless of the protocol they were part of. But a lot of those rows — it turned out — were simply regular DNS queries on port 53, traffic the attacking machine was generating as part of background activity, no different from what it might have been doing. The amount of this contamination was not constant across the session. The attacker traffic registered only very little around 0.04% of attacker rows in the beginning of the attack. That rose to 19.4% by the segment where we did testing later. We only confirmed the connection between this contamination and the model's errors afterward: of the 433 test windows the uncorrected model misclassified, 82 (18.9%) lined up almost exactly with this DNS-tagged subset. Two numbers computed completely independently landing that close together is not something we are comfortable calling a coincidence. The solution implemented was very simple: the positive label was applied only to genuine MQTT traffic (destination port 1883) and the same filter was being applied to both the Bi-flow and the Uni-flow datasets. That left the attack class as 14,114 instances (Bi-flow) and 14,115 instances (Uni-flow), which were close together enough that the two independently aggregated views were a good sanity check that the filtering logic was operating in the same way. We consider this correction to be a substantive contribution to the study not a preliminary housekeeping step because it is not known if a similar correction had been done previously with this same benchmark.

3.3 Leakage-Aware Data Partitioning

 ISSN:2302-8149
e-ISSN:2540-9719

```
    ================================================================
    Algorithm 1: Source-Aware, Leakage-Preventing Data Partitioning
    ================================================================
    Input: D raw flow dataset
    a attacker IP address
    r_tr, r_val, r_te split ratios (e.g. 0.70, 0.15, 0.15)
    s random seed
    Output:  D_train,  D_val,  D_test  (no  host  overlap  across
partitions)

    1: D_normal ← { x ∈ D : x.label = 0 }
    2: D_attack ← { x ∈ D : x.label = 1 and x.ip_src = a }
    3:
    4: # Step 1 — partition benign traffic by host identity
    5: H ← unique(D_normal.ip_src)
    6: H ← shuffle(H, seed = s)
    7: H_tr, H_val, H_te ← split(H, r_tr, r_val, r_te)
    8: for p in {tr, val, te} do
    9: D_normal[p] ← { x ∈ D_normal : x.ip_src ∈ H_p }
    10: end for
    11:
    12:  #  Step  2  —  partition  the  single  attacker's  traffic
chronologically
    13: D_attack ← sort(D_attack, by = timestamp, order = ascending)
    14: D_attack[tr], D_attack[val], D_attack[te] ←
    chronological_split(D_attack, r_tr, r_val, r_te)
    15:
    16: # Step 3 — merge partitions and verify no leakage
                    17: for p in {tr, val, te} do
    18: D_p ← D_normal[p] ∪ D_attack[p]
    19: end for
    20: assert (H_tr ∩ H_val = ∅) and (H_tr ∩ H_te = ∅) and (H_val ∩
H_te = ∅)
    21:
```

Prior to performing any train/validation/test split that can be believed, there were two distinct leakage risks that needed to be addressed, but it was not a general problem. First, by construction, the windows are already highly overlapping, so splitting after the windows would leave nearly duplicate windows on either side of the boundary for the model to be evaluated on material that it has already seen. Second, if attack traffic originates from one host, and benign traffic from more than a dozen, an unconstrained random split can lead to the benign traffic fingerprint being distributed across partitions, while the benign traffic from a given host is spread across partitions indicating 'over-generalization'. Both of these problems were addressed by dividing the data.They both were fixed by dividing the data. We used source-based logic for normal traffic. We shuffled the benign host IPs randomly, and then split them into training (70%), validation (15%), and test (15%) sets. The important thing was to ensure that no host's traffic goes divided between sections. The attacking traffic is unique, as only one source exists. We were unable to divide it in the same manner. Rather, we divide it according to time. The first 70% were used for training, the next 15% for validation and the remaining 15% for testing. This split by time is nice because it doesn't leak.

This also involves testing the model on unseen data, which is more realistic as it would be used in a real-life situation, and is done in a forwards-looking manner. The quantities required for each component is given in Table 2.

Table 2 Partition sizes and attack-instance count

Partition	Raw flow rows	Attack rows
Train	62,650	9,879

 ISSN:2302-8149
e-ISSN:2540-9719

Partition	Raw flow rows	Attack rows
Validation	15,412	2,117
Test	22,060	2,118

3.4 Sequence Construction and Window Labeling

The rows were staggered 20 steps long with a distance of 5. between the rows. This equated to a 75% overlap of windows. These windows were hand-made for each host and partition. In this manner, no windows jumped or spanned a partition. This maintained the leakage guarantees and traffic actual

```
    Algorithm 2: Sliding-Window Sequence Construction with
    Majority-Threshold Labeling
    ================================================================
=
    Input: D_p one partition (train, val, or test)
    T window length (e.g. 20)
    sigma stride (e.g. 5)
    theta labeling threshold (e.g. 0.5)
    phi feature scaler, fit on D_train only
    F list of feature columns
    Output: X_p, Y_p sequence set and corresponding labels

    1: X_p ← ∅ , Y_p ← ∅
    2: H_p ← unique(D_p.ip_src)
    3:
    4: for each host h in H_p do # build windows per host
    5: G_h ← { x ∈ D_p : x.ip_src = h }
    6: G_h ← sort(G_h, by = timestamp, order = ascending)
    7: if |G_h| < T then
    8: continue # not enough rows for one window
    9: end if
    10: Z_h ← phi.transform(G_h[F]) # scale features, no refitting
    11: for i = 0, sigma, 2*sigma, ... while (i + T) <= |G_h| do
    12: w ← Z_h[i : i+T] # feature window
    13: l ← G_h.label[i : i+T] # timestep labels in window
    14: rho ← (1/T) * sum(l_j) # fraction of attack timesteps
    15: if rho >= theta then
    16: y ← 1
    17: else
    18: y ← 0
    19: end if
    20: X_p ← X_p ∪ { w }
    21: Y_p ← Y_p ∪ { y }
    22: end for
    23: end for
    24:
    25: return X_p, Y_p
```

for each host over time defined in Section 3.3. It was not so easy to label these windows. They first did this by assigning a window to be positive if it was marked as having any timestep marked as an attack. This had ruined the data set quite a lot. With the stride, one attack would appear in approximately four overlapping windows, due to the stride. So, the ‘any' rule really confused the percentage of attack windows with the actual 16%. It has risen to 97% which equals a nearly 180% change.

So, they adopted a majority threshold rule. An attack was only added to the "positive" total if it occurred half or more of the window's steps. This brought the window proportions back close to the original instance level numbers (Table 3). Feature standardization was done using the scaler fitted on just the training partition and then reused without refitting on validation and test data.

 ISSN:2302-8149
e-ISSN:2540-9719

Table 3 Sequence-level dataset composition (Bi-flow)

Partition	Sequences	Benign windows	Attack windows	Attack proportion
Train	12,495	10,523	1,972	15.8%
Validation	3,072	2,652	420	13.7%
Test	4,398	3,978	420	9.5%

Class imbalance was addressed at the loss-function level through inverse-frequency class weighting rather than through undersampling, a choice made after observing that undersampling — tried initially — discarded a large share of the already scarce attack windows without any corresponding benefit. Weights of approximately 0.59 (benign) and 3.17 (attack) were applied during training.

3.5 TinyTCN Architecture: Mechanism and Design Rationale

It is important to note that the term "TinyTCN" throughout this paper represents two separate design considerations, not just one. The first is a compact seed architecture, designed from the ground up with much fewer channels and layers than those of traditional TCN designs (detailed below). The second is the extra parameter reduction that can be achieved later via Neural Architecture Search (Section 3.6) which only explores architectures within a search space defined by the number of parameters of the initial model. The compactness of the resulting model, rather, is the result of both phases; not the compactness of the search procedure itself.

There is a point to be made here before the specifics of the seed configuration. The architecture underlying temporal convolutional networks is different in how it processes a sequence than the flow-independent classifiers discussed in Section 2 because it is a meaningful choice for this task.It is worth showing here why the underlying architecture of a temporal convolutional network is different for processing a sequence than the classification models reviewed in Section 2.

3.5.1 Causal Dilated Convolution

Although a standard 1D convolution can move a fixed size filter over a sequence, in its standard implementation it could use data from future and past time steps relative to a given time step of the output, something that is fine for offline signal processing and not desirable for a real-time detector where traffic after the current time step is, by definition, not available. TinyTCN, on the other hand, employs causal convolutions, where the output at time step t only depends on the input at time step t and prior steps. The value (at step t) of an output channel m is obtained by calculating:

$$y(t, m) = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} x(l, t - d \times i) \cdot W(l, m, i)$$

In words: This is the sum of the input values at each of the K locations in the filter W, from all the Cin input timesteps, but — critically — each input timestep appears in the sum at a different location, starting from the offset d times i, where d is a non-negative integer corresponding to the number of timesteps in a window. The computation is always performed subtractively, meaning there is no computation to be done forward in time from t, and this makes the convolution causal, by construction.

It includes the filter (kernel) size K, the number of input channels Cin, the filter weights W and the dilation rate d — an integer fixed between filter sizes which indicates a step added between the timestep that the filter is reading. For d = 1, the filter will read consecutive timesteps as in a regular convolution; for d = 2, it will read every other timestep, skipping the timestep in between; and so on for larger dilation values.

 ISSN:2302-8149
e-ISSN:2540-9719

3.5.2 Why Dilation Matters Here

The purpose of dilation is to let the network "see" further back in time without a proportional increase in parameters or computation. The receptive field of a single dilated convolutional layer — the span of input timesteps that can influence one output value — is given by:

$$F = (K - 1) \times d + 1$$

For example, with a filter size of K = 3 and a dilation rate of d = 4, the receptive field is (3 − 1) × 4 + 1 = 9 timesteps: a single layer with this configuration already looks back across 9 consecutive timesteps, despite having only 3 actual filter weights along the time axis.

Stacking several such layers with progressively doubling dilation rates (d = 1, 2, 4, 8, and so on) grows the overall receptive field of the network exponentially with depth, while the number of parameters grows only linearly. This is precisely why TCNs are considered well suited to tasks where a decision must integrate information across a moderately long window of past events using few layers: for brute-force detection specifically, a single connection attempt is rarely informative on its own, and what actually distinguishes an attack is the pattern of repeated attempts across the window — exactly the kind of dependency a wide receptive field is designed to capture, and exactly what a model classifying individual flow instances independently (Section 2) cannot represent at all.

Jointly considering all four layers of the seed architecture (Section 3.6), with a filter size of 3 and dilation rates of 1, 2, 4 and 8, the receptive field of the combined layers is approximately 31 timesteps, which covers (and slightly exceeds) the full 20-timestep input window of Section 3.4. It is a deliberate design property which means there is no part of the 20-step context that is structurally unreachable by the network's last representation. The NAS-optimized architecture (Section 3.7), with only two dilated layers (d = 1 and d = 2) and a smaller kernel, has a correspondingly narrower receptive field — a trade-off the search was free to explore, and one whose lack of any resulting accuracy cost (Section 4.2) suggests the task's discriminative signal is concentrated in a shorter temporal span than the seed architecture was originally designed to accommodate.

3.5.3 Full Forward Pass

Concretely, a single 20-timestep, 29-feature input window (Bi-flow; 16 features for Uni-flow, Section 3.10) is processed as follows. Each of the stacked causal dilated convolutional blocks applies its filters across the sequence dimension, producing a same-length output with a channel count equal to that block's filter count; batch normalization and dropout are applied after each block, in that order, before the result is passed to the next block. The last convolutional block is followed by a global average pooling layer which reduces the temporal dimension by averaging over 20 timesteps of each channel, yielding a single fixed length vector, independent of the number of convolutional blocks before it. This vector is fed into one dense layer, followed by another dense layer with a ReLU activation, and then through a single sigmoid unit which generates the predicted probability that the window is a brute force window. This is shown schematically for the seed configuration in Figure 2.

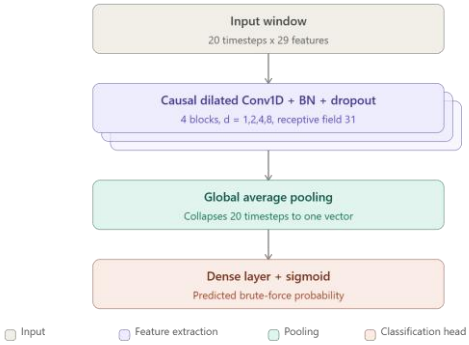


Figure 2 schematic diagram of the TinyTCN

 ISSN:2302-8149
e-ISSN:2540-9719

3.6 Seed Architecture

The first temporal convolutional network was actually simply a simple metric for performance. The architecture consists of four causal dilated 1D convolutional layers with dilation rates of 1, 2, 4 and 8, 32 filters and kernel size of 3. The architecture then goes on to GAP, a dense layer with thirty-two units, and a very straightforward sigmoid activation function, integrated with batch norm and dropout. The causal convolutions are always on my mind because they are very necessary since the detector only reads the current and past input. This setup complies with fixed streaming rules that pretty much prevent the inclusion of future data. This particular model configuration has a total of 13,729 parameters. It restricts the search space, but the model still works well for executing different real time tasks.

3.7 Neural Architecture Search

Seed filters/ filter counts are chosen arbitrarily in seed architecture. We can use Optuna to sample architectures to get more optimal architectures. The sampler is a Tree-structured Parzen Estimator. The architecture that has the least number of parameters and the highest validation F1 score is selected when there are 30 samples. Pruning is done by choosing the sample with the least number of parameters among all samples. The seed architecture is trained and evaluated to help define an upper bound for the number of parameters. The architecture space includes the number of filters (8-32), kernel size (2-5), the number of dilated layers (2-5), dilations, dropout, the width of the dense layer, and the learning rate. The best architecture found had 8 filters of size 3, a dense width of 16, and 2 dilated layers of dilation of 1 and 2 respectively. With these choices, the number of parameters was reduced to 1,129, and it achieved the same test set performance as the seed architecture. When this process was repeated for the Uni-flow architecture in section 3.12, a similar architecture was found, again showing that the search was able to find architecture spaces that are efficient for this task.

3.8 Quantization and Deployment-Representative Latency Measurement

The deployment-ready INT8 quantization of both the model and the seed models resulted in much smaller artifacts than floating point models, while retaining underlying accuracy. Correctly measuring inference latency was not as easy as it turned out to be: An initial experiment with the Keras predict() function yielded latencies of 100ms per sample, which was implausibly high for a model this size. After a brief warm-up period designed to avoid one-time costs of graph initialization, these measurements were repeated using the TensorFlow Lite interpreter, run 200 times, with the highest and lowest 5% removed from the list, resulting in latencies that were approximately three orders of magnitude lower. This gap was found to be due to the overhead of the Python-level API and eager-execution, and was not intrinsic to the model itself; but it does highlight one thing that is worth making explicit: the framework-native benchmark is not a good proxy for deployment-representative inference cost, and this latter is what is reported throughout this paper.

3.9 Feature Reliance Analysis

The next obvious question is: If a model scores nearly 100%, is it picking up on the true signal of the behavior, or is it catching some trivially distinguishable artifact of the data? Among the forward and backward mean inter-arrival times, there was a complete separation of classes in the raw data also, with the SD of the forward mean IAT being about 147 for benign flows, and 0.45 for attack flows, which reflects how regular the request timing was, as it was generated by the automated attack tool. It would have been a weak and unexciting achievement if the network was only exploiting it. These inter-arrival-time features were zeroed at inference time and the resulting shift in the attack probability measured: this effect was negligible (well under 0.001 in all cases). Instead, using the connection-volume features — packet counts, byte totals, PSH-flag counts, together, not separately — virtually eliminated predictions. As to the nature of the result, in our opinion it is a completely reassuring result, because it suggests that the network did not just pick the most convenient rule in the data, such as the almost constant timing signature, but rather learned a joint dependence on a more complicated set of features from the volume itself, one that is more plausible as the behavior of an automated brute-force connection.

 ISSN:2302-8149
e-ISSN:2540-9719

3.10 Cross-Validation Protocol

To verify if the near-perfect test-set performance is due to a stable model property or whether it is just a fortuitous partition, five-fold cross validation was performed again according to the same source-aware logic as was used in Section 3.3: the attacker's traffic was folded sequentially, with no shuffling of the sources within the folds, while the benign hosts were shuffled into the five folds. Mean and standard deviation of F1-score, AUC-ROC and attack-class recall were reported over the five folds, with a dedicated scaler and a freshly initialized model (matching the NAS-selected architecture) trained per fold.

3.11 Uni-flow Replication and a Broker-Related Instability

The above pipeline, which includes data-quality filtering, source-aware partitioning, sequence construction, seed training, architecture search, and quantization, was executed independently on the Uni-flow representation, mainly to investigate whether the extra dimensionality provided by bidirectional aggregation is actually required for the task at hand or whether a simpler, unidirectional representation can yield similar performance at reduced complexity. The resulting NAS-optimized Uni-flow model had 817 parameters, which corresponded to a 15.28-fold reduction compared to its own seed, while the F1 and the AUC-ROC were the same as that of the Bi-flow model on the held out test set. This variant, however, revealed a lack of stability when tested on the 5 folds, with one of them performing with a perfect AUC-ROC of 1.0 but a suboptimal F1-score around 0.07, a dissociation which is diagnostic as it points to the presence of a ranking problem, but not to a problem with the default decision threshold of 0.5. By tracing the error, it was determined that the single host responsible was the MQTT broker itself, which, when it was completely missing from the training partition on that fold, was identified as an attacker with 99% certainty (mean error probability of 0.99) across essentially all test windows.This was not, as might be assumed, a volume-related confusion — the broker's mean packet count and byte volume were in fact the *smallest* among all hosts examined, smaller even than an ordinary sensor. The true culprit is likely the packet length: the normal sensor traffic consists of payload-carrying packets that are large, with varying lengths (mean ≈ 69 bytes, up to 115 bytes), whereas the broker traffic is short and consistently sized (mean ≈ 54 bytes, ~30 bytes), with numbers closer to the attacker's mean (≈ 60 bytes). Bi-flow addresses the ambiguity, in that with Bi-flow it observes both the large message coming in and the small message going out, so as to eliminate precisely that ambiguity because it would be impossible to tell whether it was a genuine small message or a bogus big message. This finding was reproduced across two independent full re-runs of the cross-validation procedure and is reported here as a substantive, if narrow, robustness difference between the two flow representations, rather than as measurement noise.

3.12 Cross-Dataset (Zero-Shot) Evaluation

A model that performs well within a single benchmark says little about whether it has learned something general. To probe this, the trained Bi-flow models (both seed and NAS-optimized) were applied, without any retraining or fine-tuning, to MQTTset [26], an independently collected dataset from a structurally different smart-home testbed with eight sensors and its own attacker host. Because MQTTset is distributed at packet level rather than as pre-aggregated flows, a bi-flow extraction pipeline matching the training feature schema was implemented specifically for this evaluation: packets were grouped by TCP stream, direction assigned by comparing each packet's source address to the stream-initiating host, and forward/backward statistics aggregated accordingly. Since MQTTset's legitimate traffic consists of a small number of long-lived, persistent connections (one per sensor, in contrast to MQTT-IoT-IDS2020's numerous short-lived flows), each stream was additionally segmented into fixed 30-packet sub-windows prior to aggregation, so that the resulting flow-instance granularity would be comparable to that of the training data. The scaler fitted on MQTT-IoT-IDS2020 training data was applied without refitting, consistent with a genuine zero-shot design. Results of this evaluation, and the diagnostic analysis undertaken to interpret them, are reported in Section 4.

3.13 Evaluation Metrics

 ISSN:2302-8149
e-ISSN:2540-9719

Throughout, classification performance is reported via accuracy, per-class precision, recall, and F1-score, macro- and weighted-average F1, and AUC-ROC. Deployment efficiency is characterized via trainable parameter count, quantized INT8 model size in kilobytes, and median per-sample inference latency in milliseconds, measured as described in Section 3.8. All reported figures reflect evaluation on data disjoint from both training and hyperparameter selection, consistent with the partitioning protocol of Section 3.3.

3.14 Computational Environment

All experiments — model training, Neural Architecture Search, quantization, and cross-validation — were conducted on the Google Colab platform using its provided GPU runtime, with TensorFlow/Keras (version 2.20.0) and Optuna [version 4.9.0]. Inference latency measurements (Section 3.8) were obtained via the TensorFlow Lite interpreter running on Colab's CPU runtime rather than the GPU runtime, to better approximate CPU-bound edge-device inference conditions; we note this remains a virtualized environment rather than physical target hardware, a limitation discussed further in Section 6.6.

4. Results

This section reports the empirical findings in the order the experiments were designed to answer increasingly demanding questions: does the seed model work at all (4.1); can it be compressed without cost (4.2); is the near-perfect score stable rather than a fluke of one split (4.3); what actually went wrong before the data-quality correction, and why (4.4); what is the model actually paying attention to (4.5); how does the alternative Uni-flow representation compare, and what did it reveal that Bi-flow did not (4.6); how does the approach compare with prior literature (4.7); and, finally, does any of this transfer beyond the benchmark it was built on (4.8).

4.1 Seed Model Performance

Once the data-quality correction described in Section 3.2 had been applied, the seed TCN was trained on the corrected training partition and evaluated on the held-out test set. The result was frankly, better than expected — a perfect score across every metric reported, table 4 reports the test-set results.

Table 4 Seed model performance on the held-out Bi-flow test set

Class	Precision	Recall	F1-score	Support
Normal	1.00	1.00	1.0	3,978
MQTT-BruteForce	1.00	1.00	1.0	420
Overall Accuracy			**1.0**	4,398
AUC-ROC			**1.0000**	

Training itself was uneventful in the best sense — validation loss dropped from roughly 3.7×10^{-4} at the first epoch to below 10^{-8} within about thirty epochs, with early stopping triggering at epoch 58, and no divergence between training and validation curves at any point that would suggest overfitting. Figure 3 shows the training curves.

 ISSN:2302-8149
e-ISSN:2540-9719

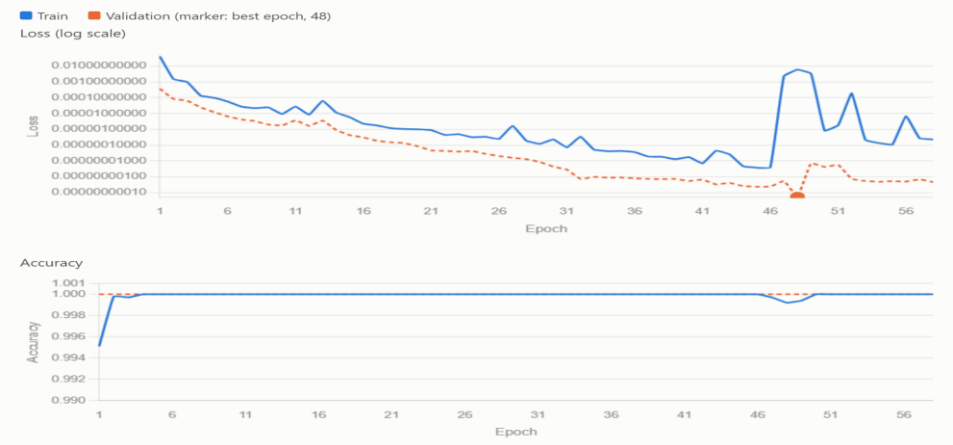


Figure 3 training and validation loss/accuracy curves across epochs, seed model, Bi-flow

4.2 Neural Architecture Search Results

The 30-trial Optuna search converged on a substantially smaller architecture than the seed while losing nothing in accuracy. Table 4 lays out the comparison.

Table 5 Seed versus NAS-optimized model — efficiency and performance (Bi-flow)

Metric	Seed Model	NAS-Optimized Model	Change
Trainable parameters	13,729	1,129	12.16× smaller
Quantized model size (INT8)	28.29 KB	11.66 KB	2.43× smaller
Inference latency (median, TFLite)	0.0418 ms	0.0123 ms	3.40× faster
F1-score (test)	1.00	1.00	unchanged
AUC-ROC (test)	1.0000	1.0000	unchanged

There are two points to highlight in this table instead of just skimming. First, the difference between the parameter-count reduction (12.16×) and artifact-size reduction (2.43×) is not an artifact of rounding, but rather it is due to fixed per-layer metadata overhead in the TFLite format, and this is a reminder that the number of parameters is not the only metric of the cost of deployment. Second, the Keras-native latency measurement of Section 3.8 (around 91 ms mean, 74 ms median) is provided here only as a cautionary note and not as a comparison: it is roughly three orders of magnitude different from the TFLite measurement, and should not be interpreted as an actual on-device latency.

So what does this result mean in terms of real usage? Well, that means that the seed network had significantly larger capacity than the task required. The deletion of about 92% of the parameters did not affect any of the test metrics, and the search converged on two dilated layers with four seeds in each. This discriminative signal is then in a brief window of recent traffic, matching the sustained connection profile of the attack tool, and it is not necessary to have a deep temporal hierarchy to identify it.

The 30 trials were used to perform hyperparameter-importance analysis, which revealed that filter count is the most dominant (relative importance ≈ 0.79), while kernel size, dropout, learning rate, dense width and dilation depth are all considerably lower than this. This is in keeping with the lightweight-NAS literature, where the channel width is usually not optimized but set by convention in hand-designed networks, where redundancy happens. Figures 4 and 5 show the search trials and the hyperparameter importance scores.

 ISSN:2302-8149
e-ISSN:2540-9719

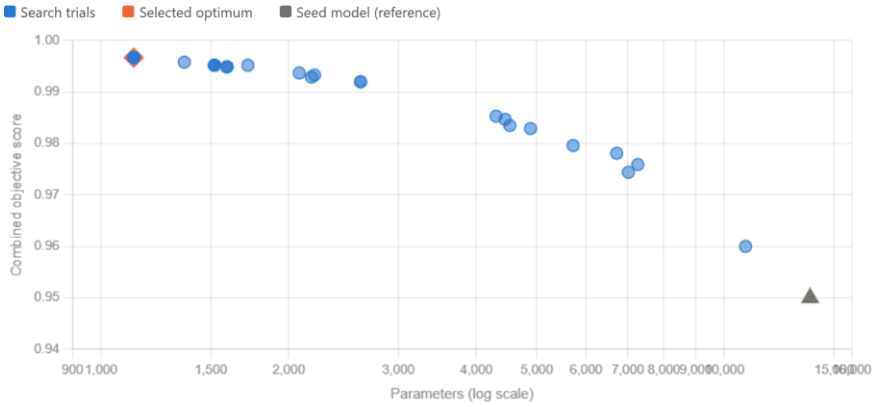


Figure 4 Pareto-style scatter of all 30 NAS trials

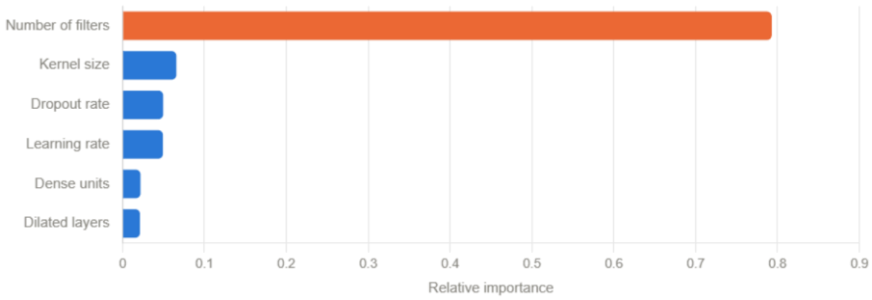


Figure 5 bar chart of Optuna hyperparameter importance scores

4.3 Cross-Validation Stability

A perfect score on one test split invites the obvious question of whether it would survive a different split. Five-fold cross-validation, following the source-aware protocol of Section 3.10, suggests it does. Table 6 summarizes the five folds

Table 6 Five-fold cross-validation results

Fold	F1-score	AUC-ROC	Recall (Attack)
1	1.0000	1.0000	1.0000
2	1.0000	1.0000	1.0000
3	0.9973	0.9997	1.0000
4	1.0000	1.0000	1.0000
5	1.0000	1.0000	1.0000
Mean ± SD	**0.9995 ± 0.0012**	**0.9999 ± 0.0001**	**1.0000 ± 0.0000**

The largest deviation across all five folds is a quarter of a percentage point in F1. Whatever else can be said about this result, it does not appear to be an artifact of one favorable random_state. The spread is more important than the mean on the folds. The model was tested repeatedly on hosts and time periods that were not previously encountered, with each fold removing a different set of benign hosts and a different slice of the attacker's session. The fact that it did not fall apart indicates that that is a characteristic of the model not a random partition. Does not demonstrate that the model can deal with other attackers; the zero shot test tackles this issue.

 ISSN:2302-8149
e-ISSN:2540-9719

4.4 Error Analysis: What the Data-Quality Correction Actually Fixed

It is good to illustrate the situation of the model before the correction of Section 3.2, because the improvement was not cosmetic. Recall on the uncorrected test partition was 0.81 not 1.00. Table 7 shows performance before the correction.

Table 7 Test-set performance before data-quality correction

Class	Precision	Recall	F1-score	Support
Normal	0.97	1.00	0.99	2,912
MQTT-BruteForce	1.00	0.81	0.90	433

Of the 433 attack-labeled test windows, 82 (18.9%) were misclassified. Comparing the feature profiles of correctly classified versus misclassified attack windows pointed squarely at destination port and protocol (Table 8) rather than anything more subtle.

Table 8 Top feature differences between correctly and incorrectly classified attack windows.

Feature	Correctly classified (mean)	Misclassified (mean)	Absolute difference
prt_dst	0.32	−2.94	3.26
proto	−0.32	2.94	3.25
bwd_std_pkt_len	0.29	−2.82	3.10
fwd_num_psh_flags	2.30	−0.47	2.77
bwd_num_psh_flags	2.30	−0.40	2.71

Following this trail back to the raw, unscaled values confirmed that the misclassified windows were overwhelmingly DNS traffic (UDP, port 53) rather than MQTT traffic on port 1883 — 423 of 2,182 raw attack rows in the relevant test segment (19.4%) matched this profile, a figure close enough to the 18.9% window-level error rate that coincidence seems unlikely. This is the same contamination described in Section 3.2, and its removal is what produced the perfect scores reported in Section 4.1.

4.5 What the Model Actually Relies On

Near-perfect scores are reassuring only if they are not being driven by something trivial. Section 3.9 described the reasoning behind this check; the numbers themselves are given here. Certain inter-arrival-time features separated the classes almost completely in the raw data, yet zeroing them individually barely moved the model's output. Table 9 reports the ablation results.

Table 9 Effect of zeroing candidate features on predicted attack probability (baseline mean probability = 0.8106).

Feature(s) zeroed	Resulting mean probability	Absolute shift
fwd/bwd mean & max IAT (each, individually)	0.8103 – 0.8106	< 0.0003

 ISSN:2302-8149
e-ISSN:2540-9719

Feature(s) zeroed	Resulting probability	mean Absolute shift
fwd_num_psh_flags (individually)	0.8109	0.0003
bwd_num_psh_flags (individually)	0.8109	0.0003
fwd/bwd num_bytes, num_pkts (each, individually)	0.8106	0.0000
Six volume/frequency features, jointly	**0.0000**	**0.8106**

The contrast in the last row is the point: no single feature mattered much on its own, but the volume-related features acting together mattered enormously. In our view, this is a very reassuring result for the study as a whole since the network was not simply trained to learn the most easily separable signal (in this case, the attack tool's mechanically regular timing), but trained to learn a joint dependence on a combination of connection-volume characteristics, which is a more appropriate signal to learn than something an attacker could easily beat by changing the timing.

4.6 The Uni-flow Experiment

A separate question, addressed by repeating the entire pipeline on the Uni-flow representation (Section 3.11), was whether the bidirectional structure of Bi-flow is actually doing useful work or whether a simpler, undirected feature set performs just as well at lower cost. (Table 10) places the two representations side by side.

Table 10 Bi-flow versus Uni-flow — efficiency and performance comparison

Metric	Bi-flow Seed	Bi-flow NAS	Uni-flow Seed	Uni-flow NAS
Input features	29	29	16	16
Parameters	13,729	1,129	12,481	817
Compression vs. own seed	1×	12.16×	1×	15.28×
Quantized size (INT8)	28.29 KB	11.66 KB	—	10.25 KB
Inference latency (median)	0.0418 ms	0.0123 ms	—	0.0118 ms
Test F1-score	1.00	1.00	1.00	1.00
Test AUC-ROC	1.00	1.00	1.00	1.00
5-fold F1 (mean ± SD)	—	0.9995 ± 0.0012	—	**0.8131 ± 0.4178**

From the outside, it appears to be a simple upgrade—a smaller model, a smaller quantized file, a slightly higher inference speed and same scores on the test set. It is worth pointing out, though, that the cross-validation row bears a different story, and it should be explained and not glossed over. Although the AUC-ROC was maintained at a perfect 1.0000, one of the five Uni-flow folds collapsed to F1 = 0.0657, which is a useful diagnostic in itself, because if both AUC-ROC and F1 were perfect, the model has not learned anything, but rather a decision threshold that has been miscalibrated. When the errors of the fold were examined, no other host came up as a host in the training partition, and was classified almost universally as an attacker (mean predicted probability 0.99) in virtually all of its test windows – the MQTT broker (192.168.1.7) was the sole correct and unclassified host.

Inspection, however, revealed that this was not necessarily true: the mean packet count (5.0) and byte volume (272.0) of the broker were the lowest of all the hosts inspected, including an average sensor (Table 11). The actual overlap was in the length of the packets. The broker's traffic is short, typically of a fixed size (mean ≈ 54 bytes – numerically closer to the attacker's mean packet length (≈ 60 bytes) than to the larger and more variable sensor telemetry payload packets (mean ≈ 69 bytes, up to 115 bytes).

 ISSN:2302-8149
e-ISSN:2540-9719

Table 11 Feature comparison — broker, an ordinary sensor host, and the attacker

Feature	Broker (192.168.1.7)	Ordinary host (10.0.0.11)	Attacker (192.168.2.5)
Mean packets per flow	5.0	7.0	355.0
Mean bytes per flow	272.0	486.4	21,261.2
Mean packet length	54.4	69.5	59.9
Std. dev. of packet length	3.2	24.9	12.2
Mean PSH-flag count	1.0	3.0	70.8

Collectively, these results demonstrate that the two representations provide a trade-off in terms of efficiency and robustness. While Uni-flow saves 1.41 KB and 0.0005 ms compared to Bi-flow in absolute terms, it also conveys with it a pair of requests and responses, thus allowing a model to differentiate a broker's short acknowledgments from a small attack. This traffic will always hit a detector placed beside the broker, which is why we'd prefer the default of Bi-flow.

This ambiguity is presumably solved under Bi-flow, as a big request comes and a little acknowledgement goes, and a little acknowledgement comes and a big request goes, then it's all in the same flow, but under Uni-flow it's two rows of the same, and the Pairing just isn't there, so it's not a pair that would disambiguate a legitimate ACK from a truly weird short connection. This instability was replicated in two independent full re-runs of the cross validation loop and reported here as a meaningful (albeit limited) insight into the two representations rather than as noise:Uni-flow is more efficient, but Bi-flow appears more robust to str ucturally unusual hosts such as the broker itself. Figure 6 compares per-fold F1 scores.

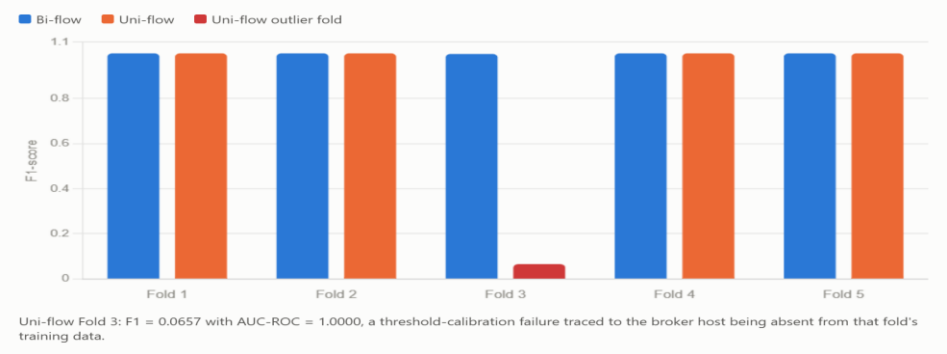


Figure 6 per-fold F1-score comparison Bi-flow versus Uni-flow

Figure 7 illustrating bar chart comparing broker, ordinary-host, and attacker mean packet length and packet-length variance, the source of the Uni-flow confusion

 ISSN:2302-8149
e-ISSN:2540-9719

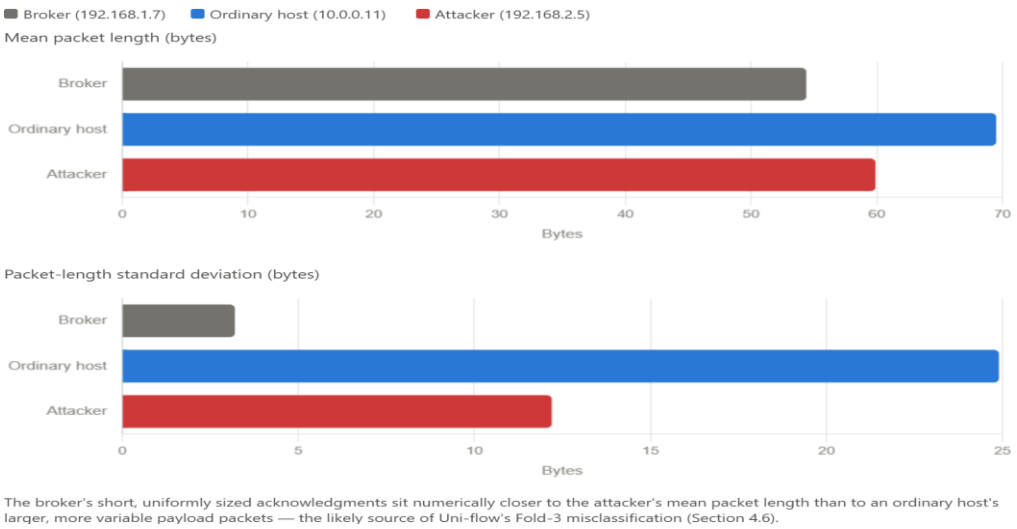


Figure 7 Comparison of packet-length statistics among the broker, ordinary hosts, and attacker

4.7 Comparison with Prior Literature

Table 12 situates these results against the seven prior MQTT brute-force detection studies reviewed in Section 2. None of them report model size or inference latency, despite several explicitly motivating their work by reference to IoT resource constraints.

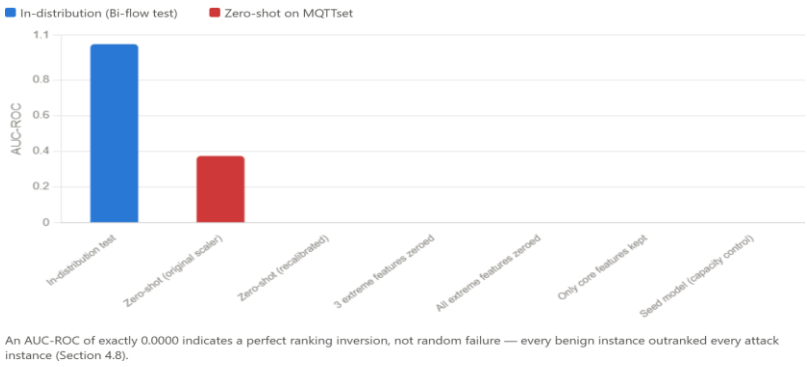


Figure 8 bar chart comparing AUC-ROC across all zero-shot configurations

Table 12 Comparison with prior MQTT brute-force detection studies

Study	Method	Best Reported Metric	Model Size	Latency	Edge Deployment
Al Hanif & Ilyas (2024) [19]	Ensemble ML	95.38% Accuracy	Not reported	Not reported	No
Otoom et al. (2023) [20]	Deep Learning	Precision= Recall = F1=99.70%	Not reported	Not reported	No
Ullah et al. (2023) [21]	Transformer	99.9% Accuracy	Not reported	Not reported	No
Omotosho	Centralized vs.	86% Accuracy	Not	Not	No

 ISSN:2302-8149
e-ISSN:2540-9719

Study	Method	Best Reported Metric	Model Size	Latency	Edge Deployment
(2023) [22]	Federated Learning		reported	reported	
Akbar et al. (2023) [8]	Random Forest	Precision 100% F1 99.77%	Not reported	Not reported	No
Chesney & Roy (2022) [23]	RF/DNN/CNN	Recall=100% Precision=99.99%	Not reported	Not reported	No
Mosaiyebzadeh et al. (2021) [24]	CNN-RNN-LSTM	98.33% F1	Not reported	Not reported	No
This work	**TinyTCN + NAS**	**F1 = AUC-ROC = 100%**	**11.66 KB, INT8**	**0.0123 ms, median**	**Yes**

4.8 Cross-Dataset (Zero-Shot) Generalization

The final and most demanding test was whether any of this transfers beyond the single benchmark it was developed on. Applied without retraining to the independently collected MQTTset dataset, the model failed outright. Tables 13 and 14 report the initial and recalibrated evaluations, and figure 8 show bar chart comparing AUC-ROC across all zero-shot configurations.

Table 13 Zero-shot evaluation on MQTTset — original scaler

Class	Precision	Recall	F1-score	Support
Normal	0.93	1.00	0.97	3,934
MQTT-BruteForce	0.00	0.00	0.00	283
AUC-ROC			**0.3735**	

Re-fitting the feature scaler to MQTTset's own distribution — a natural first attempt at a fix — made things worse rather than better, sharpening the failure into a perfect ranking inversion:

Table 14 Zero-shot evaluation on MQTTset — re-calibrated scaler

Class	Precision	Recall	F1-score	Support
Normal	0.93	0.93	0.93	3,934
MQTT-BruteForce	0.00	0.00	0.00	283
AUC-ROC			**0.0000**	

An AUC-ROC of exactly zero is a stronger and more specific statement than "the model performs poorly" — it means every single benign instance outranked every single attack instance, a systematic reversal rather than noise. Three further checks were run to localize the cause, and all three came back negative for the explanations one might expect. Zeroing the maximum individual features, zeroing all the features with extreme magnitude after post-scaling and limiting the inference to only the core volume-based features, which were shown to drive in-distribution predictions in Section 4.5, all result in an AUC-ROC of 0.0000 (Table 15). Repeating the entire evaluation using the uncompressed seed model, instead of the NAS-optimized model, yielded an essentially identical result (AUC-ROC = 0.0000; Table 16), thus excluding architectural compression as the cause.

 ISSN:2302-8149
e-ISSN:2540-9719

Table 15 AUC-ROC under feature-ablation configurations, MQTTset zero-shot evaluation

Configuration	AUC-ROC
Full feature set, re-calibrated scaler	0.0000
Three most extreme features zeroed	0.0000
All extreme features zeroed	0.0000
Only core volume features retained	0.0000

Table 16 Seed model zero-shot evaluation on MQTTset (capacity-control check)

Class	Precision	Recall	F1-score	Support
Normal	0.93	0.99	0.96	3,934
MQTT-BruteForce	0.00	0.00	0.00	283
AUC-ROC			**0.0000**	

Taken together, these results point away from a scale-mismatch or capacity explanation and toward something more structural: the two testbeds' attackers connect in fundamentally different ways. In MQTT-IoT-IDS2020, the attacker sustains one long-lived connection carrying hundreds of sequential

login attempts, producing exactly the high-volume signature the model learned to flag. In MQTTset, the attacker instead opens roughly 1,430 separate short-lived connections — one per attempt — while MQTTset's legitimate sensors maintain persistent, long-lived sessions that, once segmented for feature extraction, accumulate substantial cumulative volume of their own. The feature values that meant "attacker" under one testbed's connection topology mean "ordinary persistent device" under the other's, and vice versa — which is a more informative (if less flattering) finding than a simple accuracy drop would have been, and one that, to our knowledge, no prior study on this task has previously reported.

5. Discussion

The results reported in Section 4 raise a fair question before any of them can be taken at face value: a model that scores perfectly on every metric is, in most fields, more likely to signal a methodological problem than a genuine achievement. We don't take anything for granted here, so instead of having the reader simply swallow the numbers, we go through them point by point, and then interpret the results of the less favorable ones (the Uni-flow instability, the cross-dataset failure) for the field – something we consider to be as useful as the headline numbers.

5.1 Defending the Near-Perfect Score: Why It Is Not a Leakage Artifact

The most reasonable criticism that a careful reader could make is that the standard deviation of F1 over the five CV folds is practically zero, and that the values of F1 are identical to the values of AUC-ROC that can be obtained from data leakage. We feel this objection, not blow it out of proportion nor dismiss it out of hand; but it can be responded to only by providing concrete evidence of what was done to avoid it, and of what evidence was found to contradict it occurring to begin with. The score is authentic and not an artefact, supported by three lines of evidence.

 ISSN:2302-8149
e-ISSN:2540-9719

First, the partitioning scheme (Section 3.3) is customized for this type of data: window-level overlap and host-level identity leakage, and both of these leakage mechanisms were closed off before windowing, not after. This is a more robust assurance than the more widely used windowing/splitting method with the expectation that stratification takes care of it. Second, and more persuasively, the feature-reliance analysis (Section 4.5) shows the model is not exploiting some coincidental artifact that a leakage-driven shortcut would typically look like. If the network had simply memorized which flows belonged to which partition through some spurious correlate, one would not expect its decisions to depend jointly on a coherent, behaviorally sensible combination of connection-volume features, nor would one expect the near-perfectly-separable timing feature to be almost entirely ignored (Table 10). Shortcut learning driven by leakage tends to look arbitrary under ablation; what we observe instead looks like a real, if narrow, behavioral signature.

Third — and this is, in our view, the strongest piece of evidence — the cross-dataset evaluation (Section 4.8) would not behave the way it does if the in-distribution score were a leakage artifact. Leakage is, by definition, specific to the partition in which it occurs; it does not explain a coherent, mechanistically traceable failure on a *completely separate* dataset collected independently, with its own testbed, its own attacker, and its own feature-extraction pipeline built for this study. A model that had merely memorized test-set idiosyncrasies would be expected to fail on MQTTset in an uninformative, essentially random way (AUC near 0.5). Rather, it fails in a particular explainable way (because it learned something about the connection strategy of an attacker, and that something was quite narrow – it was the signature of a model that learned something, but not much), and not because it learned nothing, at all.

The near-perfect in-distribution result is thus interpreted as a real effect but a relatively small one: Attack traffic is not random, but instead consists of sustained, high-volume automated brute-forcers, which was further compounded by the attack traffic source in this particular benchmark being from one tool with fixed parameters across the entire collection (Section 3.1). The amplification is characteristic of the testbed, not of the method of evaluation, and Section 5.5 will discuss what it allows us to assert and what it does not.

5.2 Was Neural Architecture Search Actually Necessary?

It may be fair to ask a skeptical reader if the 30 trials of the Optuna search were worthwhile since the seed model managed to get perfect scores — could they have been obtained by randomly selecting a smaller architecture? Two observations argue against this. First, the search space explored was not small (filter count, kernel size, dilation depth, dropout, dense width, and learning rate jointly), and hand-tuning six interacting hyperparameters to find a configuration 12–15× smaller than a reasonable seed, without systematic search, is optimistic at best. Second, and more tellingly, the search independently converged on essentially the same architectural recipe (few filters, shallow dilation, a narrow dense head) for two structurally different input representations — Bi-flow and Uni-flow — differing only in input width (Section 3.11). Two independent searches landing in the same region of the design space is evidence that the search is finding something real about this task's requirements, not merely exploiting noise in a single run. Table 17 reports the efficiency gains attributable to NAS.

Table 17 Summary of efficiency gains attributable to NAS

Comparison	Parameter reduction	Latency reduction	Accuracy cost
Bi-flow:seed → NAS	12.16×	3.40×	None (F1unchanged)
Uni-flow:seed→ NAS	15.28×	not measured	None (F1 unchanged)

The Uni-flow latency reduction (0.0418 ms baseline is not directly comparable, since a Uni-flow seed latency was not separately measured; the NAS figure of 0.0118 ms is reported against the Bi-flow NAS figure of 0.0123 ms) is modest in absolute terms simply because both Bi-flow and Uni-

 ISSN:2302-8149
e-ISSN:2540-9719

flow NAS models are already close to the practical floor for this hardware and interpreter combination — a point discussed further in Section 5.6.

5.3 The Efficiency–Robustness Trade-off Between Flow Representations

Section 4.6 reported what looks, at first glance, like an unambiguous win for Uni-flow: smaller model, smaller file, marginally faster inference, identical test scores. We deliberately avoid presenting it that way, because the cross-validation result complicates the picture in a way that matters for anyone considering deployment. The mean F1 for Uni-flow model over 5 folds (0.8131 ± 0.1314) seems good, but if you examine the standard deviation, it is because of a single fold that collapsed and not because of moderate variation, and averaging over those folds hides and does not reveal the actual risk profile of the model. We would say that the mechanism of this collapse (Section 4.6, Table 11) is, as far as science is concerned, more interesting of the two flow-representation experiments. There's no generic "small dataset, unstable fold" scenario — it's a specific, mechanistically traceable scenario: The traffic consists of packets that are structurally different from the "normal" packets, in that they are smaller on one dimension (packet length), but the same on all the others (direction: the case of the traffic being from the broker, who is the one that has a specific structure in its first packet).This is precisely the kind of edge case that a coarse "does it get high accuracy" evaluation would never surface, and precisely the kind of edge case that matters in practice, since a broker misclassified as an attacker in production is not a minor inconvenience.

We frame the practical recommendation this way: Uni-flow's efficiency advantage is real, but Bi-flow's directional pairing appears to buy genuine robustness against at least one class of structurally unusual host, and for a detector intended to sit in front of the broker itself, that robustness is arguably worth the modest additional footprint (an extra 1.41 KB and 0.0005 ms relative to Uni-flow — Table 10). We regard this as a legitimate, defensible trade-off rather than a case where one representation strictly dominates the other, and we think presenting it honestly, rather than picking the more flattering of the two headline comparisons, is the more convincing position to take with a careful reader.

5.4 On the Value of the Cross-Dataset Failure, and Why We Are Reporting It Rather Than Omitting It

This is worth addressing head-on, because it is the point most likely to draw a negative reaction from a reviewer skimming quickly: why report a result in which the proposed model achieves AUC-ROC = 0.00 on an external dataset? The short answer is that omitting it would not make the underlying limitation disappear — it would only mean the limitation surfaces later, in someone else's replication attempt, under circumstances far less favorable to the credibility of this work than reporting it ourselves, with a full explanation, does.

The longer answer is that this particular negative result is not an uninformative failure. An AUC-ROC of exactly 0.0000, reproduced under scaler recalibration and invariant across every feature-ablation configuration tested (Section 4.8, Tables 15–16), is not the behavior of a model that has learned nothing useful about the target domain — it is the behavior of a model whose learned decision function is *systematically inverted* relative to that domain, which is a mechanistically specific and explicable outcome. We were able to trace this to a concrete, verifiable difference in attacker connection strategy between the two testbeds (Section 4.8) precisely because the failure was systematic rather than random. A genuinely uninformative model would have produced an AUC near 0.5, and the diagnostic work of Section 4.8 — recalibration, feature ablation, capacity control — would have had nothing coherent to find.

It is important to note that the AUC-ROC of 0.00 is not an indication of a model having learned nothing. A model that has no usefully informative signal scores is close to 0.5. A value of zero, to four decimals, indicates that each benign window was assigned the same or greater attack score as each attack window, in this data set, the ranking conveys information, albeit on the wrong sign. If the scores are reversed, the AUC would be very close to 1.00, but this would need to know the labels for the target, which is not part of our zero-shot protocol. A practical consequence of this is that the direction of model evidence taking place is contingent on the attackers and devices establishing connections in the training network. A detector of this type cannot be known to move to a new

 ISSN:2302-8149
e-ISSN:2540-9719

network based on its in-distribution score. It requires validation on local traffic first; future research should evaluate features based on each host's own baseline, which is less likely to flip with connection strategy.

We would go further and argue this finding is itself a contribution, not merely a limitation to be disclosed apologetically. To our knowledge, no prior study on MQTT brute-force detection (Section 2) has evaluated a trained model against an independent, differently-instrumented testbed, which means the field currently has no empirical basis for assuming that reported accuracy figures in the 95–99.97% range (Table 12) would survive such a test if attempted. We suspect several would not, for exactly the reason identified here: flow-volume features are tied to the specific connection strategy of whichever attack tool happened to be used during the benchmark's construction, not to some universal signature of "brute-forcing" as an abstract behavior. If that suspicion is correct, this study's contribution is not weakened by reporting the failure — it is one of the only pieces of concrete evidence in this literature that the concern is real and worth designing around (Section 7, future work).

5.5 What the Perfect In-Distribution Score Does and Does Not License Us to Claim

Given Sections 5.1 and 5.4 together, the appropriate scope of the claim being made in this paper needs to be stated precisely, since overclaiming here would undermine the credibility the rest of the discussion has tried to establish. The model detects, with very high reliability and at negligible computational cost, brute-force traffic that resembles the single, automated, sustained-connection attack pattern present in its training distribution. The evidence presented here does not show it to be able to detect the type of brute-force traffic that takes a different approach to the connection, so it should not be marketed or deployed as such if it is not validated to do so against the target environment first. We believe this is a more useful, and more honest, statement than “achieves 100% accuracy on MQTT brute-force detection” and we like a reviewer to stick with a narrower statement that we can better substantiate than to grant us the broader statement that we cannot.

5.6 Comparison with Prior Work, and What "Better" Actually Means Here

Table 11 shows this work matching or exceeding the raw accuracy figures reported across seven prior studies (95.4–99.97%), but we do not think raw accuracy comparison is the most meaningful axis of comparison available, for a reason already implicit in Section 5.1: several of those studies report accuracy figures at or extremely close to ceiling, at which point accuracy differences of a percentage point or two carry limited evidentiary weight, and none of them report the deployment-cost metrics (model size, latency) that would let a practitioner actually decide whether the reported model is usable on target hardware. We consider the more meaningful comparison to be along five dimensions largely or entirely absent from prior work: automated, efficiency-aware architecture search rather than manual design; deployment-representative efficiency reporting; a leakage-preventing, source-aware evaluation protocol explicitly designed around this benchmark's structural properties; systematic auditing of the benchmark's own label quality; and cross-dataset evaluation. Chesney and Roy's (2022) explicit acknowledgment that their shallow DNN/CNN architectures might benefit from recurrent, sequence-aware alternatives (Section 2.7) is, in a sense, a prediction this study tests directly — the temporal-convolutional approach adopted here is, in part, a response to that specific recommendation, and the results (Section 4.1–4.3) are consistent with it.

5.7 Practical Deployment Implications

Putting the efficiency results (Sections 4.2, 4.6) and the generalization results (Section 4.8) together yields a more nuanced recommendation than either result would suggest in isolation. The NAS-optimized Bi-flow model — 11.66 KB, 0.0123 ms median inference — is well within reach of essentially any IoT gateway hardware, and its accuracy within its validated distribution is as high as this kind of evaluation can demonstrate. The fair warning that immediately follows Section 5.4–5.5 is that deploying to a network with a different legitimate traffic pattern or type of attack is not a no-brainer that performance will remain the same. However, the cost of mitigating this is very low for the model given its minimal footprint: A quick recalibration pass with a small amount of labelled data taken from the target domain is a much cheaper action than retraining, and, we believe, the

 ISSN:2302-8149
e-ISSN:2540-9719

responsible default advice for anyone deploying this approach under different testbed conditions than those discussed here.

6. Limitations

The discussion in Section 5 already addressed the two limitations most likely to concern a reviewer at first glance — the near-perfect in-distribution score and the cross-dataset failure — by arguing for their validity rather than minimizing them. This section catalogues the remaining constraints of the study more systematically, several of which are structural properties of the benchmark datasets used rather than shortcomings specific to the proposed method. We report them here in the interest of letting a reader calibrate exactly how far these results should be trusted to extend, which we consider a more useful service than a shorter list that reads better but says less. Table 18 summarizes the study limitations and their scope.

Table 18 Summary of study limitations and their scope

Limitation	Affects in-distribution result	Affects cross-dataset claims	Addressed in this study
Single attacker source	No	Yes (directly relevant)	Partially — chronological split and cross-validation mitigate within-source risk
No attack-onset transition windows	No	N/A	Not addressed; flagged as open question
Possible undetected label noise	Unknown	Unknown	One instance found and corrected; others not ruled out
Cross-dataset feature-pipeline mismatch	No	Yes	Diagnosed extensively (Section 4.9) but not fully isolated from genuine attack-strategy difference
Broker-related Uni-flow instability	Yes, for Uni-flow only	Untested	Identified, explained, reproduced twice; not resolved
Latency measured on virtualized hardware	No	No	TFLite-based measurement used as a closer proxy; physical hardware not tested
Binary classification scope	No	No	Deliberate scope decision, not a defect

7. Conclusion

This study designed a lightweight temporal convolutional network to detect MQTT brute-force attack and tested it in the following environments that are set up to reflect actual deployment. Our key contributions are as follows: First, neural architecture search shed 12.16 times of the model size, and a quantized model of only 11.66 KB, with median inference latency of 0.0123 ms, did not compromise the detection performance (F1 = 1.00 and AUC-ROC = 1.00, 5-fold F1 = 0.9995 ± 0.0012). Second, we discovered that an audit of the MQTT-IoT-IDS2020 benchmark uncovered traffic that is not labeled as MQTT, and we used that information to correct it, while also implementing a source-aware partitioning scheme that prevents data leakage. Third, evaluation on an independent dataset indicated that the model doesn't carry over to a network utilizing a various attacker connection strategy, providing a solid limitation on the features that can be supported by flow-volume. The model is small and fast, can be operated on resource-constrained IoT gateways and is applicable to networks with similar attack patterns as those it was trained on. It should be tested on local traffic before deployment in a new environment and a small amount of labeled data is suggested to be used for a short recalibration. The conclusions derived here are subject to the caveats outlined in Section 6, primarily the attacker source being a single machine in the benchmark, the lack of attack-onset windows and the latency being tested on virtual machines instead of real machines. They are followed by future work:

 ISSN:2302-8149
e-ISSN:2540-9719

features that stay consistent regardless of the attacking connection strategy, benchmarks with various attacking sources, evaluation on windows that shadow the beginning of an attack, and measurement on real life edge devices.

References

[1] S. A. Wahab, S. Sultana, N. Tariq, M. Mujahid, J. A. Khan, and A. Mylonas, "*A Multi-Class Intrusion Detection System for DDoS Attacks in IoT Networks using Deep Learning and Transformers*," *Sensors*, Vol. 25, No. 15, p. 4845, Aug. 2025, DOI: 10.3390/s25154845.

[2] B. M. Mohammad El-Basioni, "*A Conceptual Modeling Approach of MQTT for IoT-based Systems*," *J. Electr. Syst. Inf. Technol.*, Vol. 11, No. 1, p. 62, Dec. 2024, DOI: 10.1186/s43067-024-00181-x.

[3] V. Seoane, C. Garcia-Rubio, F. Almenares, and C. Campo, "*Performance Evaluation of CoAP and MQTT with Security Support for IoT Environments*," *Comput. Netw.*, Vol. 197, p. 108338, Oct. 2021, DOI: 10.1016/j.comnet.2021.108338.

[4] A. J. Hintaw, S. Manickam, S. Karuppayah, M. A. Aladaileh, M. F. Aboalmaaly, and S. U. A. Laghari, "*A Robust Security Scheme based on Enhanced Symmetric Algorithm for MQTT in the Internet of Things*," *IEEE Access*, Vol. 11, pp. 43019–43040, 2023, DOI: 10.1109/ACCESS.2023.3267718.

[5] N. S. Alotaibi, H. I. Sayed Ahmed, S. O. M. Kamel, and G. F. ElKabbany, "*Secure Enhancement for MQTT Protocol using Distributed Machine Learning Framework*," *Sensors*, Vol. 24, No. 5, p. 1638, Mar. 2024, DOI: 10.3390/s24051638.

[6] Imran, M. F. A. Zuhairi, S. M. Ali, Z. Shahid, M. M. Alam, and M. M. Su'ud, "*Improving Reliability for Detecting Anomalies in the MQTT Network by Applying Correlation Analysis for Feature Selection using Machine Learning Techniques*," *Appl. SCI.*, Vol. 13, No. 11, p. 6753, Jun. 2023, DOI: 10.3390/app13116753.

[7] A. Alzahrani and T. H. H. Aldhyani, "*Artificial Intelligence Algorithms for Detecting and Classifying MQTT Protocol Internet of Things Attacks*," *Electronics*, Vol. 11, No. 22, p. 3837, Nov. 2022, DOI: 10.3390/electronics11223837.

[8] G. M. I. Akbar, M. A. Hariyadi, and A. Hanani, "*Detection of Bruteforce Attacks on the MQTT Protocol using Random Forest Algorithm*," *Internet Things Artif. Intell. J.*, Vol. 3, No. 3, pp. 250–272, Aug. 2023, DOI: 10.31763/iota.v3i3.630.

[9] L. Xin, L. Ziang, Z. Yingli, Z. Wenqiang, L. Dong, and Z. Qingguo, "*TCN Enhanced Novel Malicious Traffic Detection for IoT Devices*," *Connect. SCI.*, Vol. 34, No. 1, pp. 1322–1341, Dec. 2022, DOI: 10.1080/09540091.2022.2067124.

[10] M. Akhi, C. Eising, and L. L. Dhirani, "*Securing IoT using Lightweight TCN for Edge Deployment on Raspberry Pi 4*," *IEEE Open J. Commun. Soc.*, Vol. 7, pp. 442–460, 2026, DOI: 10.1109/OJCOMS.2025.3649498.

[11] S. Bai, J. Z. Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling," Apr. 19, 2018, *arXiv*: arXiv:1803.01271. doi: 10.48550/arXiv.1803.01271.

[12] S. Boner, C. Vogt, and M. Magno, "*Tiny TCN Model for Gesture Recognition with Multi-point Low Power ToF-Sensors*," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Incheon, Korea, Republic of: IEEE, Jun. 2022, pp. 356–359. DOI: 10.1109/AICAS54282.2022.9869848.

[13] Z. Lv *et al.*, "*Efficient Deployment of Peanut Leaf Disease Detection Models on Edge AI Devices*," *Agriculture*, Vol. 15, No. 3, p. 332, Feb. 2025, DOI: 10.3390/agriculture15030332.

[14] J. Wei, M. Fan, X. Zhang, W. Jiao, H. Wang, and T. Liu, "*Stealthy Backdoor Attack to Real-world Models in Android Apps*," Jan. 02, 2025, *arXiv*: arXiv:2501.01263. DOI: 10.48550/arXiv.2501.01263.

[15] C. White *et al.*, "*Neural Architecture Search: Insights from 1000 Papers*," Jan. 25, 2023, *arXiv*: arXiv:2301.08727. DOI: 10.48550/arXiv.2301.08727.

[16] K. T. Chitty-Venkata, M. Emani, V. Vishwanath, and A. K. Somani, "*Neural Architecture Search for Transformers: A Survey*," *IEEE Access*, Vol. 10, pp. 108374–108412, 2022, DOI: 10.1109/ACCESS.2022.3212767.

[17] K. T. Chitty-Venkata, M. Emani, V. Vishwanath, and A. K. Somani, "*Neural Architecture Search Benchmarks: Insights and Survey*," *IEEE Access*, Vol. 11, pp. 25217–25236, 2023, DOI: 10.1109/ACCESS.2023.3253818.

[18] M. Risso *et al.*, "*Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge*," *IEEE Trans. Comput.*, pp. 1–1, 2022, DOI: 10.1109/TC.2022.3177955.

[19] A. A. Hanif and M. Ilyas, "*Enhance the Detection of DoS and Brute Force Attacks within the MQTT Environment Through Feature Engineering and Employing an Ensemble Technique*," *Int. J. Artif. Intell. Appl.*, Vol. 15, No. 4, pp. 01–19, Jul. 2024, DOI: 10.5121/ijaia.2024.15401.

[20] A. F. Otoom, W. Eleisah, and E. E. Abdallah, "*Deep Learning for Accurate Detection of Brute Force Attacks on IoT Networks*," *Procedia Comput. SCI.*, Vol. 220, pp. 291–298, 2023, DOI: 10.1016/j.procs.2023.03.038.

[21] S. Ullah *et al.*, "*TNN-IDS: Transformer Neural Network-based Intrusion Detection System for MQTT-Enabled IoT Networks*," *Comput. Netw.*, Vol. 237, p. 110072, Dec. 2023, DOI: 10.1016/j.comnet.2023.110072.

[22] A. Omotosho, Y. Qendah, and C. Hammer, "*IDS-MA: Intrusion Detection System for IoT MQTT Attacks using Centralized and Federated Learning*," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, Torino, Italy: IEEE, Jun. 2023, pp. 678–688. DOI: 10.1109/COMPSAC57700.2023.00093.

[23] S. Chesney and K. Roy, "*AI Empowered Intrusion Detection for MQTT Networks*," in *2022 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*, Durban, South Africa: IEEE, Aug. 2022, pp. 1–6. DOI: 10.1109/icABCD54961.2022.9856124.

[24] F. Mosaiyebzadeh, L. G. Araujo Rodriguez, D. Macedo Batista, and R. Hirata, "*A Network Intrusion Detection System using Deep Learning against MQTT Attacks in IoT*," in *2021 IEEE Latin-American Conference on Communications (LATINCOM)*, Santo Domingo, Dominican Republic: IEEE, Nov. 2021, pp. 1–6. DOI: 10.1109/LATINCOM53176.2021.9647850.

[25] H. Hindy, E. Bayne, M. Bures, R. Atkinson, C. Tachtatzis, and X. Bellekens, "*Machine Learning based IoT Intrusion Detection System: An MQTT Case Study (MQTT-IoT-IDS2020 Dataset),*" in *Selected Papers from the 12th International Networking Conference*, Vol. 180, B. Ghita and S. Shiaeles, Eds., in Lecture Notes in Networks and Systems, Vol. 180. , Cham: Springer International Publishing, 2021, pp. 73–84. DOI: 10.1007/978-3-030-64758-2_6.

[26] I. Vaccari, G. Chiola, M. Aiello, M. Mongelli, and E. Cambiaso, "*MQTTset, a New Dataset for Machine Learning Techniques on MQTT*," *Sensors*, Vol. 20, No. 22, p. 6578, Nov. 2020, DOI: 10.3390/s20226578.